212 - 266

CHAPTER

6

Sequential Logic

In the previous five chapters we described the design procedure for combinatorial components. The main characteristic of all combinatorial components is the fact that their output values are computed entirely from their present input values. For any change in input values, the output values appear at the output pins with the delay Δ needed to compute those output values. Sequential components differ because they contain memory elements, which combinatorial components do not. As a result, the output values of sequential components are computed using both the present and past input values.

This dependence on the past input values requires the presence of memory elements in sequential circuits. The values stored in memory elements define the state of a sequential component. Thus any change in the input values at time t_i will change the state of the sequential component at time $t_i + \Delta_1$ and the output values at time $t_i + \Delta_2$.

As an example of a sequential circuit, let us consider the ring counter in a telephone answering machine, which counts the number of incoming rings and turns on the message recorder after four rings. Since the ring counter counts up to four, the counter memory must be able to store four digits: 0, 1, 2, and 3. When it is the number 0 that is stored in the memory, we say that the answering machine is in state 0, in which it expects the first ring. In states 1, 2, and 3, the answering machine will have received one, two, or three rings, respectively. When the counter is in state 3 and the fourth ring arrives, the recorder is activated. At this time, further rings will be disabled and the counter will return to state 0. In this example we see that the output of the ring counter depends not just on one ring but on the sequence of rings that occurred before the fourth ring arrives.

This is generally true for all sequential circuits. Their outputs depend on the sequence of input values that have occurred over a period of time. In fact, the term *sequential* comes from this dependence on an input-value sequence instead of just a current input value. As we have seen, this sequence of input values, or some derivative of it, is stored in the memory. Since this memory is always finite, the sequence size must always be finite, which means that the sequential logic can contain only a finite number of states, although the number of states could be quite large.

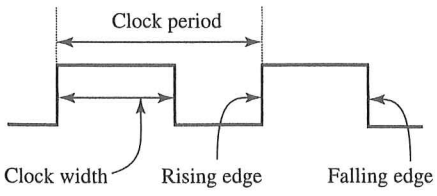


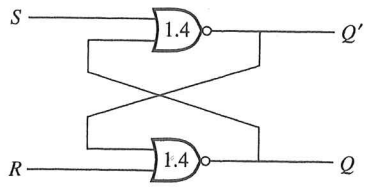
FIGURE 6.1
Clock signal.

In general, sequential circuits can be asynchronous or synchronous. **Asynchronous sequential circuits** change their state and output values whenever a change in input values occurs, whereas all **synchronous sequential circuits** change their states and output values at fixed points of time, which are specified by the rising or falling edge of a free-running **clock** signal. In Figure 6.1 we show the timing diagram and nomenclature for a typical clock signal. In this diagram you can see that the **clock period** is the time between successive transitions in the same direction, that is, between two rising or two falling edges. The reciprocal of the clock period is referred to as the **clock frequency**. Usually, the clock period is measured in nanoseconds (ns) and frequency is measured in megahertz (MHz). The **clock width** is the time during which the value of the clock signal is equal to 1. The ratio of clock width and clock period is referred to as the **duty cycle**. A clock signal is said to be **active high** if the state changes occur at the clock's rising edge or during the clock width. Otherwise, the clock signal is said to be **active low**.

In this chapter we use this nomenclature to discuss synchronous sequential logic. First, we introduce the basic storage elements that can store one bit of information, known as latches and flip-flops. We then give the analysis procedure for sequential logic and establish the finite-state-machine model used for the modeling of sequential logic. We also discuss the synthesis procedure for converting finite-state-machine descriptions into sequential logic schematics. In the course of this presentation, analysis and synthesis procedures are demonstrated on several practical examples.

6.1 SR LATCH

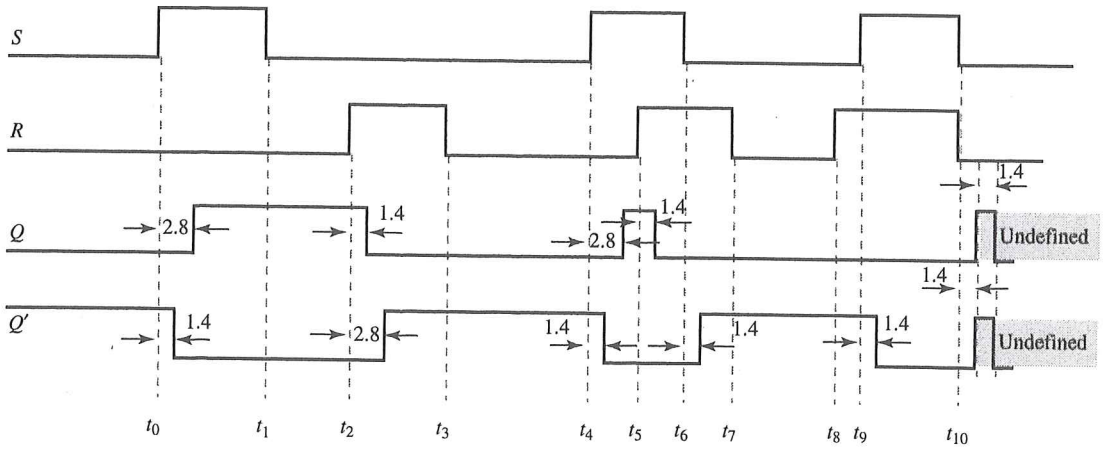
The simplest memory element in digital design is the SR latch, which consists of two cross-coupled NOR gates. As you can see in Figure 6.2(a), the **SR latch** has two input signals, the set signal S and the reset



(a) Logic schematic

S	R	Q	Q(next)	Q'(next)
0	0	0	0	1
0	0	1	1	0
0	1	X	0	1
1	0	X	1	0
1	1	X	0	0

(c) Truth table



(b) Timing diagram

FIGURE 6.2
SR latch (NOR implementation).

signal R; it also has two output signals, Q and Q'; finally, it has two states, a **set state** when $Q = 1$ ($Q' = 0$) and a **reset state** when $Q = 0$ ($Q' = 1$). As long as both input signals S and R are equal to 0, the SR latch persists in the same state. For example, if $Q = 1$, the output of the top NOR will be equal to 0, which, in turn, will keep the output of the bottom NOR at 1. Similarly, if $Q = 0$, the output of the top NOR will be equal to 1, which will make the output of the bottom NOR equal to 0.

If, however, the S input (R input) becomes equal to 1, the SR latch goes to the set (reset) state. This is demonstrated in the timing diagram in Figure 6.2(c). For example, when S equals 1 at t_0, the output Q' equals 0 at $t_0 + 1.4$ ns, which, in turn, forces Q to go to 1 at $t_0 + 2.8$ ns. Note that when S goes back to 0 at t_1, the SR latch stays in the set state. Similarly, the SR latch can be reset by asserting the reset signal R while keeping S at the value 0. For example, when R becomes 1 at t_2, the Q output becomes 0 at $t_2 + 1.4$ ns, forcing Q' to become 1 at $t_2 + 2.8$ ns. Note also that after R is disasserted at t_3, the latch will stay in the reset state.

If input signals S and R are both equal to 1, both output signals, Q and Q', must be equal to 0. If one of the input signals is disasserted earlier than the other, the latch will end up in the state forced by the signal that was disasserted later. This situation is demonstrated in Figure 6.2(b) at the point when S equals 1 at t_4. As you can see, the latch will follow the normal pattern and enter the set state at $t_4 + 2.8$ ns. Furthermore, when R is asserted at t_5, the Q output equals 0 at $t_5 + 1.4$ ns. Note that as long as both S and R equal 1, both outputs Q and Q' equal 0. However, when S is disasserted at t_6, the latch will enter the reset state at $t_6 + 1.4$ ns and stay in the reset state until S or R is reasserted again.

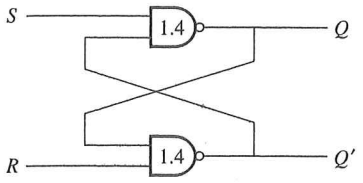
One problem inherent in the SR latch is the fact that if both S and R are disasserted at the same time, we cannot predict the latch output. In case both gates have exactly the same delay, both gates will become 1 at the same time and then 0 at the same time, and so on, thus oscillating forever, as shown in Figure 6.2(b). For example, when both input signals get disasserted at t_{10}, both NOR gates will become 1 at $t_{10} + 1.4$ ns. Since one of the inputs to each NOR gate is now equal to 1, both NOR gates will become equal to 0 at $t_{10} + 2.8$ ns. This oscillation, usually called a **critical race**, will continue with outputs of both gates equal to 1 at $t_{10} + 1.4 \times (2n + 1)$ and 0 at $t_{10} + 1.4 \times (2n)$ ns for any nonnegative integer n. When both NOR gates do not have exactly the same delay, one NOR gate may be slightly faster than the other. In such a case, the faster NOR gate will prevail and set its output to 1, forcing the other latch output to 0 at $t_{10} + 2.8$ ns. Therefore, when both input signals get disasserted at the same time, the next latch's state is undefined, since we do not know which condition will occur.

In order to avoid this indeterministric behavior, we must ensure that S and R signals are never disasserted at the same time. Unfor-

tunately, this rule is difficult to enforce because of unknown delays in the logic circuits generating the values of S and R. As a result, we must follow a much stricter rule when designing with SR latches; we must ensure that S and R signals should never be asserted at the same time.

From this analysis of SR-latch operation, we can construct the truth table of the latch behavior, shown in Figure 6.2(c). As this table shows, for each moment t_i, $Q(next)$ and $Q'(next)$ indicate the value of output Q at time $t_i + \Delta$, where Δ is equal to or greater than 2.8 ns. The table gives the output values for every combination of input values and the state of the SR latch.

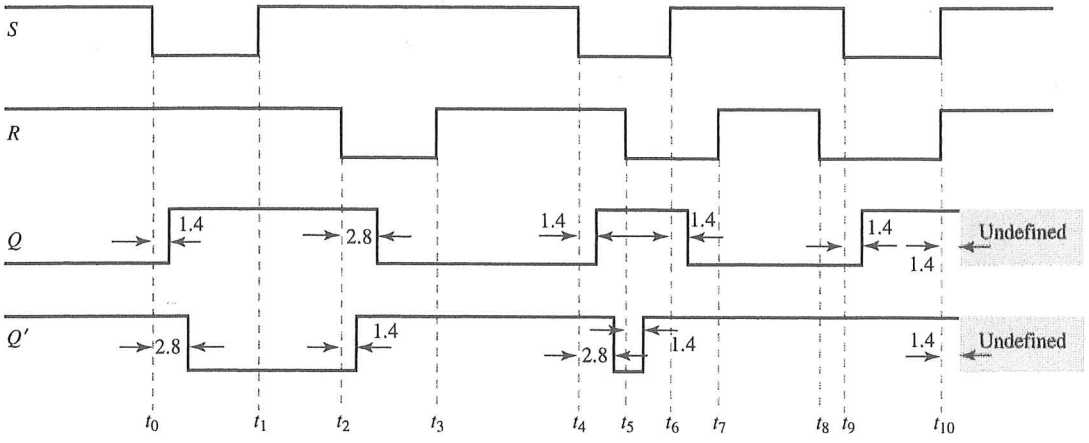
The SR latch can also be implemented with NAND gates. In this case, however, S and R inputs are normally equal to 1. Setting S or R to 0 will set or reset the latch, which is just the opposite of the NOR implementation, where asserting S or R caused the latch to be set or reset. The NAND implementation is shown in Figure 6.3(a), (b), and (c) for completeness.



(a) Logic schematic

S	R	Q	Q(next)	Q'(next)
0	0	X	1	1
0	1	X	1	0
1	0	X	0	1
1	1	0	0	1
1	1	1	1	0

(c) Truth table

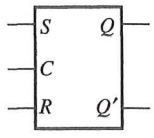


(b) Timing diagram

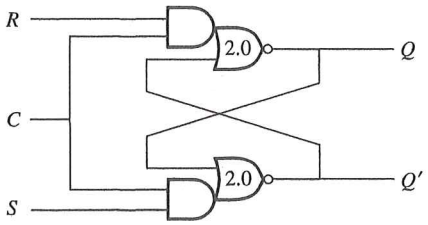
FIGURE 6.3
SR latch (NAND implementation).

6.2 GATED SR LATCH

The **gated SR latch** is similar to the SR latch, with one exception. As shown in Figure 6.4, this latch has a third control input, C, which enables or disables the operation of the SR latch. In practice, this means that when C equals 1, the gated SR latch operates as an SR latch. When $C = 0$, however, setting or resetting of the latch is disabled and the circuit persists in the preceding state. Consider, for example, the timing diagram in Figure 6.4(d), which demonstrates this behavior, showing that although the latch is in the reset state at t_0, it does not get set when S becomes 1. Note, however, that the setting of the latch is allowed at t_1, when C is asserted; the Q' output becomes 0 at $t_1 + 2.0$ ns and Q equals 1 at $t_1 + 4.0$ ns. Once C is disasserted again at t_2, the changes in



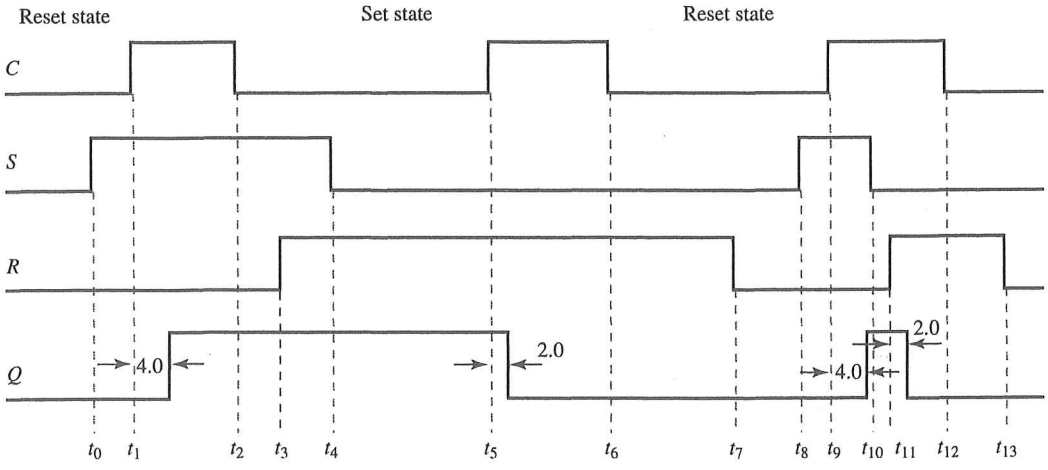
(a) Graphic symbol



(b) Logic schematic

C	S	R	Q	Q(next)
0	X	X	0	0
0	X	X	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	X	0
1	1	0	X	1
1	1	1	X	NA

(c) Truth table



(d) Timing diagram

FIGURE 6.4
Gated SR latch.

the values of the input signals S and R at t_3 and t_4 cannot affect the state of the latch. When C is asserted again at t_5, however, the value of the input signals is allowed to propagate through the latch. Therefore, the latch is reset at $t_5 + 2.0$ ns.

In general, we can see that the latch follows the changes in the input signals as long as C is equal to 1. For example, when C becomes 1 at t_9, S is equal to 1, so the latch enters set state ($Q = 1$) at $t_9 + 4.0$ ns. When S is disasserted at t_{10} and R is asserted at t_{11}, the latch enters the reset state ($Q = 0$) at $t_{11} + 2.0$ ns.

One precaution is relevant when working with the gated SR latch: The designer must make sure that the input signals do not change during the time window around the falling edge of the control input C. This window starts at setup time t_{setup} before the falling edge of C and ends with hold time t_{hold} after the falling edge of C. In Figure 6.4(d), for example, the following conditions must hold for the third control pulse:

$$t_{12} - t_{11} \geq t_{\text{setup}}$$
$$t_{13} - t_{12} \geq t_{\text{hold}}$$

Similar inequalities can be written for other falling edges of the control signal in Figure 6.4(d). In the majority of digital designs, the control input is connected to the system clock signal. For this reason, the gated SR latch is also frequently called a **clocked SR latch**.

6.3 GATED D LATCH

As indicated in Section 6.2, designers who are working with SR latches must ensure that inputs S and R never equal 1 at the same time. This nuisance can be removed by using D latches, which have only one input D. The **gated D latch** is constructed from a gated SR latch by connecting the D input to the S input and D' to the R input of the SR latch, as shown in Figure 6.5(b). By connecting D and D' to S and R inputs, we ensure that both S and R will never equal 1 at the same time. A D latch also has a C input, which enables the D latch as it did with the gated SR latch described above. When C equals 1, the output Q will assume the same value as the input D after a short delay time. Conversely, when C equals 0, the output Q maintains the last value of D established before the falling edge of the clock.

