

---

## chapter

# 5

# FLIP-FLOPS, REGISTERS, AND COUNTERS

## CHAPTER OBJECTIVES

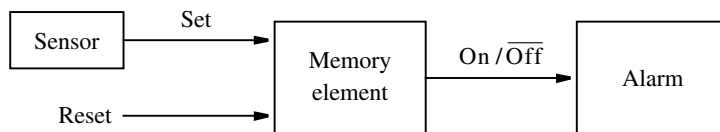
In this chapter you will learn about:

- Logic circuits that can store information
- Flip-flops, which store a single bit
- Registers, which store multiple bits
- Shift registers, which shift the contents of the register
- Counters of various types
- Verilog constructs used to implement storage elements

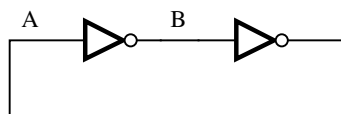
In previous chapters we considered combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit. Such circuits include storage elements that store the values of logic signals. The contents of the storage elements are said to represent the *state* of the circuit. When the circuit's inputs change values, the new input values either leave the circuit in the same state or cause it to change into a new state. Over time the circuit changes through a sequence of states as a result of changes in the inputs. Circuits that behave in this way are referred to as *sequential circuits*.

In this chapter we will introduce circuits that can be used as storage elements. But first, we will motivate the need for such circuits by means of a simple example. Suppose that we wish to control an alarm system, as shown in Figure 5.1. The alarm mechanism responds to the control input  $On/\overline{Off}$ . It is turned on when  $On/\overline{Off} = 1$ , and it is off when  $On/\overline{Off} = 0$ . The desired operation is that the alarm turns on when the sensor generates a positive voltage signal, *Set*, in response to some undesirable event. Once the alarm is triggered, it must remain active even if the sensor output goes back to zero. The alarm is turned off manually by means of a *Reset* input. The circuit requires a memory element to remember that the alarm has to be active until the *Reset* signal arrives.

Figure 5.2 gives a rudimentary memory element, consisting of a loop that has two inverters. If we assume that  $A = 0$ , then  $B = 1$ . The circuit will maintain these values indefinitely because of the feedback loop. We say that the circuit is in the *state* defined by these values. If we assume that  $A = 1$ , then  $B = 0$ , and the circuit will remain in this second state indefinitely. Thus the circuit has two possible states. This circuit is not useful, because it lacks some practical means for changing its state. Useful circuits that exhibit such memory behavior can be constructed with logic gates.