The functionality of a gated D latch is demonstrated in the timing diagram in Figure 6.5(d). Note that when C becomes equal to 1 at t_1, Q will become equal to D at $t_1 + 4.0$ ns. Similarly, when C becomes equal to 1 at t_4, Q will become equal to D at $t_4 + 2.0$ ns. Note also that

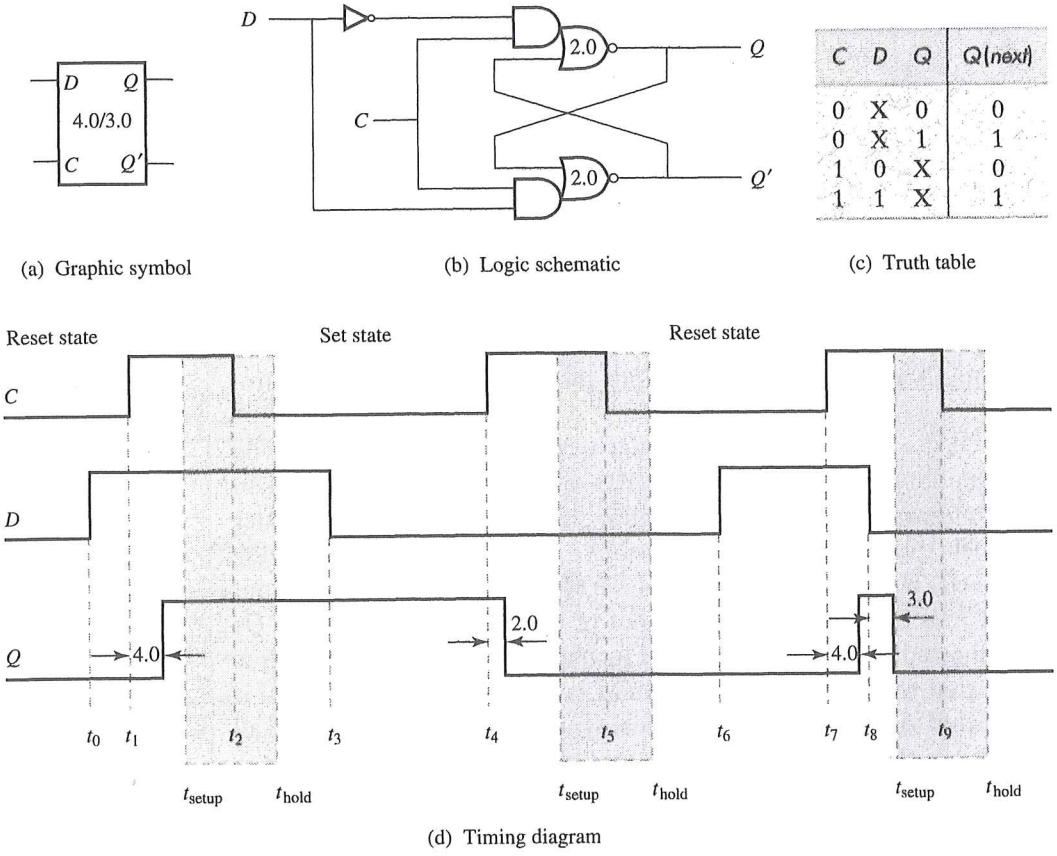


FIGURE 6.5
Gated D latch.

if D changes during the clock width, Q will follow the changes in input D as long as the changes occur before t_{setup} and after t_{hold}. t_{setup} and t_{hold} are times before and after the falling edge of the C signal during which the D input must be stable. The time interval between t_{setup} and t_{hold} is shown shaded in Figure 6.5(d). In the figure, for example, Q will become equal to 1 at $t_7 + 4.0$ ns, since D was equal to 1 when C became 1. On the other hand, when D changes to 0 at t_8, Q will follow at $t_8 + 3.0$ ns, as long as $t_9 - t_8 \geq t_{setup}$. In general, the gated D latch is easy to work with because the input signal D and the slightly delayed output signal Q always have the same value during the time period in which C is asserted.

As we explained in the preceding paragraph, output Q is delayed by 4.0 ns in the L-to-H transition and 3.0 ns in the H-to-L transition, which is indicated by 4.0/3.0 in the graphic symbol when used in larger schematics.

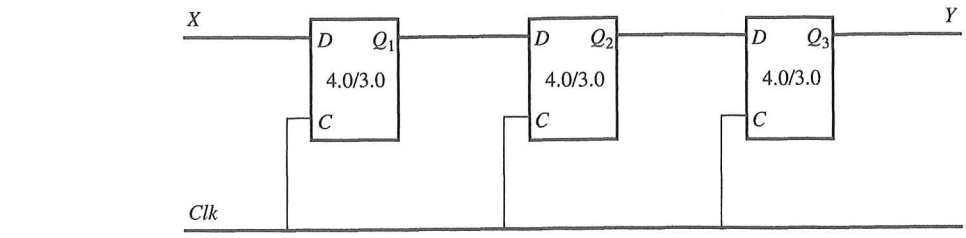
6.4 FLIP-FLOPS

As explained earlier, gated latches are simple memory elements that are enabled during the entire time interval, during which the control signal C equals 1. These latches are often called **level-sensitive latches** because they are enabled whenever the control signal is at level 1. At any point during that time, the latches will be transparent, in the sense that any input changes will propagate to the output with some small delay. These latches behave as memory elements only after the falling edge of the control signal, when they retain the state set by the last input value that occurred before the falling edge of the control signal.

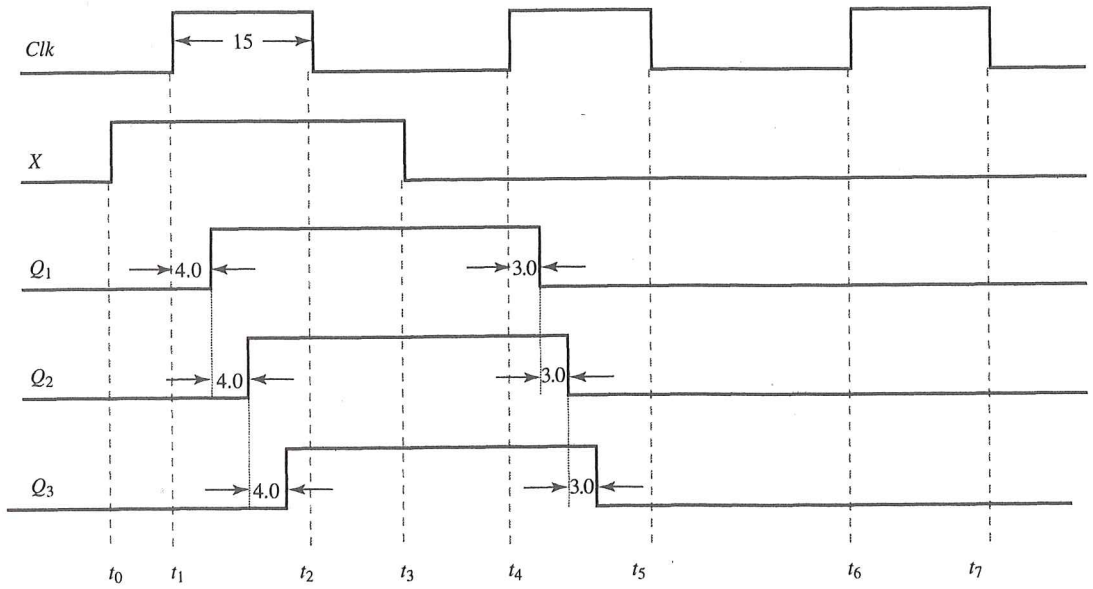
On the basis of this description you can see that designers must be very cautious when using these latches since long time intervals when latch is transparent can sometimes allow unwanted information to enter the latch. As an example, let's consider a 3-bit shift register consisting of three D latches, as shown in Figure 6.6(a). In this example the input signal X is connected to the D input of the first latch, its output Q_1 is connected to the D input of the second latch, and its output Q_2 is connected to the D input of the third latch. The control input C is connected to the system clock *Clk* that synchronizes the operation of all the latches. Ideally, this shift register should work in the following way: During each clock width, the X value will enter the first latch, the value in the first latch will be moved into the second latch, and the second value will be moved to the third latch.

As you can see in the detailed timing diagram in Figure 6.6(b), however, the information shift that actually occurs will not be exactly as desired. Let us assume, for example, that all latches are in the reset state ($Q_1 = Q_2 = Q_3 = 0$) and that the input signal X has a value of 1 during the first clock pulse and 0 afterward. In other words, the shift register should start with a content of 000 and contain 100, 010, and 001 after the first, second, and third clock pulses. In reality, however, if our shift register starts with a content of 000, it will be followed by 111, 000, and 000 after the first, second, and third clock pulses. In other words, the shift register has behaved as a single D latch which stores the value of the input signal X in each clock cycle.

Let's observe this behavior in even greater detail, assuming that the clock width $t_w = 15$ ns and that the input signal X becomes equal to 1 at t_0. When the clock signal enables the latching at t_1, the first latch will change to the set state ($Q_1 = 1$) at $t_1 + 4.0$ ns. Since the clock signal retains its value for another 11 ns, however, the second latch will switch to the set state at $t_1 + 8.0$ ns, as will the third latch at $t_1 + 12.0$ ns. After the first falling edge at t_2, then, the content of our shift register will be equal



(a) Logic schematic



(b) Timing diagram

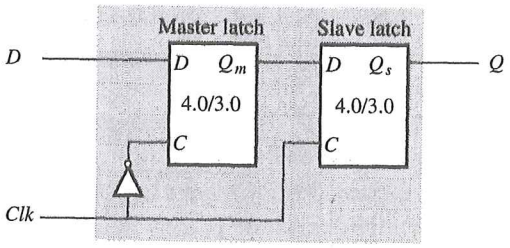
FIGURE 6.6
Erroneous shifting with D latches.

to 111. Similar behavior will be exhibited when the second and all succeeding clock pulses have enabled further latching in the shift register.

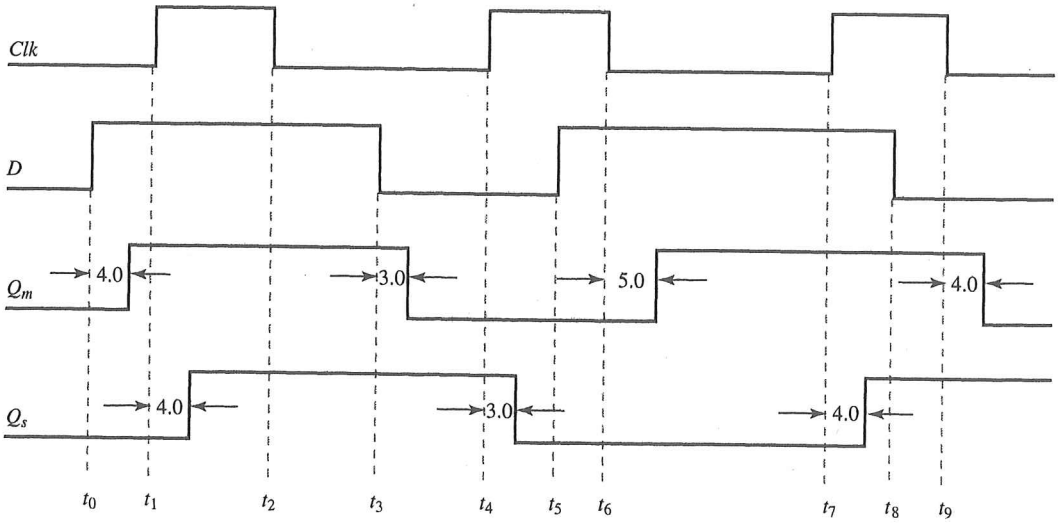
One possible idea to rectify this erroneous behavior is to shorten the clock width to one propagation delay. The difficulty in this approach, though, is that the delays in setting and resetting latches are not the same, which means that a clock width that works to set the latch may not work for resetting the latch, and vice versa. Furthermore, if the clock width is limited to a time less than the latch delay, the latch may not have time to catch the input value. In any case, the latch manufacturer cannot really guarantee exact delay values simply because of variations in the fabrication process. In this sense, we must keep in mind that the setting and resetting delays that we use are just expected delay values, whereas real delay values are normally distributed around these expected delays.

Given these constraints, there are two possible solutions to our problem: master-slave and edge-triggered flip-flops. **Master-slave flip-flops** are implemented using two latches, referred to as a master latch and a slave latch. As shown in Figure 6.7(a), the input to the master latch is the input to the flip-flop, while input to the slave latch is the output of the master latch. The output of the slave latch is the output of the entire flip-flop. Within the flip-flop, both the master and slave latches are driven by the same clock signal, *Clk*, the crucial difference being that the master latch will be enabled when the clock signal is equal to 0, and the slave latch will be enabled when the clock signal is equal to 1.

The advantage in using these flip-flops should be clear: Since master and slave latches are never enabled at the same time, the entire master–slave flip-flop is never transparent. When the clock signal *Clk* is 0, for example, only the master latch is enabled and its content is



(a) Logic schematic



(b) Timing diagram

FIGURE 6.7
Master-slave flip-flop.

transferred to the slave latch only after the clock signal becomes 1. Note that when the clock signal becomes 1, the master latch is disabled and its content does not change.

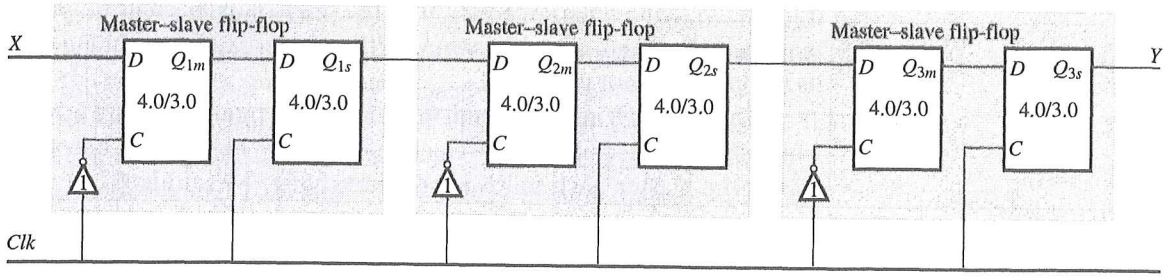
This is shown in more detail in the timing diagram of Figure 6.7(b), which shows three pulses of the clock signal *Clk*. When *D* input becomes 1 at t_0, the master latch follows the input change by setting $Q_m = 1$ at $t_0 + 4.0$ ns, since its control input *C* is equal to 1. This change does not propogate through the slave latch until the *Clk* equals 1 at t_1 and sets $Q_s = 1$ at $t_1 + 4.0$ ns. When *D* becomes 0 again at t_3, the master latch follows at $t_3 + 3.0$ ns but the change does not propagate through the slave latch until $t_4 + 3.0$ ns. When *D* changes again to 1 at t_5, the change is not accepted by the master latch until the *Clk* equals 0 at t_6. Thus Q_m becomes 1 at $t_6 + 5.0$ ns. Note that an extra 1-ns delay was added because the inverter driving the *C* input of the master latch. Furthermore, when the slave latch is enabled at t_7, Q_s equals 1 at $t_7 + 4.0$ ns. A similar change of the input *D* to 0 at t_8 is not registered by the master latch until $t_9 + 4.0$ ns and not propagated through the slave latch until the clock signal becomes 1 again.

As demonstrated in Figure 6.7, the value of the input *D* is captured into the master latch before the rising edge of the clock signal and transferred to the slave latch immediately after the same rising edge. For all practical purposes we can say that the value of *D* was captured on the rising edge of the clock signal.

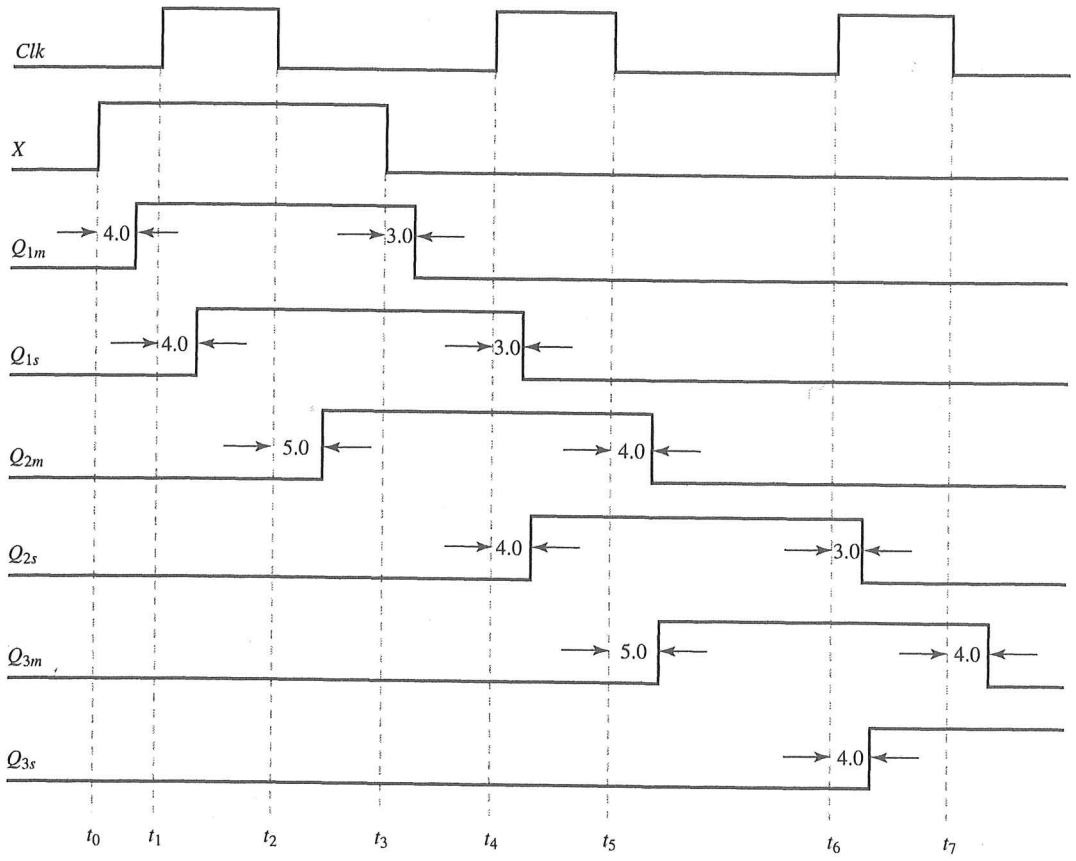
If we reconstruct the 3-bit shift register discussed above using master–slave flip-flops, we obtain the logic schematic shown in Figure 6.8(a), which corresponds to the timing diagram shown in Figure 6.8(b). Note that this new timing diagram contains the same *Clk* and input signals as in Figure 6.6(b) but has been altered so that for each flip-flop, we show two waveforms: the outputs of the master and slave latches, Q_{im} and Q_{is}, where $1 \leq i \leq 3$.

As the diagram shows, after the input signal *X* changes to 1 at t_0, only the master latch of the first flip-flop will be set ($Q_{1m} = 1$), at $t_0 + 4.0$ ns. Then, when the clock signal changes to 1 at t_1, the slave latch is set at $t_1 + 4.0$ ns. A little bit later, after the clock signal returns to 0 at t_2, the master latch of the second flip-flop is set ($Q_{2m} = 1$), at $t_2 + 5.0$ ns. Note that when the input signal *X* returns to 0 at t_3, the master latch of the first flip-flop is reset ($Q_{1m} = 0$). After the next rising edge of the clock, Q_{1s} is returned to 0 at $t_3 + 3.0$ and Q_{2s} is set to 1 at $t_4 + 4.0$. Similarly, at the third rising edge of the clock, Q_{2s} is reset and Q_{3s} is set.

As you can see, shift registers constructed from master–slave flip-flops shift their content by one position to the right on each rising edge of the clock signal. Each clock cycle therefore corresponds to one state



(a) Logic schematic

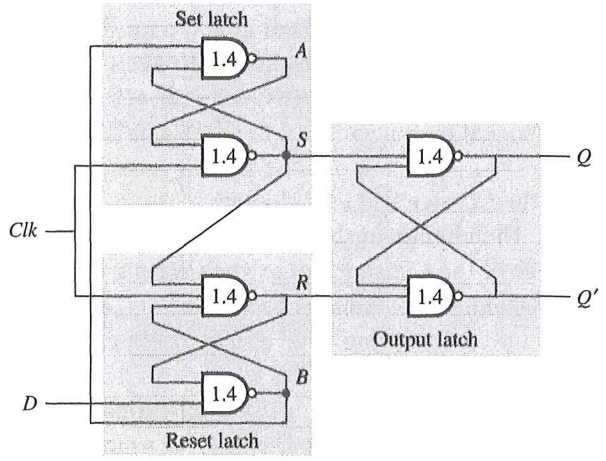


(b) Timing diagram

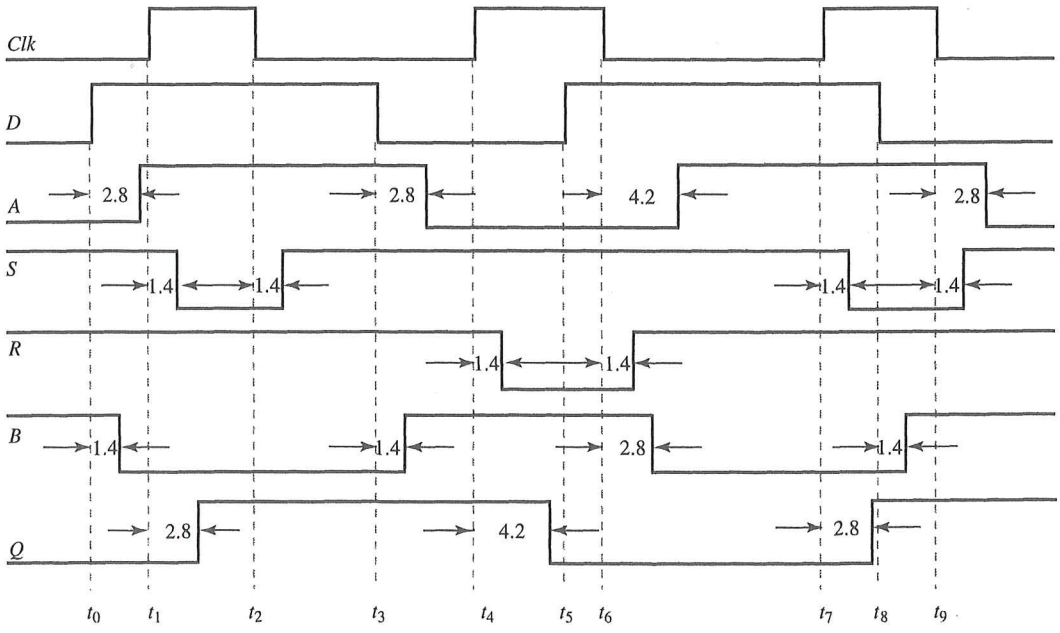
FIGURE 6.8
Shifting with master–slave flip-flops.

of the shift register, which [for the sequence of the input signal values shown in Figure 6.8(b)] goes from state 000 through states 100, 010, and 001, and finally, returns to 000.

As shown in Figure 6.9(a), an **edge-triggered flip-flop** is implemented with three interconnected SR latches: set, reset, and output. The set latch follows the changes in the *Clk* signal if *D* is equal to 1 at the rising edge of the *Clk* signal, while the reset latch follows the clock



(a) Logic schematic



(b) Timing diagram

FIGURE 6.9
Edge-triggered flip-flop.

signal if D is equal to 0 at the rising edge of the clock signal. In other words, signals A and B register changes in D as long as Clk is equal to 0. The detailed operation of the edge-triggered flip-flop is explained in the timing diagram shown in Figure 6.9(b).

While Clk equals 0, S and R signals are both 1, holding the output latch in its present state. When D changes at t_0, B goes to 0 at $t_0 + 1.4$ ns and A goes to 1 at $t_0 + 2.8$ ns. The output Q is not affected until Clk equals 1 at t_1, forcing S to become 0 at $t_1 + 1.4$ ns and Q to become 1 at $t_1 + 2.8$ ns. When Clk returns to 0 at t_2, S returns to 1 at $t_2 + 1.4$ ns, leaving Q at 1. When D goes to 0 at t_3, B becomes 1 at $t_3 + 1.4$ ns and A becomes 0 at $t_3 + 2.8$ ns.

After the rising edge of the Clk signal at t_4, R changes to 0 at $t_4 + 1.4$ ns and Q to 0 at $t_4 + 4.2$ ns. The change in the value of D at t_5 is not registered until the falling edge of the Clk signal at t_6 when R returns to 1 at $t_6 + 1.4$ ns and B becomes 0 at $t_6 + 2.8$ ns and A becomes 1 at $t_6 + 4.2$ ns. At $t_7 + 2.8$ ns, Q is set to 1 after the rising edge of Clk at t_7 and the lowering of S at $t_7 + 1.4$ ns.

Note again that subsequent change in D at t_8 does not propagate to the output Q but is captured when B becomes 1 at $t_8 + 1.4$ ns and A becomes 0 at $t_9 + 2.8$ ns. The value of Q will become 0 at the next rising edge of the Clk signal if D does not change after t_8. If D changes after t_8 and before the next rising edge, values of A and B signals will capture this change and propagate it to the output Q at the next rising edge of the Clk signal.

Since both master–slave and edge-triggered flip-flops change states only during positive clock transitions as we have demonstrated in Figures 6.7 and 6.9, we can define one state of the sequential circuit containing flip-flops to be a time interval between the two rising edges of the clock signal. The value of the sequential circuit in each of its states is defined by the content of all its flip-flops.

6.5 FLIP-FLOP TYPES

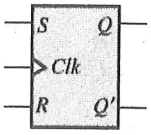
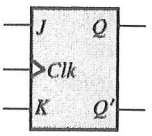
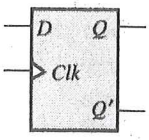
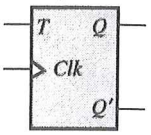
In Section 6.4 we showed how to construct a master-slave and edge-triggered flip-flop. Although there are many different ways to construct flip-flops, they all exhibit the following two characteristics: first, a flip-flop will change state only on the positive or negative edge of the clock signal, and second, its data inputs must not change after time t_{setup} before and until time t_{hold} after the triggering edge of the clock signal.

All flip-flops can be divided into four basic types: SR, JK, D, and T. They differ in the number of inputs and in the response evoked by

different values of input signals. The four types of flip-flops are defined in Table 6.1. Each of these flip-flops can be uniquely described by its graphical symbol, its characteristic table, its characteristic equations or excitation table. Graphical symbols specify the number and types of inputs and outputs. All flip-flops have output signals Q and Q'. All of them also have the clock signal input. The small triangle at the clock input indicates that the flip-flop is triggered by the rising edge of the clock signal. Conversely, a circle in front of the triangle would indicate that the flip-flop is triggered by the falling edge of the clock signal.

TABLE 6.1

Flip-Flop Types

FLIP-FLOP NAME	FLIP-FLOP SYMBOL	CHARACTERISTIC TABLE	CHARACTERISTIC EQUATION	EXCITATION TABLE
SR		S R | Q(next) 0 0 | Q 0 1 | 0 1 0 | 1 1 1 | NA	$Q(next) = S + R'Q$ $SR = 0$	Q Q(next) | S R 0 0 | 0 X 0 1 | 1 0 1 0 | 0 1 1 1 | X 0
JK		J K | Q(next) 0 0 | Q 0 1 | 0 1 0 | 1 1 1 | Q'	$Q(next) = JQ' + K'Q$	Q Q(next) | J K 0 0 | 0 X 0 1 | 1 X 1 0 | X 1 1 1 | X 0
D		D | Q(next) 0 | 0 1 | 1	$Q(next) = D$	Q Q(next) | D 0 0 | 0 0 1 | 1 1 0 | 0 1 1 | 1
T		T | Q(next) 0 | Q 1 | Q'	$Q(next) = TQ' + T'Q$	Q Q(next) | T 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

For the sake of simplicity we use throughout this book only flip-flops triggered by the rising edge. Each flip-flop has one or two data inputs that characterize the flip-flop and give it its name, as described earlier. The **SR flip-flop** has two inputs, S (set) and R (reset), that set or reset the flip-flop when asserted. In other words, when $S = 1$ and $R = 0$, the flip-flop output Q is set to 1 and when $S = 0$ and $R = 1$ it is reset to 0. Similar to the SR flip-flop, the **JK flip-flop** has two inputs, J and K, which set or reset the flip-flop when asserted. In addition, when both inputs J and K are asserted at the same time, the JK flip-flop changes its state. As mentioned before, the **D flip-flop** has one input D (data), which sets the flip-flop when $D = 1$ and resets it when $D = 0$. The **T flip-flop** has one input T (toggle), which forces the flip-flop to change states when T equals 1.

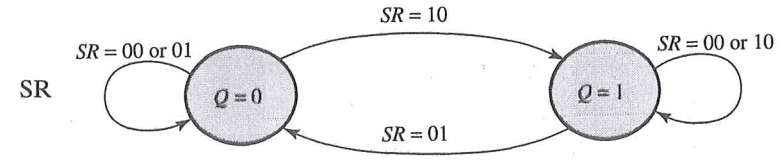
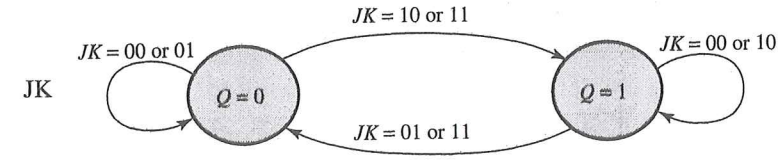
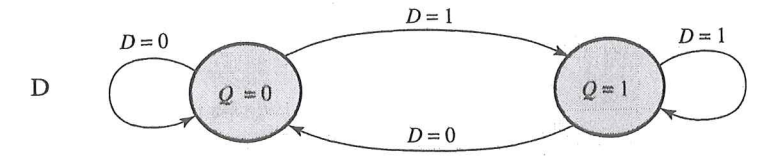
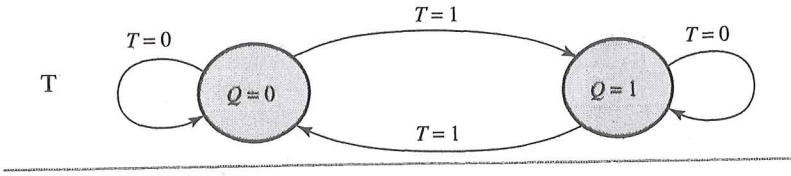
In the second column of Table 6.1 we see the **characteristic table**, which is a shorter version of the truth table, that gives for every set of input values and the state of the flip-flop before rising edge the corresponding state of the flip-flop after the rising edge of the clock signal. In the table, Q and $Q(next)$ designate the state of the flip-flop before and after the rising edge of the clock, respectively. The characteristic table is used during the analysis of sequential circuits when the value of flip-flop inputs are known and we want to find the value of the flip-flop output Q after the rising edge of the clock signal. As with any other truth table, we can use the map method to derive a **characteristic equation** for each flip-flop, which are shown in the third column of Table 6.1.

In the fourth column of the table, we show an **excitation table** which is used during the synthesis of sequential circuits. The excitation table is derived from the characteristic table by transposing input and output columns. It gives the value of flip-flop inputs that are necessary to change the flip-flop's present state to the desired next state after the rising edge of the clock signal.

In addition to graphical symbols, tables, or equations, flip-flops can also be described uniquely by means of state diagrams or state graphs, in which case each state would be represented by a circle, and a transition between states would be represented by an arrow. In Table 6.2, for example, the four types of flip-flops are described by this method. Note that each arrow is labeled with the values of its input signals, which will cause a transition from one state to the other. Note also that the same state can be both the source and the destination of a transition. Since transitions occur at the clock edge, each state can be thought of as a time interval between two rising edges of the clock signal.

In the table you can see that the state diagrams of all four flip-flops have the same number of states and transitions: Each flip-flop is in the set state when $Q = 1$ and in the reset state when $Q = 0$. Furthermore,

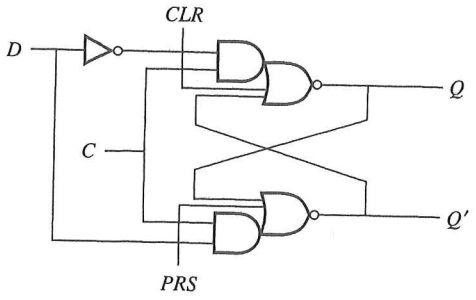
TABLE 6.2
State Diagrams for Various Flip-Flops

NAME	STATE DIAGRAM
SR	
JK	
D	
T	

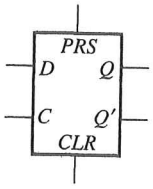
each flip-flop can move from one state to the other, or it can reenter the same state. The only difference between the four types lies in the values of input signals that cause these transitions. A state diagram is a very convenient way to visualize the operation of a flip-flop or even of large sequential components. In Section 6.7 we generalize these state diagrams, to define the finite-state-machine model used in the design of sequential logic.

Each flip-flop is usually available with and without **asynchronous inputs** that are used to preset and clear the flip-flops independently of other flip-flop inputs. These inputs are used to set flip-flops into the initial state for their standard operation. For example, after power is turned on, the state of each flip-flop is not predictable and thus we must use asynchronous inputs to set the flip-flops properly before the start of their synchronous operation. Preset and clear inputs are called asynchronous because they do not depend on the clock signal and therefore have precedence over all other synchronous operations. In other words, when

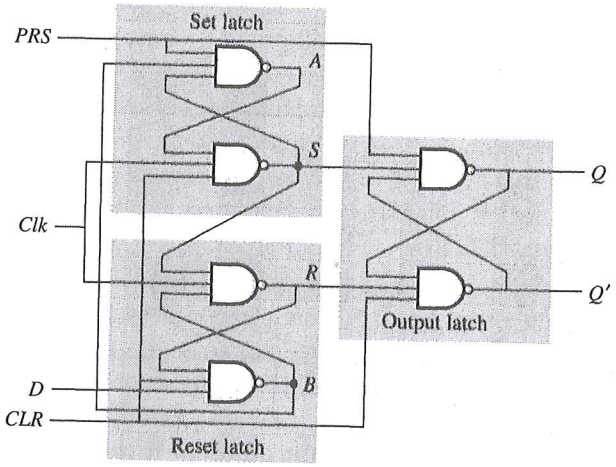
asynchronous inputs are active, the values on other flip-flop inputs are ignored. This can also be concluded from the logic schematics of a D latch and a D flip-flop with asynchronous inputs that are given in Figure 6.10.



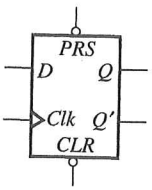
(a) D latch



(b) Graphic symbol



(c) D edge-triggered flip-flop



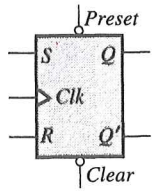
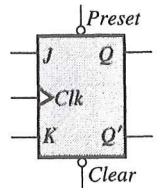
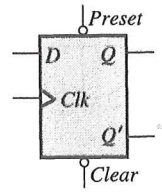
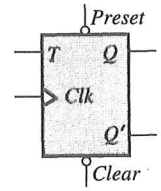
(d) Graphic symbol

FIGURE 6.10
Storage elements with asynchronous inputs.

As shown in Figure 6.10(a), the gated D latch with asynchronous inputs is the same gated D latch from Figure 6.5, with two additional inputs: *PRS* and *CLR*. When preset input (*PRS*) is equal to 1, the output Q' is equal to 0 and Q to 1. Conversely, when clear input (*CLR*) is equal to 1, the output Q is equal to 0. As long as one of the *PRS* or *CLR* inputs are 1, the flip-flop will persist in the states imposed by the asynchronous inputs. The graphic symbol for a D latch with asynchronous input is shown in Figure 6.10(b).

In contrast to the D latch, the edge-triggered flip-flop in Figure 6.10(c) is preset by the signal *PRS* when it is equal to 0 and cleared by the signal *CLR* when it is equal to 0. That fact that a low value of asynchronous signals affects the flip-flops is indicated by the small

TABLE 6.3
Graphic Symbols for Flip-Flops with Asynchronous Inputs

FLIP-FLOP NAME	FLIP-FLOP SYMBOL
SR	
JK	
D	
T	

circles in the graphic symbol shown in Figure 6.10(d). Note that the preset and clear signals force all the latches in the Figure 6.10(c) into proper states that correspond to $Q = 1$ and $Q = 0$, respectively. Since active-low preset and clear signals are more frequently found in practice, we will assume throughout this book that all asynchronous inputs are active-low. We show the graphic symbols for all types of flip-flops with active-low asynchronous inputs in Table 6.3.

6.6 ANALYSIS OF SEQUENTIAL LOGIC

Sequential logic is usually specified by means of a logic schematic, which incorporates the flip-flops given in Table 6.1 and the gates given in Tables 3.14, 3.15, and 3.16. Unfortunately, while such a logic schematic may clearly represent the structure of the sequential logic, it does not readily disclose its function, which can be a problem for the designer. During a product redesign, for example, designers usually want to modify the function of the product in order to add new features, or conversely, they may want to use new components and need to verify that the component replacements have not modified the function of the product. In either case, the designers will need to derive the functionality of the sequential logic schematic.

This process, called **analysis**, requires the designer to generate one or more functional descriptions, using state diagrams, state and output tables, and input and output Boolean equations. Once the functional description is derived, designers may also want to develop timing diagrams, which allow them to check their predictions against simulated results. In the following section we demonstrate the complete analysis procedure with several examples.

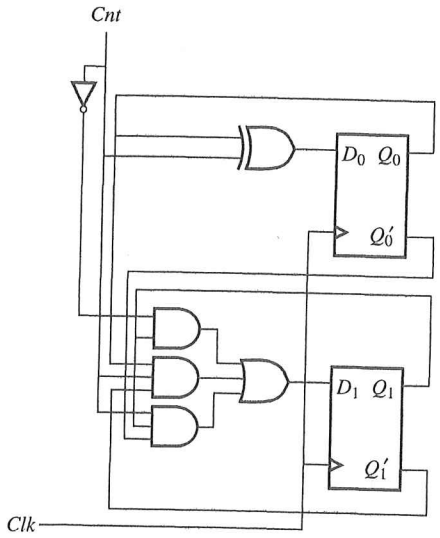
EXAMPLE 6.1 Modulo-4 counter

PROBLEM Derive the state table and state diagram for the sequential circuit represented by the schematic in Figure 6.11(a).