**Figure 5.1** Control of an alarm system.

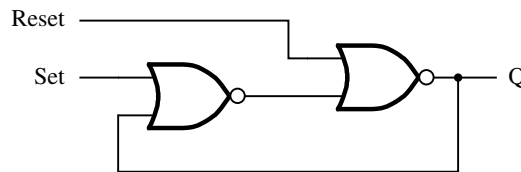


**Figure 5.2** A simple memory element.

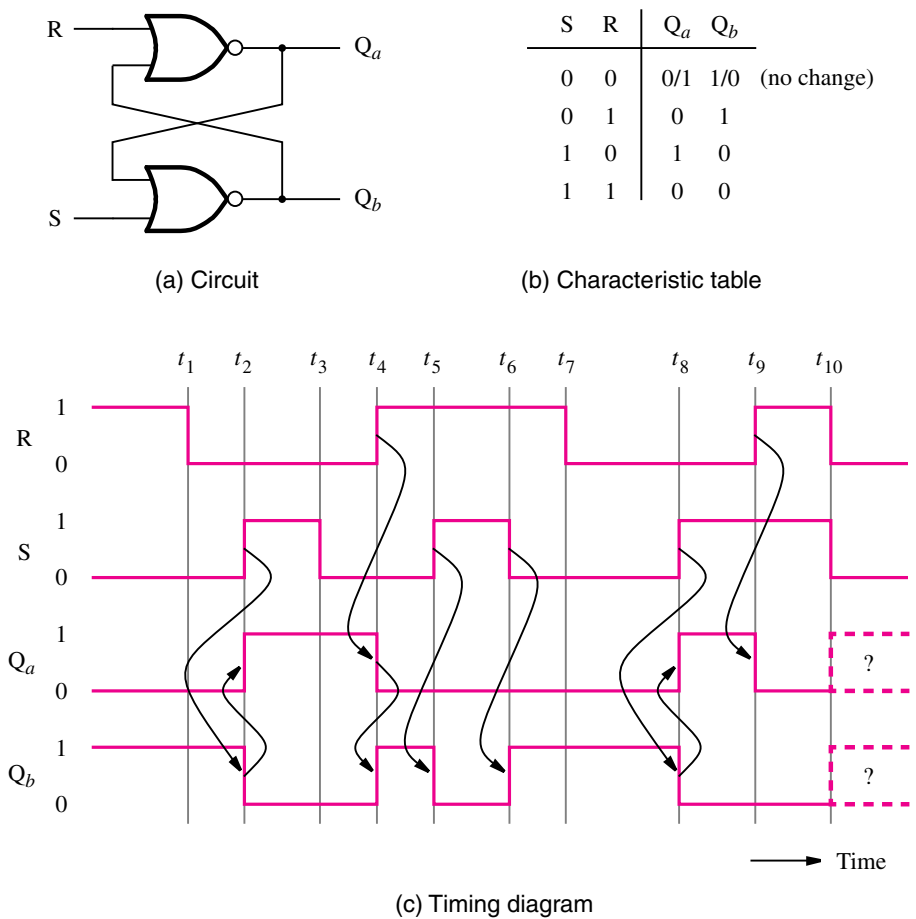
## 5.1 BASIC LATCH

Figure 5.3 presents a memory element built with NOR gates. Its inputs, *Set* and *Reset*, provide the means for changing the state,  $Q$ , of the circuit. A more usual way of drawing this circuit is given in Figure 5.4a, where the two NOR gates are said to be connected in cross-coupled style. The circuit is referred to as a *basic latch*. Its behavior is described by the table in Figure 5.4b. When both inputs,  $R$  and  $S$ , are equal to 0 the latch maintains its existing state because of the feedback loop. This state may be either  $Q_a = 0$  and  $Q_b = 1$ , or  $Q_a = 1$  and  $Q_b = 0$ , which is indicated in the table by stating that the  $Q_a$  and  $Q_b$  outputs have values 0/1 and 1/0, respectively. Observe that  $Q_a$  and  $Q_b$  are complements of each other in this case. When  $R = 1$  and  $S = 0$ , the latch is *reset* into a state where  $Q_a = 0$  and  $Q_b = 1$ . When  $R = 0$  and  $S = 1$ , the latch is *set* into a state where  $Q_a = 1$  and  $Q_b = 0$ . The fourth possibility is to have  $R = S = 1$ . In this case both  $Q_a$  and  $Q_b$  will be 0. The table in Figure 5.4b resembles a truth table. However, since it does not represent a combinational circuit in which the values of the outputs are determined solely by the current values of the inputs, it is often called a *characteristic table* rather than a truth table.

Figure 5.4c gives a timing diagram for the latch, assuming that the propagation delay through the NOR gates is negligible. Of course, in a real circuit the changes in the waveforms would be delayed according to the propagation delays of the gates. We assume that initially  $Q_a = 0$  and  $Q_b = 1$ . The state of the latch remains unchanged until time  $t_2$ , when  $S$  becomes equal to 1, causing  $Q_b$  to change to 0, which in turn causes  $Q_a$  to change to 1 because  $R + Q_b = 1$ . The causality relationship is indicated by the arrows in the diagram. When  $S$  goes to 0 at  $t_3$ , there is no change in the state because both  $S$  and  $R$  are then equal to 0. At  $t_4$  we have  $R = 1$ , which causes  $Q_a$  to go to 0, which in turn causes  $Q_b$  to go to 1 because  $S + Q_a = 1$ . At  $t_5$  both  $S$  and  $R$  are equal to 1, which forces both  $Q_a$  and  $Q_b$  to be equal to 0. As soon as  $S$  returns to 0, at  $t_6$ ,  $Q_b$  becomes equal to 1 again. At  $t_8$  we have  $S = 1$  and  $R = 0$ , which causes  $Q_b = 0$  and  $Q_a = 1$ . An interesting situation occurs at  $t_{10}$ . From  $t_9$  to  $t_{10}$  we have  $Q_a = Q_b = 0$  because  $R = S = 1$ . Now if both  $R$  and  $S$  change to 0 at  $t_{10}$ , both  $Q_a$  and  $Q_b$  will go to 1. But having both  $Q_a$  and  $Q_b$  equal to 1 will immediately force  $Q_a = Q_b = 0$ . There will be an oscillation between  $Q_a = Q_b = 0$  and  $Q_a = Q_b = 1$ . If the delays through the two NOR gates are exactly the same, the oscillation will continue indefinitely. In a real circuit there will invariably be some difference in the delays through these gates, and the latch will eventually settle into one of its two stable states, but we don't know which state it will be. This uncertainty is indicated in the waveforms by dashed lines.