SOLUTION The first step in our analysis is to derive Boolean expressions for the inputs of each flip-flop in the schematic, in terms of external input *Cnt* and the flip-flop outputs Q_1 and Q_0. Since there are two D flip-flops in our example, we derive two expressions for D_1 and D_0:

$$D_0 = Cnt \oplus Q_0 = Cnt'Q_0 + CntQ_0' \tag{6.1}$$

$$D_1 = Cnt'Q_1 + CntQ_1'Q_0 + CntQ_1Q_0' \tag{6.2}$$



(a) Logic schematic

$$D_0 = Cnt \oplus Q_0 = Cnt'\, Q_0 + Cnt\, Q_0'$$
$$D_1 = Cnt'\, Q_1 + Cnt\, Q_1' Q_0 + Cnt\, Q_1 Q_0'$$

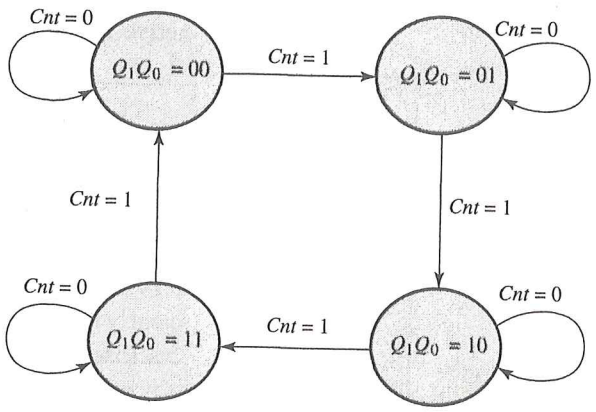
(b) Excitation equation

$$Q_0(next) = D_0 = Cnt'\, Q_0 + Cnt\, Q_0'$$
$$Q_1(next) = D_1 = Cnt'\, Q_1 + Cnt\, Q_1' Q_0 + Cnt\, Q_1 Q_0'$$

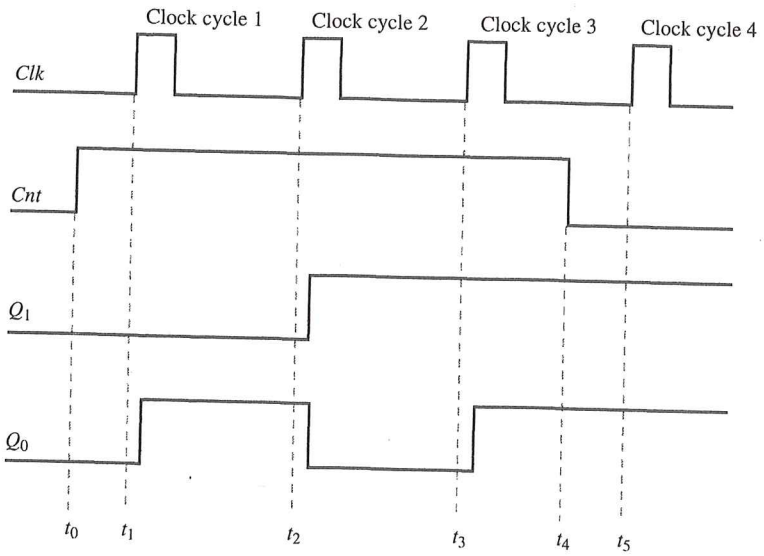
(c) Next-state equations

PRESENT STATE	NEXT STATE	
Q_1Q_0	$Q_1(next)\,Q_0(next)$	
	$Cnt = 0$	$Cnt = 1$
0 0	0 0	0 1
0 1	0 1	1 0
1 0	1 0	1 1
1 1	1 1	0 0

(d) Next-state table



(e) State diagram



(f) Timing diagram

FIGURE 6.11
Analysis of a sequential circuit.

These Boolean expressions are called **excitation equations** since they represent the inputs to the flip-flops, which, in turn, will determine the state of the sequential circuit in the

next clock cycle. After the excitation equations are derived, we can derive the **next-state equations** by converting these excitation equations into flip-flop characteristic equations. In the case of D flip-flops, $Q(next) = D$. Therefore, the next-state equations equal the excitation equations:

$$Q_1(next) = Cnt'Q_0 + CntQ_0' \tag{6.3}$$

$$Q_0(next) = Cnt'Q_1 + CntQ_1'Q_0 + CntQ_1Q_0' \tag{6.4}$$

We now convert these next-state equations into tabular form called the **next-state table**, in which each row corresponds to a state of the sequential circuit and each column represents one set of input values. In general, each state of a sequential circuit is defined by the binary values stored in its flip-flops. In this case, then, since we have only two flip-flops, the number of possible states is four—that is, Q_1Q_0 can be equal to 00, 01, 10, or 11. These values are shown as present states in Figure 6.11(d).

In the next-state part of the table, each entry defines the value of the sequential circuit in the next clock cycle after the rising edge of the *Clk*. Since this value depends on the present state and the value of the input signals, the next-state table will contain one column for each assignment of binary values to the input signals. In our example, since there is only one input signal, *Cnt*, the next-state table shown in Figure 6.11(d) has only two columns, corresponding to $Cnt = 0$ and $Cnt = 1$. Note that each entry in the next-state table indicates the value of the flip-flops in the next state if their value in the present state is in the row header and the input values in the column header. Each of these next-state values has been computed from next-state equations (6.3) and (6.4).

Instead of a next-state table, however, we could use a state diagram to represent the behavior of the sequential circuit. A state diagram is basically a pictorial representation of the next-state table. It has exactly one node, identified by a circle, for each present state in the next-state table. It also has exactly one directed arc going out of each state for each entry in the table. Each arc is labeled with the values of the input signals that cause the transition from the present state (the source of the arc) to the next state (the destination of the arc).

In general, the number of states in a next-state table or a state diagram will equal 2^m, where m is the number of flip-

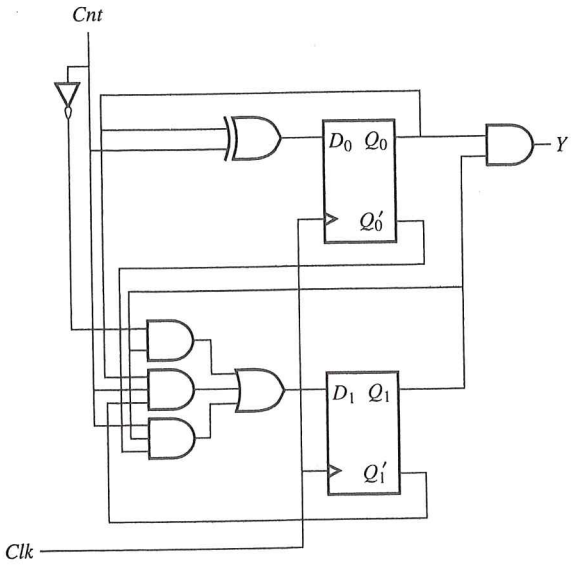
flops; similarly, the number of arcs will equal $2^m \times 2^k$, where k is the number of binary input signals. In the state diagram in Figure 6.11(e), there must be four states and eight transitions. Following these transition arcs, we can see that as long as $Cnt = 1$, the sequential circuit visits the states in the following sequence: 0, 1, 2, 3, 0, 1, 2, ... On the other hand, when $Cnt = 0$, the circuit stays in its present state until *Cnt* changes to 1, at which point the counting continues. Since this sequence is characteristic of modulo-4 counting, we can conclude that the sequential circuit represented in Figure 6.11(a) is a modulo-4 counter with one control signal, *Cnt*, which enables counting when $Cnt = 1$ and disables it when $Cnt = 0$.

In Figure 6.11(f) we show a timing diagram, representing four clock cycles, which enables us to observe the behavior of the counter in greater detail. In this timing diagram we have assumed that *Cnt* is asserted in clock cycle 0 at time t_0 and is disasserted in clock cycle 3 at time t_4. We have also assumed that the counter is in state $Q_1Q_0 = 00$ in the clock cycle 0. Note that on the clock's rising edge, at t_1, the counter will go to state $Q_1Q_0 = 01$ with a slight propagation delay; in cycle 2, after t_2, to $Q_1Q_0 = 10$; and in cycle 3, after t_3, to $Q_1Q_0 = 11$. Since *Cnt* becomes 0 at t_4, we know that the counter will stay in state $Q_1Q_0 = 11$ in the next clock cycle. To verify the behavior of a sequential circuit completely, we must construct timing diagrams for all possible sequences of input values.

In Example 6.1 we demonstrated the analysis of a sequential circuit that has no outputs by developing a next-state table and a state diagram which describe only the states and the transitions from one state to the next. In the next example we complicate our analysis by adding output signals, which means that we have to upgrade the next-state table and the state diagram to identify the value of output signals in each state. Such a sequential circuit, in which the output values depend solely on its present state, is usually referred to as a **state-based** or **Moore-type sequential circuit**.

EXAMPLE 6.2 State-based modulo-4 counter

PROBLEM Derive the next state, the output tables, and the state diagram for the sequential circuit given by the schematic shown in Figure 6.12(a).



(a) Logic schematic

$$D_0 = Cnt \oplus Q_0 = Cnt'\, Q_0 + Cnt\, Q_0'$$
$$D_1 = Cnt'\, Q_1 + Cnt\, Q_1' Q_0 + Cnt\, Q_1 Q_0'$$

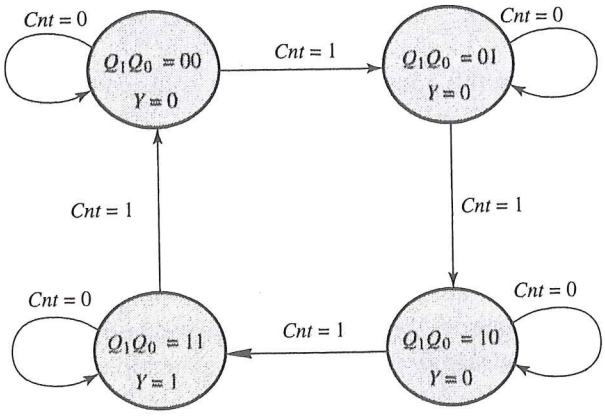
(b) Excitation equation

$$Q_0(next) = D_0 = Cnt'\, Q_0 + Cnt\, Q_0'$$
$$Q_1(next) = D_1 = Cnt'\, Q_1 + Cnt\, Q_1' Q_0 + Cnt\, Q_1 Q_0'$$
$$Y = Q_1 Q_0$$

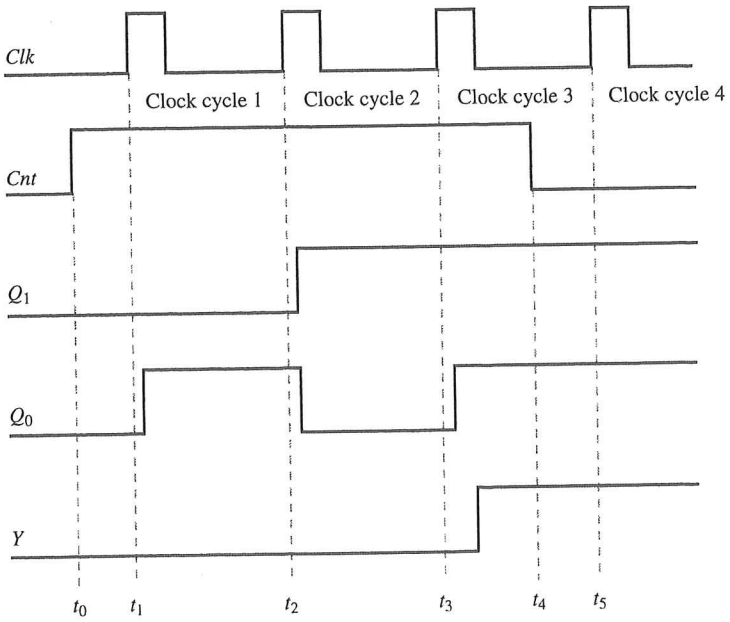
(c) Next-state and output equations

PRESENT STATE	NEXT STATE		OUTPUTS
Q_1Q_0	$Q_1(next)\,Q_0(next)$		Y
	$Cnt = 0$	$Cnt = 1$	
0 0	0 0	0 1	0
0 1	0 1	1 0	0
1 0	1 0	1 1	0
1 1	1 1	0 0	1

(d) Next-state and output table



(e) State diagram



(f) Timing diagram

FIGURE 6.12
Analysis of a state-based modulo-4 counter.

SOLUTION In the figure the input combinatorial logic is the same as in Example 6.1, so the excitation and next-state equations will be the same as in Example 6.1. For the sake of completeness, we have shown these equations in Figure 6.12(b) and (c). In addition, however, we have computed the output equation,

$$Y = Q_1 Q_0 \tag{6.5}$$

As this equation shows, the output Y will equal 1 when the counter is in state $Q_1Q_0 = 11$, and it will stay 1 as long as the counter stays in that state. Since the output Y depends only on the present state of the sequential circuit, we can add one extra column to the state table to convert it to a next-state/output table. We use Equation (6.5) to determine the values to be placed in the output column, as shown in Figure 6.12(d). In general, we add one column for each output signal to convert a next-state table to a next-state/output table.

In Figure 6.12(e) we show how the value of the output signal can be added to each state in the state diagram, and in Figure 6.12(f) how the output Y can be also added to the timing diagram. In Figure 6.12(f), note that the counter will reach the state $Q_1Q_0 = 11$ only in the third clock cycle, so the output Y will equal 1 after Q_0 changes to 1. Since counting is disabled in the third clock cycle, the counter will stay in the state $Q_1Q_0 = 11$ and Y will stay asserted in all succeeding clock cycles until counting is enabled again.

In Example 6.2 we have analyzed the sequential logic of a state-based circuit, in which the output signal values were dependent on the state of the sequential circuits but not on the input signal values. In the case of an **input-based** or **Mealy-type sequential circuit**, however, the output values are dependent on the input values as well as the state of the circuit. In analyzing such a circuit, the state and output tables must be modified in order to describe this input-based sequential circuit, which means that every entry in the next-state table will represent the next-state and the output value, separated by a slash (/).

In a state diagram of this type of sequential circuit, the output is not associated with the state but with the transition arc. In this case, each arc is labeled with both the input values that move the circuits from the present state to the next state, and the output values, which

correspond to the input-signal values in the present state. This modification is demonstrated by the following example.

EXAMPLE 6.3 Input-based modulo-4 counter

PROBLEM Derive the state/output table and the state diagram for the sequential circuit given by the schematic shown in Figure 6.13(a).

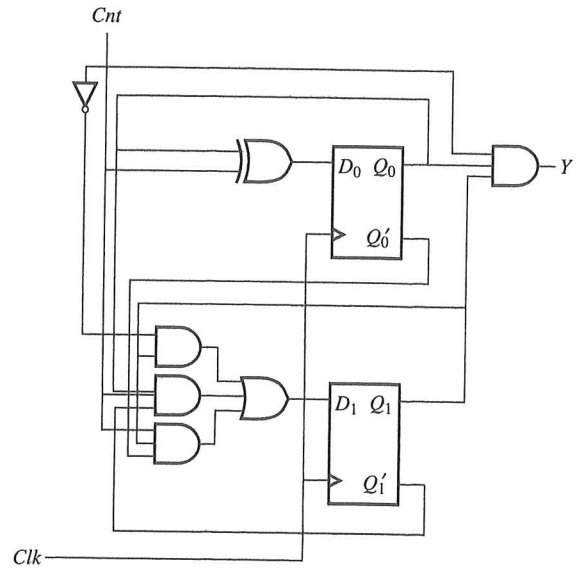
SOLUTION As you can see from Figure 6.13, this circuit differs from the circuit in Example 6.2 only in the expression of its output, which now depends on the value of the input signal *Cnt*. From the schematic we can see that the value Y equals 1 when the counter is in state $Q_1Q_0 = 11$ and counting. Thus

$$Y = CntQ_1Q_0 \tag{6.6}$$

Given this output equation, the state table from Example 6.1 must be modified so that each entry represents the next-state and output values; that is, the next-state value must be computed from the next-state equations, and the output values must be computed from the output equations, shown in Figure 6.13(c). This modified state/output table is shown in Figure 6.13(d).

The state diagram of this input-based circuit will have four states, as before. However, unlike the diagram of the state-based circuit, it now labels each arc with the input and output values in the form X/Y. The label X/Y can be interpreted as follows: If the input signal value at the next positive edge of the clock signal is equal to X, the circuit will transition to the state pointed to by the arc in the next clock cycle. Furthermore, its output during the present clock cycle will be equal to Y as long as the input signal value is equal to X. In Figure 6.13(e), for example, the arc between states $Q_1Q_0 = 01$ and $Q_1Q_0 = 10$ has been labeled $Cnt = 1/Y = 0$, which means that if $Cnt = 1$ during the cycle in which $Q_1Q_0 = 01$, the counter's output will be $Y = 0$ in the present clock cycle and in the next clock cycle the counter will be in state $Q_1Q_0 = 10$. This situation is demonstrated in clock cycle 1 in the timing diagram presented in Figure 6.13(f).

Thus the counter will reach state $Q_1Q_0 = 10$ in clock cycle 2, in which the output signal $Y = 0$. In clock cycle 3,



(a) Logic schematic

$$D_0 = Cnt \oplus Q_0 = Cnt' Q_0 + Cnt Q_0'$$
$$D_1 = Cnt' Q_1 + Cnt Q_1 Q_0 + Cnt Q_1 Q_0'$$

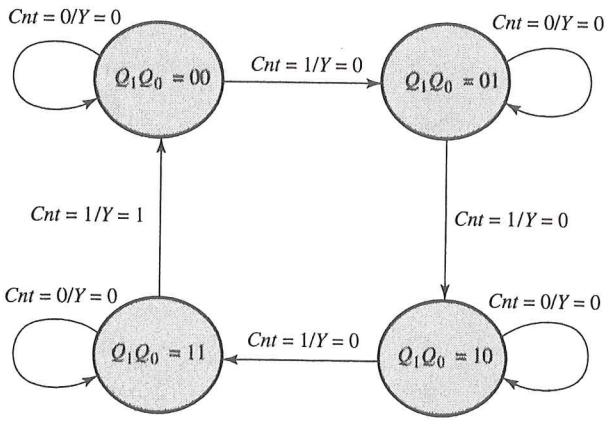
(b) Excitation equation

$$Q_0(next) = D_0 = Cnt' Q_0 + Cnt Q_0'$$
$$Q_1(next) = D_1 = Cnt' Q_1 + Cnt Q_1' Q_0 + Cnt Q_1 Q_0'$$
$$Y = Cnt Q_1 Q_0$$

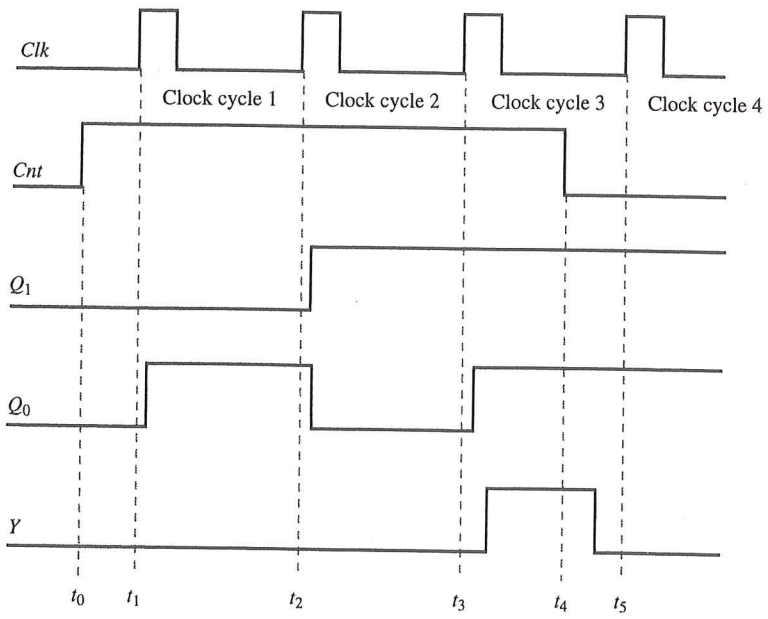
(c) Next-state and output equations

PRESENT STATE	NEXT STATE/OUTPUTS	
Q_1Q_0	$Q_1(next)\,Q_0(next)/Y$	
	$Cnt = 0$	$Cnt = 1$
0 0	0 0/0	0 1/0
0 1	0 1/0	1 0/0
1 0	1 0/0	1 1/0
1 1	1 1/0	0 0/1

(d) Next-state and output table



(e) State diagram



(f) Timing diagram

FIGURE 6.13
Analysis of input-based modulo-4 counter.

the counter will be in state $Q_1Q_0 = 11$ and the output signal Y will equal 1. At t_4, however, the output Y will become equal to 0 because the input signal *Cnt* now equals 0 even though the counter is still in state $Q_1Q_0 = 11$. In contrast, note that the output Y in Example 6.2 maintained a value of 1 in clock cycle 3, because its value was dependent only on the counter state, which had not changed. ■

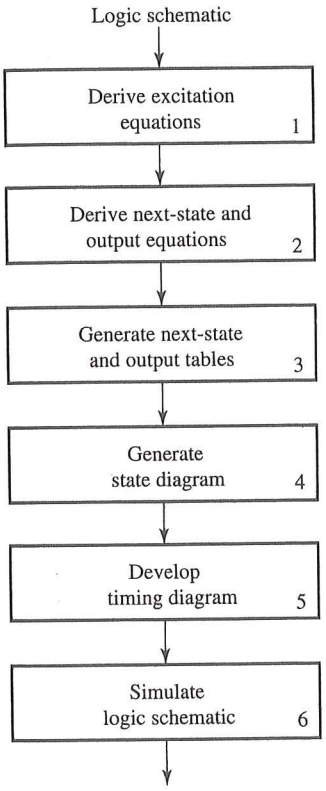


FIGURE 6.14
Analysis procedure for sequential circuits.

On the basis of the tasks applied in Examples 6.1 to 6.3, you should now have a general sense of the various tasks involved in an analysis of sequential circuits. As indicated in Figure 6.14, we start with a logic schematic from which we can derive excitation equations for each flip-flop input. Then, to obtain next-state equations, we insert the excitation equations into characteristic equations. The output equations can also be derived from the schematic, and once we have our output and next-state equations, we can generate next-state and output tables as well as state diagrams. When we reach this stage, we use either the tables or the state diagram to develop a timing diagram, which can then be verified through simulation. We can obtain timing waveforms through simulation on two different levels. On the structural level, we can describe the logic schematic in a simulation language and run the simulation for each set of state and input values using a library of gate and flip-flop models. On the behavioral level, we can describe state diagrams or next-state/output tables in a simulation language and run that through a simulator. In this case we can verify the sequential circuit's functionality but not its real delays, since the state diagram and tables do not contain any information about circuit implementation.

6.7 FINITE-STATE-MACHINE MODEL

At this point we have analyzed several sequential circuits and described them by means of next-state and output Boolean equations, state and output tables, and state diagrams, each of which completely specifies any sequential circuit. All of these descriptions are based on the model of the finite-state machine which we describe in this section.

The finite-state machine (FSM) can be defined abstractly as the quintuple

$$< S, I, O, f, h >$$

where S, I, and O represent a set of states, set of inputs, and a set of outputs, respectively, and f and h represent the next-state and output functions. The next-state function f is defined abstractly as a mapping $S \times I \rightarrow S$. In other words, f assigns to every pair of state and input symbols another state symbol. The FSM model assumes that time is divided into uniform intervals and that transitions from one state to another occur only at the beginning of each time interval. Therefore, the next-state function f defines what the state of the FSM will be in the next time interval given the state and input values in the present interval.

The output function h determines the output values in the present state. There are two different types of finite-state machine, which correspond to two different definitions of the output function h. One type is a **state-based** or **Moore FSM**, for which h is defined as a mapping $S \rightarrow O$. In other words, an output symbol is assigned to each state of the FSM. The other type is an **input-based** or **Mealy FSM**, for which h is defined as the mapping $S \times I \rightarrow O$. In this case, an output symbol in each state is defined by a pair of state and input symbols.

According to our definition, each set S, I, and O may have any number of symbols. However, in reality we deal only with binary variables, operators, and memory elements. Therefore, S, I, and O must be implemented as a cross-product of binary signals or memory elements, whereas functions f and h are defined by Boolean expressions that will be implemented with logic gates.

Thus the FSM can model any sequential circuit with k input signals $A_1, \ldots, A_k$, m flip-flops $Q_1, \ldots, Q_m$, and n output signals $Y_1, \ldots, Y_n$, as shown in Figure 6.15. For such a sequential circuit, S, I, and O are cross products of flip-flops or signals as follows:

$$S = Q_1 \times Q_2 \times \cdots \times Q_m$$
$$I = A_1 \times A_2 \times \cdots \times A_k$$
$$O = Y_1 \times Y_2 \times \cdots \times Y_n$$

Thus each element in S, I, and O is represented by a string of 1's and 0's.

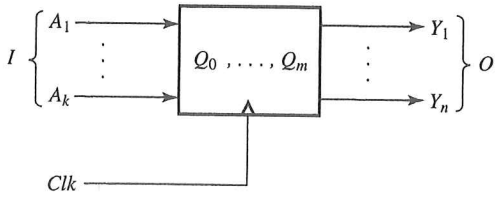


FIGURE 6.15
FSM model of general sequential circuit.

The clock signal defines the time intervals, called clock cycles. If the definition above is adopted, we can model the modulo-4 counter from Example 6.2 as a state-based FSM, where $S = \{s_0, s_1, s_2, s_3\}$, $I = \{i_0, i_1\}$, $O = \{o_0, o_1\}$, and f and h are given by the table in Figure 6.16.

PRESENT STATE	NEXT STATE ($S \times I \rightarrow S$)		OUTPUT ($S \rightarrow O$)
	i_0	i_1	
s_0	s_0	s_1	o_0
s_1	s_1	s_2	o_0
s_2	s_2	s_3	o_0
s_3	s_3	s_0	o_1

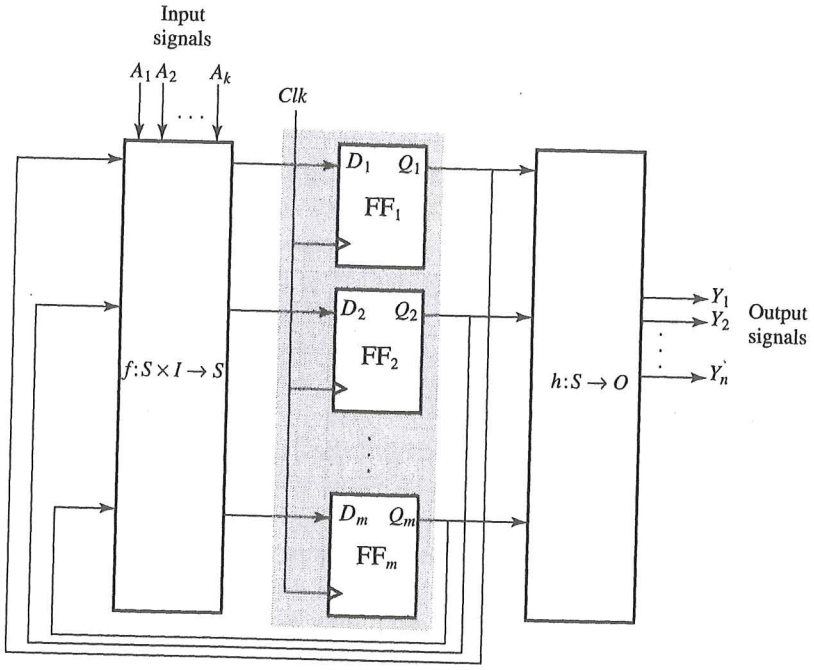
FIGURE 6.16
FSM model of modulo-4 counter from Example 6.2.

Similarly, we can model the modulo-4 counter from Example 6.3 as an input-based FSM, in which S, I, O are defined as they were above and f and h are given by the table in Figure 6.17.

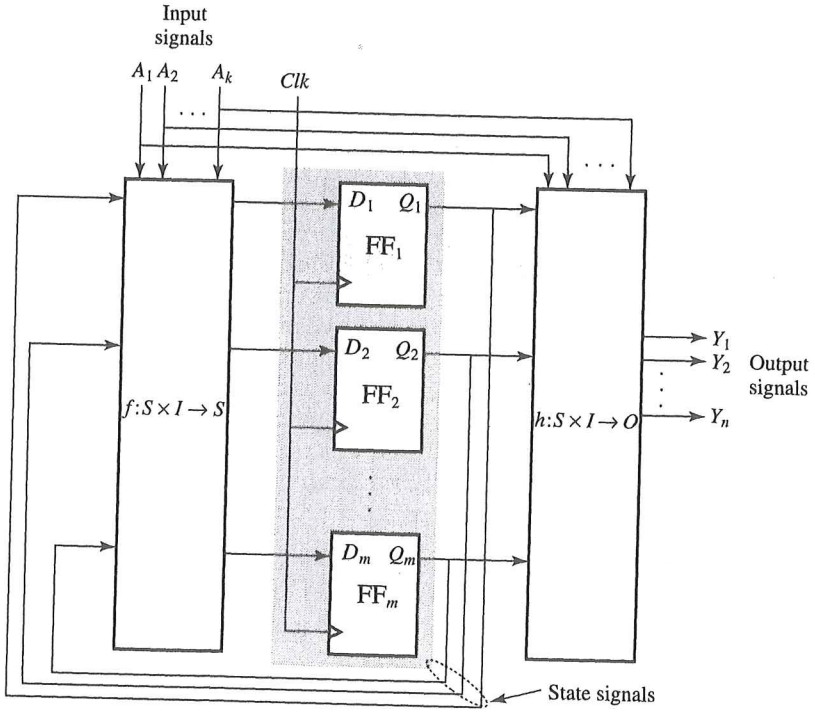
PRESENT STATE	NEXT STATE ($S \times I \rightarrow S$) / OUTPUT ($S \times I \rightarrow O$)	
	i_0	i_1
s_0	s_0/o_0	s_1/o_0
s_1	s_1/o_0	s_2/o_0
s_2	s_2/o_0	s_3/o_0
s_3	s_3/o_0	s_0/o_1

FIGURE 6.17
FSM model of modulo-4 counter from Example 6.3.

Each FSM model can be implemented with flip-flops and logic gates. The content of the flip-flops defines the state of the FSM, while f and h are implemented as combinatorial logic. The general logic block diagrams for state- and input-based FSMs are shown in Figure 6.18. These block diagrams are used in subsequent sections as templates for the synthesis of sequential logic.



(a) State-based



(b) Input-based

FIGURE 6.18
Finite-state-machine implementations.

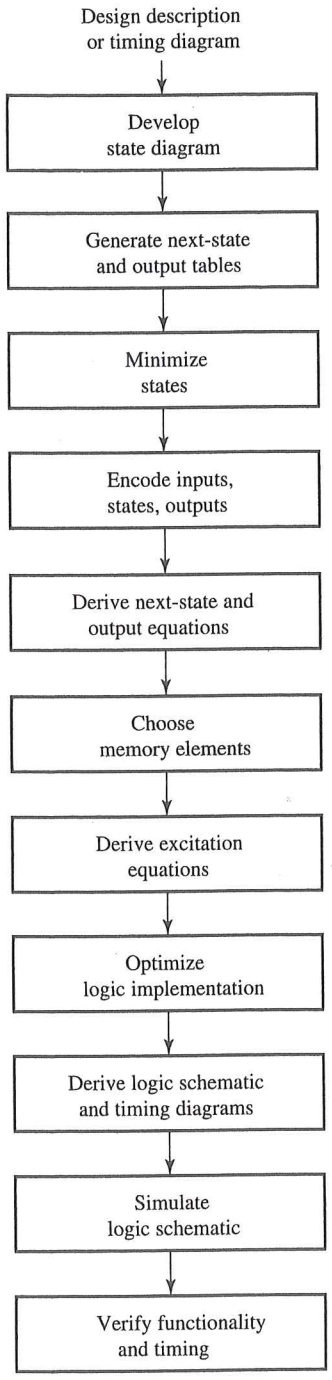


FIGURE 6.19
Synthesis procedure for FSM models.

6.8 SYNTHESIS OF SEQUENTIAL LOGIC

In previous sections we have demonstrated how to analyze various sequential circuits. In this section we turn to the synthesis procedure, which is the inverse of the task of analysis, insofar as analysis starts with the implementation and discovers the function or behavior of the sequential circuit, whereas synthesis starts with a behavioral description and generates an implementation.

A general description of the synthesis procedure is shown in Figure 6.19. As you can see, this process usually starts with an English-language description, possibly augmented with one or more timing diagrams, or sometimes with a more formal description presented in a language such as the IEEE-standard hardware description language known as VHDL. Given this description, the first task in the synthesis procedure consists of developing the state diagram and converting it into next-state and output tables. Next, we try to reduce the number of states by removing redundant states and merging equivalent states, since a smaller number of states will require fewer flip-flops. Ultimately, since each state must be expressed as an n-tuple of its flip-flop values, we need to assign different n-tuples to the various states. This procedure, called encoding, if done properly, will simplify the input and output logic. After encoding, we can generate the binary form of the next-state and output equations.

The next task in the synthesis consists of choosing flip-flop types, keeping in mind the fact that different types require different amounts of logic for implementation of the next-state functions. Having chosen the flip-flop types, we are ready to derive excitation equations for each flip-flop input, at which point we can optimize the logic implementation of the excitation and output equations and draw a logic schematic that serves as a basis for the generation of a timing diagram.

The last task in this series consists of simulating the logic schematic, then comparing the output to the timing diagram derived and checking whether the simulation actually implements the behavior we intended to produce. In the following sections we demonstrate this synthesis procedure in detail by working through a comprehensive example.

6.9 FSM MODEL CAPTURE

The design of sequential logic starts with the generation of a state diagram and/or next-state and output tables. Initially, these diagrams and tables must be derived from a natural language description of the behavior of the sequential circuit. Unfortunately, natural language

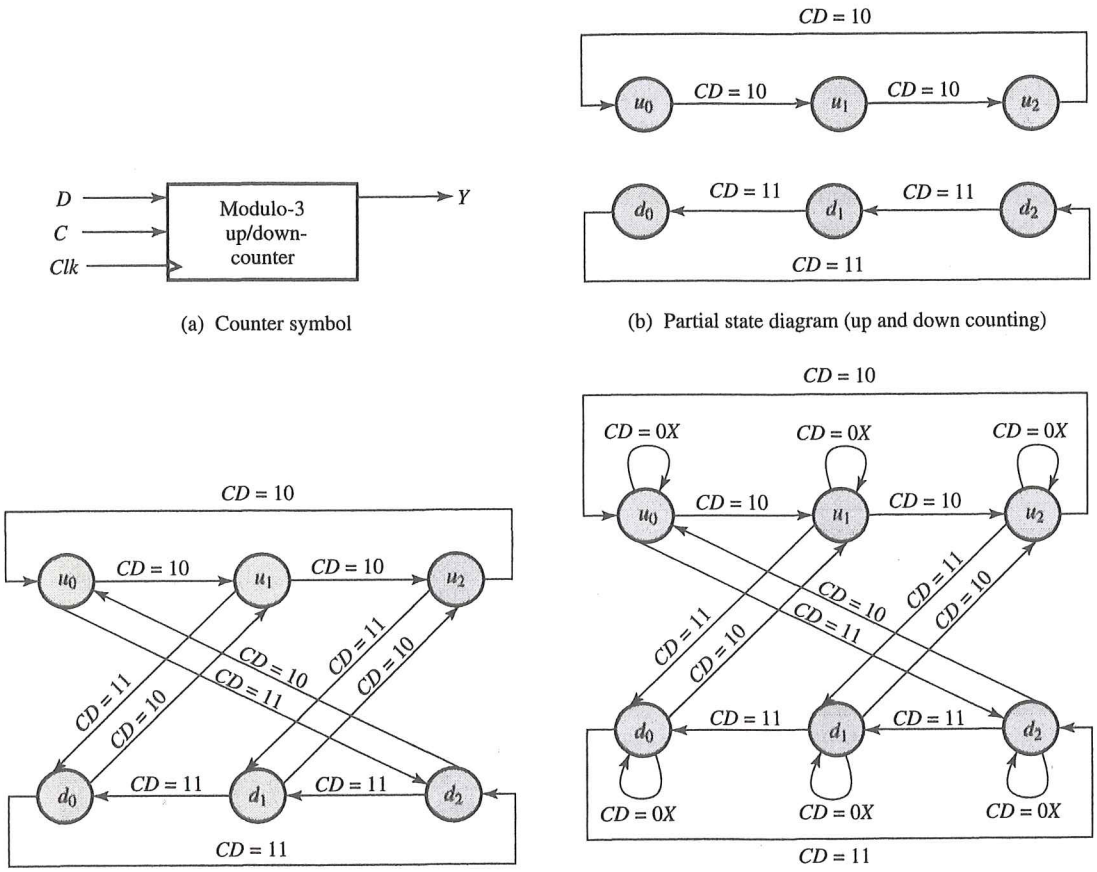
descriptions can sometimes be ambiguous or in many cases, incomplete, since they often focus exclusively on the main function of the sequential circuit, without enumerating all possible cases of its behavior. For this reason, a natural language description is sometimes supplemented with timing diagrams. Even these timing diagrams, however, can be incomplete, since they do not show circuit responses for every possible input sequence, only for the most important. In most cases, then, it is best to replace the natural language description with a more precise one based on a hardware description language such as VHDL, or with a graphical form such as a flowchart or an ASM chart, introduced in Chapters 7 and 8. Regardless of the form we use, though, the construction of the FSM model is a very creative part of the design process, requiring expertise and experience. In this section we demonstrate construction of the state diagram and state table for a simple example.