**Figure 5.3** A memory element with NOR gates.



**Figure 5.4** A basic latch built with NOR gates.

The oscillations discussed above illustrate that even though the basic latch is a simple circuit, careful analysis has to be done to fully appreciate its behavior. In general, any circuit that contains one or more feedback paths, such that the state of the circuit depends on the propagation delays through logic gates, has to be designed carefully. We discuss timing issues in Section 5.15 and in Chapter 9.

The latch in Figure 5.4a can perform the functions needed for the memory element in Figure 5.1, by connecting the *Set* signal to the *S* input and *Reset* to the *R* input. The  $Q_a$  output provides the desired *On/Off* signal. To initialize the operation of the alarm system, the latch is reset. Thus the alarm is off. When the sensor generates the logic value 1, the latch is set and  $Q_a$  becomes equal to 1. This turns on the alarm mechanism. If the sensor output returns to 0, the latch retains its state where  $Q_a = 1$ ; hence the alarm remains turned

on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the *Reset* input equal to 1.

## 5.2 GATED SR LATCH

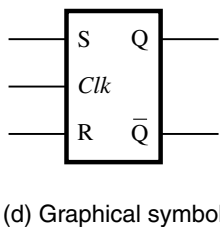
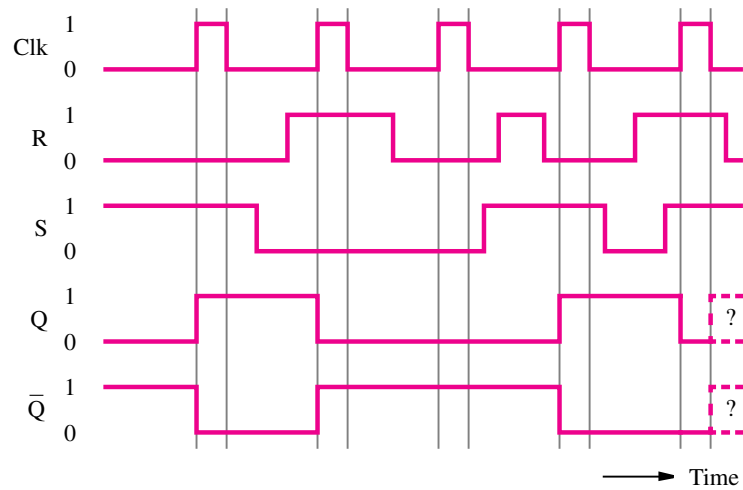
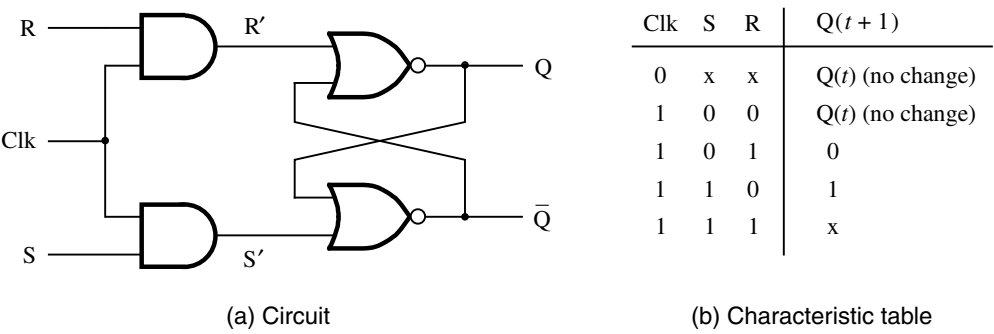
In Section 5.1 we saw that the basic SR latch can serve as a useful memory element. It remembers its state when both the *S* and *R* inputs are 0. It changes its state in response to changes in the signals on these inputs. The state changes occur at the time when the changes in the signals occur. If we cannot control the time of such changes, then we don't know when the latch may change its state.