EXAMPLE 6.4 Modulo-3 up/down-counter

PROBLEM Derive the state diagram for a modulo-3 up/down-counter. The counter has two inputs: count enable (C) and count direction (D). When $C = 1$, the counter will count in the direction specified by D, and it will stop counting when $C = 0$. The counter will count up when $D = 0$ and down when $D = 1$. The counter has one output Y, which will be asserted when the counter reaches 2 while counting up or when it reaches 0 while counting down. The counter symbol is shown in Figure 6.20(a).

SOLUTION On the basis of this description we can conclude that the counter requires at least two flip-flops, since it must remember three digits: 0, 1, and 2. Furthermore, since the counter can count up or down, we need two sequences: up and down. The up sequence consists of three states: u_0, u_1, and u_2, with the counter proceeding from u_0 to u_1 to u_2 and returning to u_0 as long as $CD = 10$. Similarly, when the counter counts down, it again goes through three states, in this case d_0, d_1, and d_2. In the down sequence, however, the counter proceeds from d_0 to d_2 to d_1 and returns to d_0 as long as $CD = 11$. The up and down sequences are shown in Figure 6.20(b).

In developing the state diagram of the counter, we need to account for the possibility that the counter might change direction while it is counting—in other words, we have to allow for the possibility that the value of input D may change



(a) Counter symbol

(b) Partial state diagram (up and down counting)

(c) Partial state diagram (changing direction)

(d) Final state diagram

FIGURE 6.20
State diagram for a modulo-3 counter.

while $C = 1$. Although this case is not mentioned in the input description given above, we can accommodate such an occurrence by assuming that the counter would continue counting in a new direction from the peer state in the other sequence—that is, if D changes from 0 to 1, the counter will go from u_i to $d_{(i-1) \bmod 3}$, and if D changes from 1 to 0, the counter will go from d_i to $u_{(i+1) \bmod 3}$, for $i = 0, 1$, or 2. This case has been added to the state diagram in Figure 6.20(c).

The state diagram will only be complete, however, when we have considered what will happen when the counter is disabled by a change of C from 1 to 0. In this case we assume that the counter will remain in that state until C becomes 1 again. This case is added to the state diagram in

Figure 6.20(d), which now accounts for all possible counter behaviors. ■

Although the state diagram derived in Example 6.4 is complete, it is important to note that the diagram does not contain the minimal number of states. Granted, it is easy to understand, since the various cases of the counter's behavior are clearly distinguished and readily apparent in the state diagram. In sequential design, however, clarity is not usually the goal; on the contrary, cost and performance are often the most important factors, and from this perspective, we need to consider minimizing the number of states as part of the design process.

6.10 STATE MINIMIZATION

The purpose of state minimization is to reduce the number of states in a sequential circuit so that the circuit requires fewer flip-flops, which in turn will reduce the cost of its implementation. However, a state reduction will not reduce the required number of flip-flops unless the number of states is reduced below the present power-of-2 level. For a sequential circuit with m states, for example, we would need $\lceil \log_2 m \rceil$ flip-flops. Since reducing the number of states by Δ would require $\lceil \log_2 (m-\Delta) \rceil$ flip-flops, Δ must be such a number that $\lceil \log_2 m \rceil$ is greater than $\lceil \log_2 (m-\Delta) \rceil$ by at least 1.

Consider, for example, a circuit with six states. Reducing the number of states to five will not reduce the number of flip-flops, because $\lceil \log_2 6 \rceil = \lceil \log_2 5 \rceil = 3$. On the other hand, reducing the number of states to four would reduce the number of flip-flops by 1, because $\lceil \log_2 4 \rceil = 2$.

A second benefit of state reduction is that it can reduce the number of gates and the number of inputs per gate that are needed to implement next-state and output functions. The advantage of such a reduction lies in the fact that gates with fewer inputs are faster; by decreasing the sequential circuit delay, we can improve the clock frequency of the circuit and thereby improve its overall performance.

In general, state minimization is based on the concept of the **behavioral equivalence** of FSMs and, by extension, the equivalence of its states. For example, we would say that two FSMs are equivalent if they produce the same sequence of output symbols for every sequence of input symbols. In some cases, equivalent FSMs may have different numbers of states and may also transition through a different sequence of states for every input sequence; nonetheless, they are considered behaviorally equivalent as long as they produce the same output sequence.

We can conclude from the discussion above that a FSM with the larger number of states will have some states that are equivalent. In cases like this, then, we can reduce the number of states in the FSM by merging those states that are equivalent.

State equivalence can be defined on the basis of the values of state outputs and next states. More formally, two states, s_j and s_k, in an FSM are said to be equivalent, $s_j \equiv s_k$, iff the following two conditions are true.

1. Both states s_j and s_k produce the same output symbol for every input symbol i: that is, $h(s_j, i) = h(s_k, i)$.
2. Both states have equivalent next states for every input symbol i: that is, $f(s_j, i) \equiv f(s_k, i)$.

From this definition of state equivalence, we can derive a simple procedure for obtaining an FSM with the minimal number of states. The procedure requires partitioning all the states in an FSM into equivalence classes and constructing the minimal-state FSM in which each state will represent one equivalence class.

In practice, this procedure consists of two steps. In the first step, we compare output symbols for each state and for each input symbol. The goal of this comparison is to combine states into groups in such a way that all states in the same group generate the same output symbol for each input symbol. In the second step, we determine, for each state in the group and for every input symbol, the next state. Then we can partition the groups into subgroups in such a way that all states in a subgroup have their next states in the same group for every input symbol.

In some cases, all the states that are in the same subgroup after partitioning do not have their next states in the same subgroup although they have their next states in the same group. In such cases, the second step must be repeated until no further partitioning is necessary. Eventually, though, each subgroup will represent an equivalence class that is equal to one state of the minimal FSM. In the following example we demonstrate this procedure on the modulo-3 counter from Example 6.4.

EXAMPLE 6.5 State reduction

PROBLEM Derive the minimal-state FSM for the modulo-3 counter.

SOLUTION As shown in Figure 6.20(c), the modulo-3 counter has six states. In Figure 6.21(a) we have converted this state diagram into a next-state/output table. Given the information provided by this table, we can now apply the procedure described above to the set of states in Figure 6.21(b). Note

PRESENT STATE	NEXT STATE		
	$CD = 0X$	$CD = 10$	$CD = 11$
u_0	$u_0/0$	$u_1/0$	$d_2/1$
u_1	$u_1/0$	$u_2/0$	$d_0/0$
u_2	$u_2/0$	$u_0/1$	$d_1/0$
d_0	$d_0/0$	$u_1/0$	$d_2/1$
d_1	$d_1/0$	$u_2/0$	$d_0/0$
d_2	$d_2/0$	$u_0/1$	$d_1/0$

(a) Initial state table

	$\{u_0,$	$u_1,$	$u_2,$	$d_0,$	$d_1,$	$d_2\}$
$CD = 0X$	0	0	0	0	0	0
10	0	0	1	0	0	1
11	1	0	0	1	0	0

Output values

001 000 010

	$G_0 = \{u_0, d_0\}$	$G_1 = \{u_1, d_1\}$	$G_2 = \{u_2, d_2\}$
$CD = 0X$	$G_0\ G_0$	$G_1\ G_1$	$G_2\ G_2$
10	$G_1\ G_1$	$G_2\ G_2$	$G_0\ G_0$
11	$G_2\ G_2$	$G_0\ G_0$	$G_1\ G_1$

Next states

(b) Partitioning into equivalence classes

PRESENT STATE	NEXT STATE		
	$CD = 0X$	$CD = 10$	$CD = 11$
s_0	$s_0/0$	$s_1/0$	$s_2/1$
s_1	$s_1/0$	$s_2/0$	$s_0/0$
s_2	$s_2/0$	$s_0/1$	$s_1/0$

(c) Final next-state/output table

FIGURE 6.21
State reduction for modulo-3 counter.

that we start with the set of all states and then determine the output values for each combination of input values. For the input values of $CD = 0X, 10, 11$, we know the value of the output signal to be $Y = 0, 0, 1$ or $0, 0, 0$ or $0, 1, 0$. At

this point we can create the following three groups: $G_0 = \{u_0, d_0\}$, $G_1 = \{u_1, d_1\}$, and $G_2 = \{u_2, d_2\}$. Next, we must determine, for each state in each group and for each set of input values, what the next state will be. As you can see, for every value of the input signals, the next states of each state in the group belong to the same group. Therefore, no further partitioning is needed. Since each group represents a class of equivalent states, we can rename the groups G_0, G_1, and G_2 as states s_0, s_1, and s_2 in the minimal-state FSM. The next-state/output table for this minimal FSM is given in Figure 6.21(c). ■

The equivalence classes of an FSM can also be found by constructing an implication table, which allows us to eliminate nonequivalent states and indicate equivalent states. As shown in Figure 6.22 an implication table is a triangular table in which every entry represents a specific pair of states. Note that the rows have been labeled with all the states but the first, whereas the columns have been labeled with all states but the last. By this means we ensure that every pair of states is assigned one entry in the table.

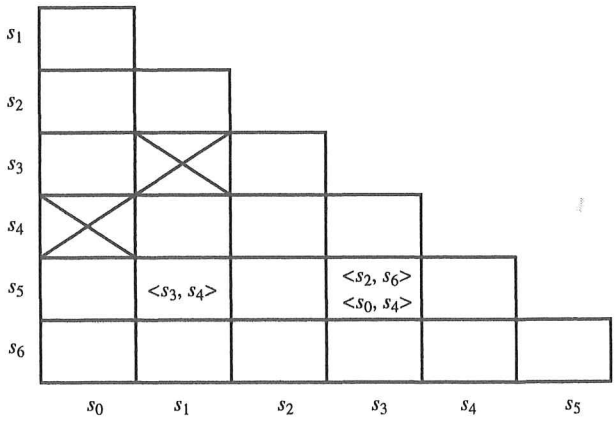


FIGURE 6.22
Implication table.

The procedure for finding equivalence classes follows from the equivalence definition. In the first step we enter an $\times$ for every pair of states that differ in their output values for at least one set of input values. In this manner, a pair of states that are not equivalent can be eliminated from consideration, as shown for pair $< s_1, s_3 >$ and $< s_0, s_4 >$ in Figure 6.22. For the remaining pairs of states, we enter into each entry the next-state pairs that would have to be equivalent if the pair of states represented by the entry are to be equivalent. In general, we say that

the equivalence of the pair of next states is implied, which gives the name to the implication table. For example, the equivalence of the pair $< s_1, s_5 >$ in Figure 6.22 implies the equivalence of $< s_3, s_4 >$. Similarly, the equivalence of the pair $< s_3, s_5 >$ implies the equivalence of $< s_2, s_6 >$ and $< s_0, s_4 >$ according to condition 2 of the equivalence definition.

In the second step we scan the table from top to bottom one square at a time, and from left to right one column at a time, and enter $\times$ into any square having at least one nonequivalent next-state pair. This second step may be iterated several times since nonequivalence of one state pair may cause the nonequivalence of another state pair, and so on. If no $\times$'s are entered during the entire table scan, we have already found all the nonequivalent pairs, which means that all the noncrossed entries indicate equivalent pairs.

The equivalence classes are formed in the third step by using the transitivity property of the equivalence relation, which states that if $s_i \equiv s_j$ and $s_j \equiv s_k$, then $s_i \equiv s_k$. Thus we can group all the equivalent states into classes by examining the table. We demonstrate this procedure in the following example.

EXAMPLE 6.6 State reduction with implication table

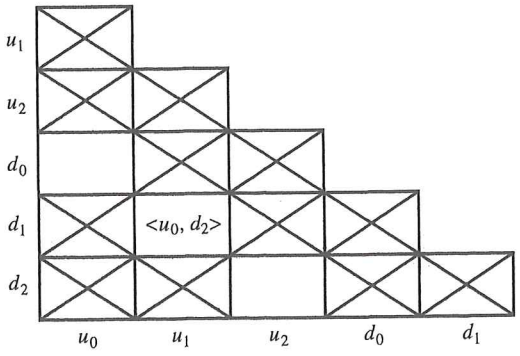
PROBLEM Find the minimal number of states for the FSM specified by the table in Figure 6.23(a). [Note that this table is a slight modification of the table in Figure 6.21(a).]

SOLUTION First, we need to create the implication table, as shown in Figure 6.23(b). Next, we cross out all the squares that represent state pairs that have a different output for at least one set of input values. That leaves only pairs $< u_0, d_0 >$, $< u_1, d_1 >$, and $< u_2, d_2 >$ to be checked for the next-state equivalence. The pair $< u_0, d_0 >$ has the same next states for every set of input values. Therefore, they are equivalent and they do not imply the equivalence of any next states. The pair $< u_1, d_1 >$ requires states u_0 and d_2 to be equivalent, so we enter $< u_0, d_2 >$ into the square representing the pair $< u_1, d_1 >$. States u_2 and d_2 are equivalent since they have the same next states for every set of input values. Next, we scan the table and insert an $\times$ into the square representing $< u_1, d_1 >$, since u_0 and d_2 are not equivalent, which also implies that u_1 and d_1 are not equivalent. This ends the second step of the procedure.

In the third step we need to group all the equivalent states into equivalence classes. In our example, only u_0 and

PRESENT STATE	NEXT STATE / OUTPUT		
	CD = 0X	CD = 10	CD = 11
u_0	$u_0/0$	$u_1/0$	$d_2/1$
u_1	$u_0/0$	$u_2/0$	$d_0/0$
u_2	$u_2/0$	$u_0/1$	$d_1/0$
d_0	$d_0/0$	$u_1/0$	$d_2/1$
d_1	$d_2/0$	$u_2/0$	$d_0/0$
d_2	$d_2/0$	$u_0/1$	$d_1/0$

(a) Next-state and output table



(b) Implication table

FIGURE 6.23
State reduction with implication table.

d_0 and u_2 and d_2 are equivalent. Therefore, the minimal FSM turns out to have four states, represented by the following four equivalence classes: $\{u_0, d_0\}$, $\{u_1\}$, $\{d_1\}$, $\{u_2, d_2\}$.

TABLE 6.4
Twenty-Four Encodings of Four States

ENCODING NUMBER	s_0	s_1	s_2	s_3
1	00	01	10	11
2	00	01	11	10
3	00	10	01	11
4	00	10	11	01
5	00	11	01	10
6	00	11	10	01
7	01	00	10	11
8	01	00	11	10
9	01	10	00	11
10	01	10	11	00
11	01	11	00	10
12	01	11	10	00
13	10	00	01	11
14	10	00	11	01
15	10	01	00	11
16	10	01	11	00
17	10	11	00	01
18	10	11	01	00
19	11	00	01	10
20	11	00	10	01
21	11	01	00	10
22	11	01	10	00
23	11	10	00	01
24	11	10	01	00

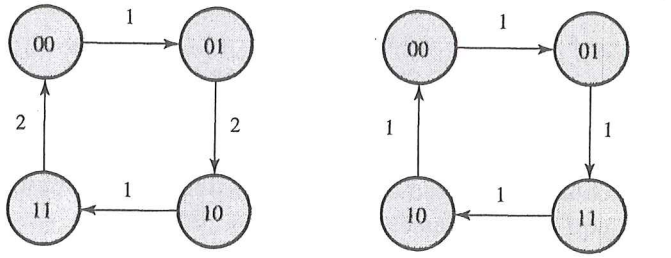
6.11 STATE ENCODING

For any given FSM, the cost and delay inherent in the input and output logic will be largely dependent on what Boolean values are assigned to the symbolic states. For example, a four-state FSM with states s_0, s_1, s_2, and s_3 could be implemented with two flip-flops that contain values 00, 01, 10, or 11. In this case, then, there would be at least $4! = 4 \times 3 \times 2 \times 1 = 24$ possible encodings of the four states to flip-flop values as shown in Table 6.4. In practice, there are generally more than $n!$ encodings for n different states, since we could use more than $\log_2 n$ bits

for the encoding of n states. For this reason, hand enumeration of all these encodings, as well as cost and delay estimation for the input and output logic, can become tedious, even for a small number of states. To avoid this problem, designers use various strategies or heuristics for state assignment. In the rest of this section we explain the three most popular strategies: minimum bit change, prioritized adjacency, and hot-one encoding.

The **minimum-bit-change** strategy assigns Boolean values to the states in such a way that the total number of bit changes for all state transitions is minimized. In other words, if every arc in the state diagram has a weight that is equal to the number of bits by which the source and destination encodings differ, the optimal encoding would be the one that minimizes the sum of all these weights.

In Figure 6.24(a) and (b) we show two encodings for the same state diagram. The straightforward encoding for this binary counter would have two arcs with a weight of 1 and two arcs with a weight of 2, for a total weight of 6. On the other hand, in the minimum-bit-change encoding, all four arcs would have a weight of 1. In the second encoding, the total weight is minimal, since any two state encodings must differ by at least one bit. This minimum-bit-change strategy is based on the premise that in a two-level implementation, we need at least an extra AND gate and an extra input to the OR gate for setting or resetting a flip-flop, for each bit change.



(a) Straightforward encoding (b) Minimum-bit-change encoding

FIGURE 6.24
Two different encodings for a 2-bit binary counter.

The second strategy for state assignment, the **prioritized-adjacency strategy**, proceeds by assigning adjacent encodings, which differ in one bit only, to all states that have a common destination, source, or output. In this procedure the highest priority is given to states that have the same next state for a given input value. The rationale for this priority derives from the fact that the same next-state encoding will appear in two adjacent entries in the Karnaugh map during the logic minimization, in

which case this strategy will lead to a reduction of one literal for each 1 in the next-state encoding. Second priority is given to the next states of the same state, on the assumption that they will also appear adjacent in the Karnaugh map during minimization. Note that this will occur only if the input values, which cause the transition, differ in only one bit. Finally, third priority is given to states that have the same output value for the same input values, on the assumption that adjacent state encodings will create a 1-cube in the Karnaugh map during logic minimization of the output signals.

Figure 6.25(a) shows a state diagram with four states. Note that states s_1 and s_2 satisfy the condition of the first priority; that is, the input value of 0 will move both states into the same state s_3. In addition, they satisfy the condition of the second priority, since they are both next states of the state s_0. In relation to the third priority, note that states s_0 and s_1 have the same output value 0 for the same input value 0 and that states s_2 and s_3 satisfy the same condition. These priorities are listed in Figure 6.25(b), and in Figure 6.25(c) we show a possible encoding that would satisfy the various adjacency priorities.

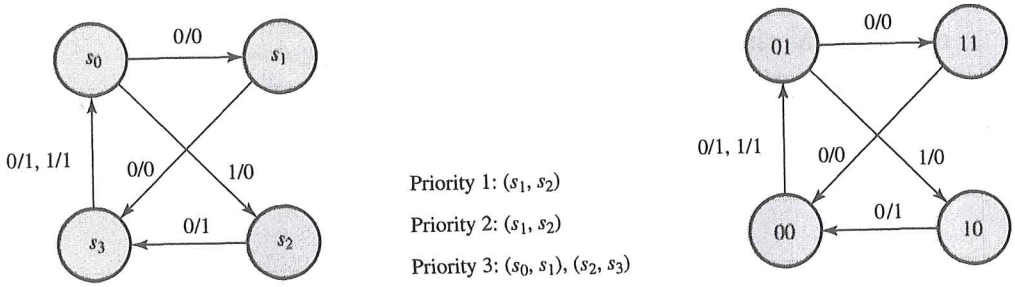


FIGURE 6.25
Encodings based on prioritized adjacency.

The third strategy for state assignment, **hot-one encoding**, uses redundant encoding in which one flip-flop is assigned to each state. In other words, each state is distinguishable by its own flip-flop having a value of 1 while all the others have a value of 0. In practice, this means that one flip-flop will be set to 1 and another reset to 0 during each transition from state to state. The name of this encoding derives from the fact that the value 1 reminds us of a hot potato being passed from one hand to another. The obvious limitation of hot-one encoding is that it works only for an FSM with a small number of empty states because the number of flip-flops becomes excessive if there are more than a few states.

In general, the best strategy for state encoding is to determine several possible options and then estimate the cost and delay of input and output logic for each candidate encoding. In the rest of this section we demonstrate this procedure using our example of the modulo-3 counter.

EXAMPLE 6.7 State encodings for modulo-3 counter

PROBLEM Given the up/down modulo-3 counter that was specified by the minimal next-state/output table in Figure 6.21(c), derive the encoding that will minimize the cost and delay of the counter logic.

SOLUTION As you can see from the next-state/output table, this counter has only three states, which means that any encoding produces two pairs of states with adjacent encodings and one pair of states whose encodings differ in two bits. Furthermore, since we can encode four states with two variables, Q_1 and Q_0, one combination of values would be redundant. Thus we could omit the combination $Q_1Q_0 = 11$, since it would allow us to reduce the number of 1's during logic minimization. One possible encoding, then, is encoding A, shown in Figure 6.26. This is the encoding we obtain if we follow the minimum-bit-change strategy.

	ENCODING A	ENCODING B	ENCODING C
STATES	Q_1Q_0	Q_1Q_0	$Q_2Q_1Q_0$
s_0	0 0	0 1	0 0 1
s_1	0 1	0 0	0 1 0
s_2	1 0	1 0	1 0 0

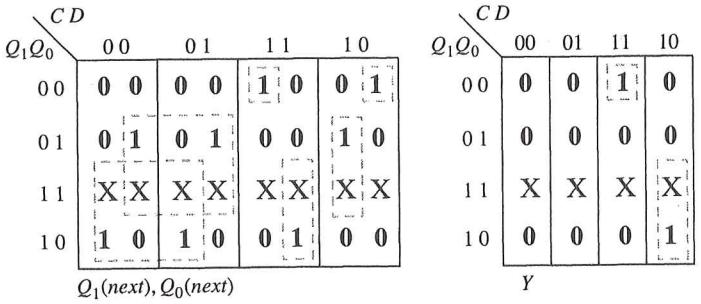
FIGURE 6.26
Possible state encodings for modulo-3 counter.

If we use the prioritized-adjacency strategy, we find that no pair of states satisfies the first and second priority rules. According to the third priority rule, we find that states s_0 and s_1, and s_1 and s_2, should be given adjacent encodings, an option already satisfied in encoding A. In this simple problem we could also use a strategy that simplifies state decoding, by assigning combinations $Q_1Q_0 = 01$ and $Q_1Q_0 = 10$ to states s_0 and s_2, which are the only states with $Y = 1$. This encoding option is shown as encoding B in Figure 6.26. Its primary advantage is that it reduces the number of AND gate inputs

in the implementation of the output logic. Finally, a third strategy is to use hot-one encoding, which yields encoding C in Figure 6.26.

At this point we have established encodings A, B, and C as possible candidate encodings. To evaluate the benefit of each, we must now estimate the cost and delay of their respective input and output logic implementations. For this purpose we need to derive excitation and output equations and estimate the cost and delay of their two-level implementation with NAND gates. To simplify this estimation, we assume that each variable's true and complement values are available at no cost or delay. This assumption is always true for the flip-flop variables, and it is also true for input variables since they require double inverters to increase their input signal strength. The input double inverters add a constant cost and delay to the estimate which can be omitted since it does not affect the comparison of two alternative implementations.

In Figure 6.27 we have shown the cost and delay estimation for encoding A. First, the Karnaugh maps for next-state

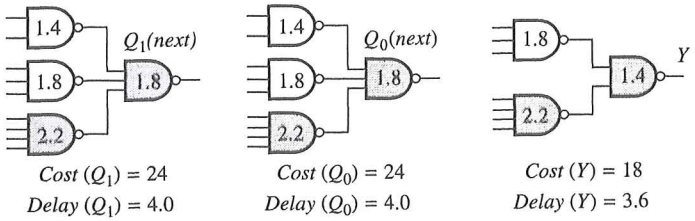


(a) Next-state map

(b) Output map

$$Q_1(next) = Q_1C' + Q_0CD' + Q_1'Q_0'CD$$
$$Q_0(next) = Q_0C' + Q_1CD + Q_1'Q_0'CD'$$
$$Y = Q_1CD' + Q_1'Q_0'CD$$

(c) Excitation and output equations

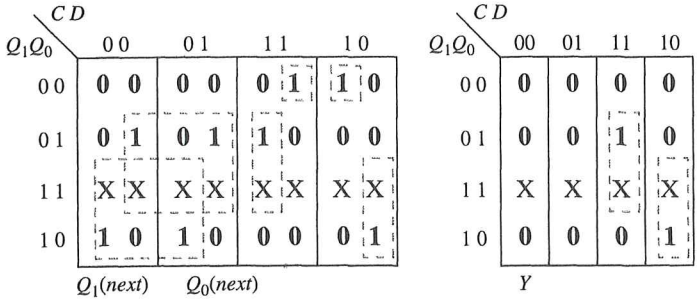


(d) Cost and delay estimation

FIGURE 6.27
Cost and delay estimation for encoding A.

and output functions are presented in Figure 6.27(a) and (b), and the excitation and output equations derived from the Karnaugh maps are given in Figure 6.27(c). Finally, each of these equations has been implemented with two-level logic networks of NAND gates. As you can see, the total cost would be equal to $cost(Q_1) + cost(Q_2) + cost(Y) = 24 + 24 + 18 = 66$, and the maximum input delay would be equal to 4.0 ns, while the output delay would be equal to 3.6 ns.

In Figure 6.28 you will find a similar estimation based on encoding B. As we expected in this encoding, the cost and delay of the output logic is slightly improved, since the total cost $cost(Q_1) + cost(Q_2) + cost(Y) = 24 + 24 + 16 = 64$, and the maximum input logic delay is 4.0 ns while the output delay is 3.2 ns.

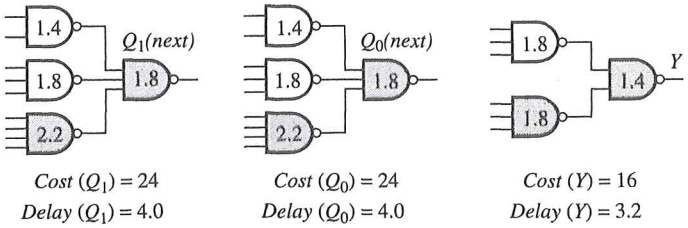


(a) Next-state map

(b) Output map

$$Q_1(next) = Q_1C' + Q_0CD + Q_1'Q_0'CD'$$
$$Q_0(next) = Q_0C' + Q_1CD' + Q_1'Q_0'CD$$
$$Y = Q_0CD + Q_1CD'$$

(c) Excitation and output equations



(d) Cost and delay estimation

FIGURE 6.28
Cost and delay estimation for encoding B.

The estimation of encoding C is shown in Figure 6.29. Note that the cost and delay of each excitation equation has been reduced but the total cost is much higher than

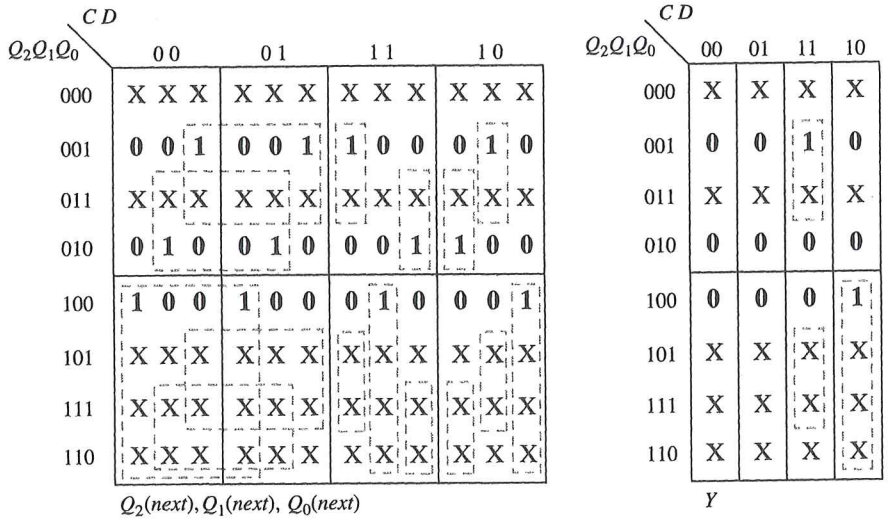


FIGURE 6.29
Cost and delay estimation for encoding C.

before, mainly because the implementation has three flip-flops instead of two. As a result, the total cost is equal to $cost(Q_0)+cost(Q_1)+cost(Q_2)+cost(Y) = 22+22+22+16 = 82$, even though the maximum input logic delay is only 3.6 ns and the output delay is the same as for encoding B.

The conclusion to be drawn from this comparison is that encoding C generates the fastest and costliest implementation, while encoding B generates an implementation that is

the least expensive of the three alternatives but also slower than that of encoding C. Finally, encoding A generates an implementation that is very similar and only insignificantly inferior to encoding B.

In general, it is very difficult to estimate the impact of each encoding unless we do as we did in Example 6.7, generating an implementation for each encoding and then comparing them on cost, delay, and other quality metrics. In the rest of this chapter, we use encoding A for implementation of the modulo-3 counter, since it is the most natural and easiest to understand.

6.12 CHOICE OF MEMORY ELEMENTS

After we complete the process of state minimization and state encoding, we are ready to choose the proper type of flip-flop for implementation of a given FSM. As shown in Tables 6.1 and 6.2, there are four types of flip-flops. In general, T flip-flops are an excellent match for counter-type sequential circuits in which the flip-flops must flip from 0 to 1 and back from 1 to 0 with great frequency. D flip-flops would usually be used in applications where input information must be stored for some time and then used later. In this sense, D flip-flops can be thought of as temporary storage or information scratch pads. The SR flip-flop is generally used in situations where different signals set and reset the flip-flops. Finally, the JK flip-flop, which is the most complex, is useful whenever we need to combine the behavior of a T and an SR flip-flop.

From the discussion above, we could conclude that the SR and JK types would be most useful. However, although they do tend to reduce the cost of the input logic, they also require twice as many connections as do T and D flip-flops. Generally, since T and D flip-flops require fewer connections, they are better suited for VLSI implementations. To compare the overall efficiency of the various flip-flops, we now derive the input logic for the modulo-3 counter using each of the four types of flip-flops.

EXAMPLE 6.8

PROBLEM Given the modulo-3 counter with encoding A, as specified in Figure 6.26, select the type of flip-flop that will minimize the cost and/or delay of the input logic.

SOLUTION To accomplish this task, we start with the next-state table shown in Figure 6.27(a), which has been duplicated in Figure 6.30(a). Then, to derive excitation equations for the various types of flip-flops, we use excitation tables for all four flip-flops, shown in Figure 6.30(b). Next, we would take each pair of present and next states from the next-state map and replace their next-state values with the required input values shown in the excitation table. In this way we can

Q_1Q_0 \ CD	00	01	11	10
00	0 0	0 0	1 0	0 1
01	0 1	0 1	0 0	1 0
11	X X	X X	X X	X X
10	1 0	1 0	0 1	0 0
	$Q_1(next)$	$Q_0(next)$		

(a) Next-state table for encoding A

Q (present)	Q (next)	S	R	J	K	T	D
0	0	0	X	0	X	0	0
0	1	1	0	1	X	1	1
1	0	0	1	X	1	1	0
1	1	X	0	X	0	0	1

(b) Flip-flop excitation table

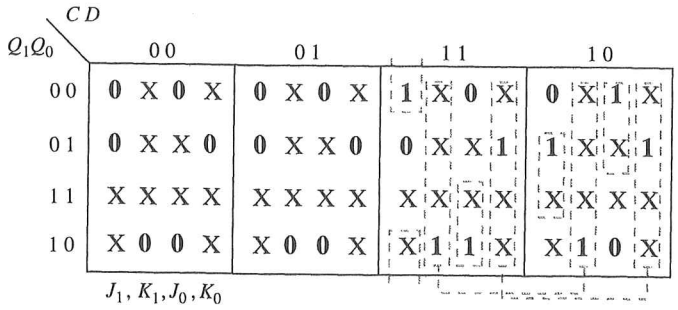
Q_1Q_0 \ CD	00	01	11	10
00	0 X 0 X	0 X 0 X	1 0 0 X	0 X 1 0
01	0 X X 0	0 X X 0	0 X 0 1	1 0 0 1
11	X X X X	X X X X	X X X X	X X X X
10	X 0 0 X	X 0 0 X	0 1 1 0	0 1 0 X

S_1, R_1, S_0, R_0

$S_1 = Q_0 CD' + Q_1'Q_0'CD$ [*cost* = 18, *delay* = 3.6]
$R_1 = Q_1C = (Q_1' + C')'$ [*cost* = 4, *delay* = 1.4]
$S_0 = Q_1CD + Q_1'Q_0'CD'$ [*cost* = 18, *delay* = 3.6]
$R_0 = Q_0C = (Q_0' + C')'$ [*cost* = 4, *delay* = 1.4]

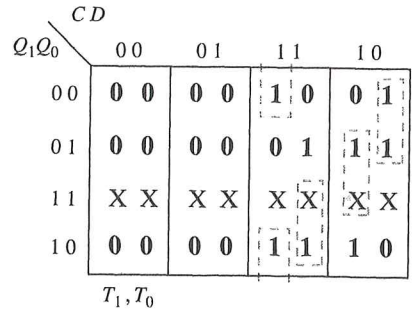
(c) Implementation with SR flip-flops

FIGURE 6.30
Modulo-3 counter implementation with various types of flip-flops.



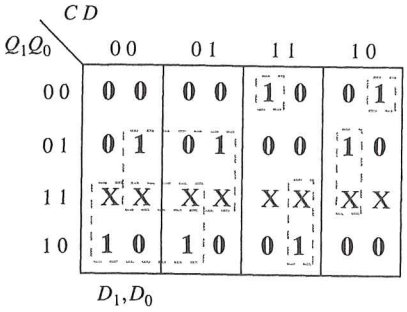
$J_1 = Q_0CD' + Q_0'CD = (C' + Q_0D + Q_0'D')'$ [*cost* = 12, *delay* = 2.4]
$K_1 = C$ [*cost* = 0, *delay* = 0]
$J_0 = Q_1CD + Q_1'CD' = (C' + Q_1D' + Q_1D')'$ [*cost* = 12, *delay* = 2.4]
$K_0 = C$ [*cost* = 0, *delay* = 0]

(d) Implementation with JK flip-flops



$T_1 = Q_1C + Q_0'CD + Q_0CD'$ [*cost* = 22, *delay* = 3.6]
$T_0 = Q_0C + Q_1CD + Q_1'CD'$ [*cost* = 22, *delay* = 3.6]

(e) Implementation with T flip-flops



$D_1 = Q_1C' + Q_0CD' + Q_1'Q_0'CD$ [*cost* = 24, *delay* = 4.0]
$D_0 = Q_0C' + Q_1CD + Q_1Q_0CD'$ [*cost* = 24, *delay* = 4.0]

(f) Implementation with D flip-flops

FIGURE 6.30
Continued.

create input maps for the SR, JK, T, and D flip-flops, which are shown in Figure 6.30(c), (d), (e), and (f), respectively. Note that the input maps really combine four maps in the case of the SR and JK flip-flops, or two maps in the case of T and D flip-flops. From these input maps we can now derive minimal expressions for each flip-flop input, shown underneath each map, along with cost and delay estimates for each expression. These cost and delay estimates do not include the input drivers (inverters) or the cost and delay of the flip-flops.

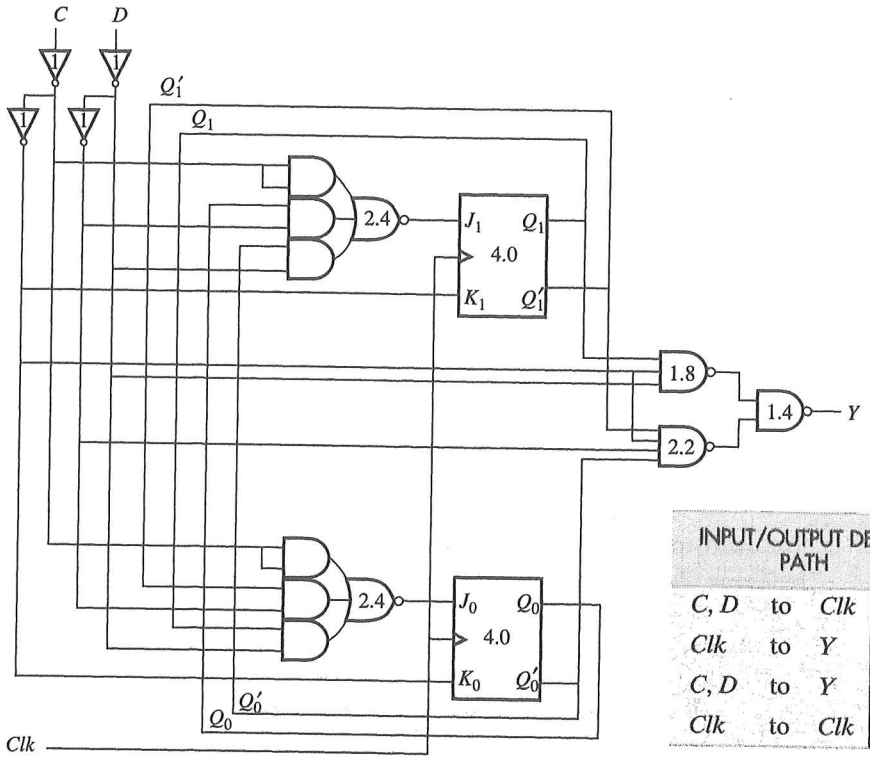
From these estimates we can see that an implementation with JK flip-flops has the lowest cost and least delay, mainly because in this case we can use one AOI gate for implementation of the J_1 and J_0 input logic instead of two levels of NAND gates. In many cases, though, when sequential circuits with many states are being implemented, JK flip-flops will not have such a great advantage over other types of flip-flops. We can also see from this example that the T, D, and SR flip-flops generate implementations with very similar costs and delays. ■

6.13 OPTIMIZATION AND TIMING

The next-to-last step in a sequential logic synthesis would consist of mapping the input and output logic to the components in the given library. In the case of the modulo-3 counter, we had already used AOI gates when we computed the delay and cost of the implementation with JK flip-flops.

After the technology mapping is completed, we can draw the schematic to visualize all the counter's gates and connections. As an example, the logic schematic for the JK implementation of the modulo-3 counter is shown in Figure 6.31. (The output logic was derived in Figure 6.27(c).) In this figure you can see that double inverters are used at the input to deliver more current, which decreases the delay caused by the charging and discharging of wire capacitances. Note that this logic schematic is generated for human consumption and will have to be converted into a component netlist expressed in a hardware description language before it can be used by the simulation and testing software.

The final step in the process of sequential synthesis consists of deriving a timing diagram from the schematic and the given gate and flip-flop delays. The timing diagram for the modulo-3 counter is shown in Figure 6.32. Note that the delays in the timing diagram correspond to the delays given in Figure 6.31(b). In Figure 6.32 the modulo-3 counter is enabled at t_0 but does not change the state until the rising edge of the *Clk* signal. It enters state $s_1(Q_1Q_0 = 01)$ at $t_1 + 4.0$ ns. After the second rising edge it enters state $s_2(Q_1Q_0 = 10)$ at $t_2 + 4.0$ ns. The output signal Y becomes 1 at $t_2 + 7.6$ ns since Y must be 1 in s_2 if $CD = 10$. When at the next rising edge modulo-3 counter enters state $s_0(Q_1Q_0 = 00)$ output Y returns to 0 at $t_3 + 7.6$ ns. When D is disasserted



(a) Logic schematic

INPUT/OUTPUT DELAY PATH			
C, D	to	Clk	4.4
Clk	to	Y	7.6
C, D	to	Y	5.6
Clk	to	Clk	6.4

(b) Delay table

FIGURE 6.31
Modulo-3 counter schematic.

at t_4 the output Y becomes 1 again at $t_4 + 5.6$ ns since output Y must be 1 in state s_0 if $CD = 11$. After that modulo-3 counter enters state $s_2(Q_1Q_0 = 10)$ at $t_5 + 4.0$ ns and Y returns to 0 at $t_5 + 7.6$ ns since Y must be 0 in s_2 if $CD = 11$. Finally, after $t_6 + 4.0$ modulo-3 counter enters state $s_1(Q_1Q_0 = 01)$. This is the last state shown in this timing diagram.

This and other timing diagrams will be used to verify the input and output behavior of the synthesized circuit and to generate input and output waveforms for simulation. Given as a set of input waveforms, a simulator will generate the waveforms for the outputs, which must be compared to the expected output waveforms obtained from the schematics, circuit specification, or behavioral description. These input and output waveforms are sometimes called test vectors since they will also be used for testing the circuit after it has been manufactured.

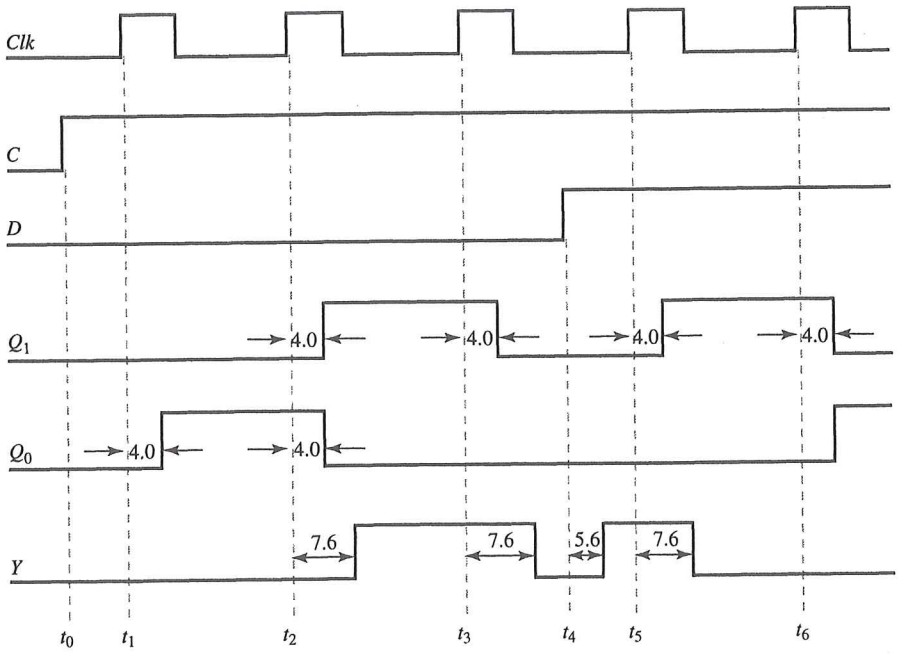


FIGURE 6.32
Timing diagram of a modulo-3 counter for a sequence of input values.

6.14 CHAPTER SUMMARY

In this chapter we have explained how to construct the basic memory elements that are used in the design of sequential logic. To this end we have introduced several different types of latches and flip-flops and described them with characteristic equations, characteristic tables, and state diagrams. We have also introduced the concept of sequential logic, as well as the finite-state-machine model, which is used to represent the sequential logic during the design process. We have provided step-by-step procedures for the analysis and synthesis of sequential logic and demonstrated these procedures on examples of modulo-4 and modulo-3 counters. In Chapter 7 we use these procedures to analyze and synthesize sequential components.

6.15 FURTHER READINGS

DeMicheli, G. *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

Describes logic and sequential synthesis concepts and algorithms in detail. Requires expert knowledge in design and in CAD tool development.

Katz, R. H. *Contemporary Logic Design.* Redwood City, CA: Benjamin-Cummings, 1994.

Introductory text on logic design, with explanations of how to use UC–Berkeley CAD tools in the design process.

Kohavi, Z. *Switching and Automatic Theory*, 2nd ed. New York: McGraw-Hill, 1978.

Thorough theoretical treatment of finite-state machines and sequential logic.

McCluskey, E. *Logic Design Principles.* Englewood Cliffs, NJ: Prentice Hall, 1986.

Provides detailed explanations of the basic concepts in sequential logic analysis, synthesis, and testing.

6.16 PROBLEMS

6.1 (Clock signal) Compute the clock frequency and duty cycle for a clock signal with a width and period of:
- **(a)** 5 ns and 20 ns
- **(b)** 10 ns and 100 ns
- **(c)** 100 ns and 1 ns

6.2 (SR latch) Draw the output timing diagram of (a) NOR and (b) NAND implementation of an SR latch for the input signals depicted in Figure P6.2.

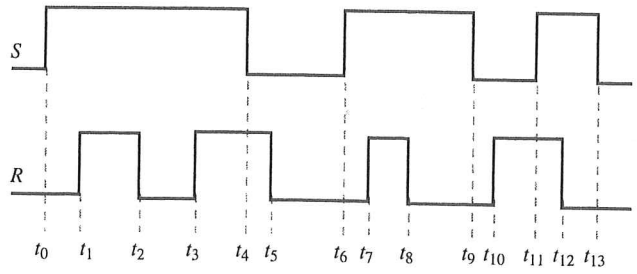


FIGURE P6.2

6.3 (SR latch) Derive an implementation of a clocked SR latch using only:
- **(a)** NOR gates
- **(b)** NAND gates
- **(c)** AND, OR, and INVERT gates

6.4 (Clocked D latch) Design a D latch using only (a) NAND gates and (b) NOR gates, and then compute *D*-to-*Q* and *Clk*-to-*Q* delays for positive and negative output transitions.

6.5 (Flip-flops) Design a master–slave (a) SR flip-flop, (b) JK flip-flop, (c) D flip-flop, and (d) T flip-flop using clocked SR latches and AND, OR, and INVERT gates.

6.6 (JK flip-flops) Derive the output waveforms of a master–slave JK flip-flop for the input waveforms depicted in Figure P6.6.

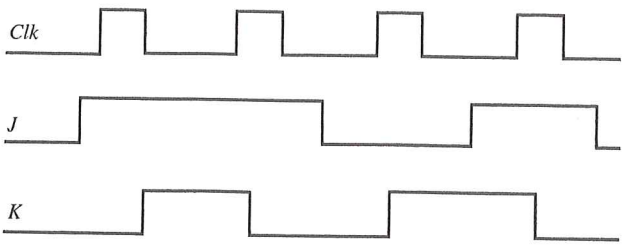


FIGURE P6.6

6.7 (Flip-flops) Using Karnaugh maps, derive the characteristic equations from the characteristic tables of the four flip-flops shown in Table 6.1.

6.8 (Flip-flops) Implement SR, JK, D, and T flip-flops using only AND, OR, and INVERT gates, and:
- **(a)** SR flip-flops
- **(b)** JK flip-flops
- **(c)** D flip-flops
- **(d)** T flip-flops

6.9 (Sequential analysis) Derive a (a) state table and (b) state diagram for the sequential circuit shown in Figure P6.9.

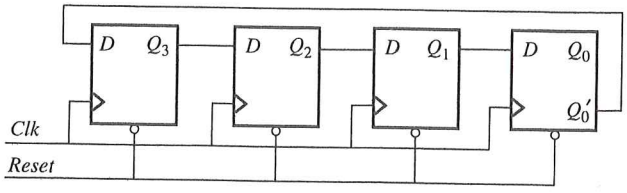


FIGURE P6.9

6.10 (Sequential analysis) Derive the (a) state/output table and (b) FSM representation of the circuit shown in Figure P6.10. What is the function of this sequential circuit?

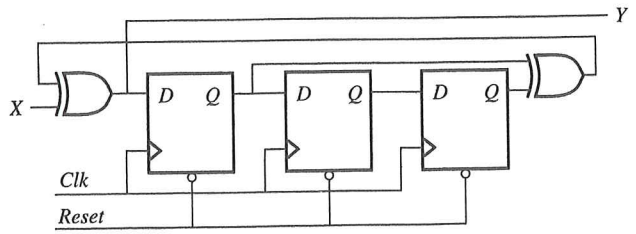


FIGURE P6.10

6.11 (Sequential analysis) Derive (a) excitation equations, (b) a next-state equation, (c) a state/output table, and (d) a state diagram for the circuit shown in Figure P6.11.

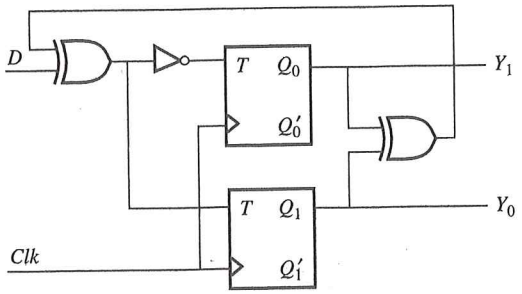


FIGURE P6.11

6.12 (State minimization) Derive the minimal-state FSM from the state/output table shown in Figure P6.12.

PRESENT STATE	NEXT STATE	
	$x = 0$	$x = 1$
s_0	$s_0/1$	$s_4/0$
s_1	$s_0/0$	$s_4/0$
s_2	$s_1/0$	$s_5/0$
s_3	$s_1/0$	$s_5/0$
s_4	$s_2/0$	$s_6/1$
s_5	$s_2/0$	$s_6/1$
s_6	$s_3/0$	$s_7/1$
s_7	$s_3/0$	$s_7/1$

FIGURE P6.12

6.13 (State minimization) Minimize the states for the FSM given in Figure P6.13, using:

(a) State partitioning
(b) An implication table

PRESENT STATE	NEXT STATE / OUTPUT		
	$AB = 00$	$AB = 01$	$AB = 10$
s_0	$s_4/1$	$s_2/0$	$s_1/1$
s_1	$s_2/0$	$s_5/1$	$s_4/1$
s_2	$s_1/1$	$s_0/0$	$s_3/1$
s_3	$s_2/0$	$s_5/1$	$s_4/1$
s_4	$s_0/0$	$s_5/1$	$s_1/1$
s_5	$s_2/0$	$s_4/1$	$s_2/1$

FIGURE P6.13

6.14 (State encoding) For the state diagram shown in Figure P6.14, derive the state encodings using:

(a) The minimum-bit-change heuristic
(b) The prioritized-adjacency heuristic
(c) Hot-one encoding

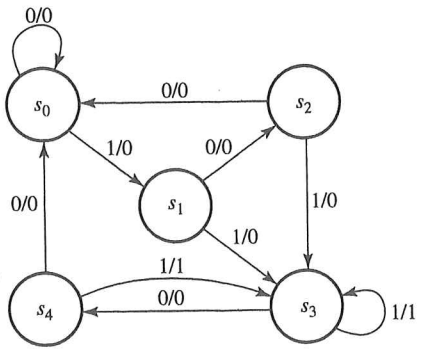


FIGURE P6.14

6.15 (State encoding) Find the state encoding that minimizes the output logic for a counter that counts in the following sequence: 0, 1, 3, 6, 10, 15, 0, ...

6.16 (Sequential synthesis) Design a counter that counts in the sequence 0, 1, 3, 6, 10, 15, using four (a) D, (b) SR, (c) JK, and (d) T flip-flops as memory elements, and natural binary encoding.

6.17 (Sequential synthesis) Design a counter that counts in the sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, ..., using natural binary encoding and D flip-flops.

6.18 (Sequential synthesis) Design a parity checker that counts the number of 1's in the input stream. This checker asserts its output Y if it has received an odd number of 1's on the input X. An asynchronous *Reset* signal returns the parity checker into its initial state. As storage elements use only (a) D, (b) JK, and (c) T flip-flops.

6.19 (Sequential synthesis) Design a recognizer that recognizes an input sequence that has at least three 1's. The recognizer has a single input X and a single output Y, in addition to an asynchronous *Reset* signal. The recognizer sets the output Y to 1 if the input signal X equals 1 at least three clock cycles after *Reset* was disasserted. For the recognizer described above:

(a) Devise the state diagram.

(b) Minimize the number of states.

(c) Encode the states to minimize the combinatorial logic.

(d) Draw a schematic diagram using D flip-flops.

6.20 (Sequential synthesis) Redo Problem 6.19, designing a recognizer that will recognize all the input sequences that have three or more consecutive 1's, or three or more consecutive 0's.

6.21 (Sequential synthesis) Implement a sequential circuit that can complement a 16-bit two's-complement number, $X = x_{15}x_{14} \cdots x_1x_0$. The circuit has one input, X, which has the value x_i in the clock cycle i, and another input *Reset*, which resets the circuit into the initial state after 16 clock cycles. The output Y is the two's complement of X. In the initial state a flag is set to 0. The circuit works as follows: For every x_i, $0 \leq i \leq 15$. If $flag = 0$, then $y_i = x_i$ and $flag = x_i$; else $y_i = x_i'$. Develop a state diagram and logic schematic using D-type flip-flops.

6.22 (Sequential synthesis) Design a simplified traffic-light controller that switches traffic lights on a crossing where a north-south (NS) street intersects an east-west (EW) street. The input to the controller is the *WALK* button pushed by pedestrians who want to cross the street. The outputs are two signals *NS* and *EW* that control the traffic lights in the NS and EW directions. When *NS* or *EW* are 0, the red light is on, and when they are 1, the green light is on. When there are no pedestrians, $NS = 0$ and $EW = 1$ for 1 minute, followed by $NS = 1$ and $EW = 0$ for 1 minute, and so on. When a *WALK* button is pushed, *NS* and *EW* both become 1 for a minute when the present minute expires. After that the *NS* and *EW* signals continue alternating. For these traffic-light controller:

(a) Develop a state diagram and a state/output table.

(b) Minimize the number of states.

(c) Encode the states

(d) Derive a logic schematic.