In the alarm system of Figure 5.1, it may be desirable to be able to enable or disable the entire system by means of a control input, *Enable*. Thus when enabled, the system would function as described above. In the disabled mode, changing the *Set* input from 0 to 1 would not cause the alarm to turn on. The latch in Figure 5.4a cannot provide the desired operation. But the latch circuit can be modified to respond to the input signals *S* and *R* only when *Enable* = 1. Otherwise, it would maintain its state.

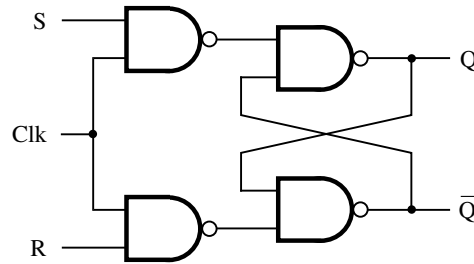
The modified circuit is depicted in Figure 5.5a. It includes two AND gates that provide the desired control. When the control signal *Clk* is equal to 0, the *S'* and *R'* inputs to the latch will be 0, regardless of the values of signals *S* and *R*. Hence the latch will maintain its existing state as long as *Clk* = 0. When *Clk* changes to 1, the *S'* and *R'* signals will be the same as the *S* and *R* signals, respectively. Therefore, in this mode the latch will behave as we described in Section 5.1. Note that we have used the name *Clk* for the control signal that allows the latch to be set or reset, rather than call it the *Enable* signal. The reason is that such circuits are often used in digital systems where it is desirable to allow the changes in the states of memory elements to occur only at well-defined time intervals, as if they were controlled by a clock. The control signal that defines these time intervals is usually called the *clock* signal. The name *Clk* is meant to reflect this nature of the signal.

Circuits of this type, which use a control signal, are called *gated latches*. Because our circuit exhibits set and reset capability, it is called a *gated SR latch*. Figure 5.5b describes its behavior. It defines the state of the *Q* output at time  $t + 1$ , namely,  $Q(t + 1)$ , as a function of the inputs *S*, *R*, and *Clk*. When *Clk* = 0, the latch will remain in the state it is in at time  $t$ , that is,  $Q(t)$ , regardless of the values of inputs *S* and *R*. This is indicated by specifying  $S = x$  and  $R = x$ , where  $x$  means that it does not matter if the signal value is 0 or 1. When *Clk* = 1, the circuit behaves as the basic latch in Figure 5.4. It is set by  $S = 1$  and reset by  $R = 1$ . The last row of the table, where  $S = R = 1$ , shows that the state  $Q(t + 1)$  is undefined because we don't know whether it will be 0 or 1. This corresponds to the situation described in Section 5.1 in conjunction with the timing diagram in Figure 5.4 at time  $t_{10}$ . At this time both *S* and *R* inputs go from 1 to 0, which causes the oscillatory behavior that we discussed. If  $S = R = 1$ , this situation will occur as soon as *Clk* goes from 1 to 0. To ensure a meaningful operation of the gated SR latch, it is essential to avoid the possibility of having both the *S* and *R* inputs equal to 1 when *Clk* changes from 1 to 0.

A timing diagram for the gated SR latch is given in Figure 5.5c. It shows *Clk* as a periodic signal that is equal to 1 at regular time intervals to suggest that this is how the



**Figure 5.5** Gated SR latch.



**Figure 5.6** Gated SR latch with NAND gates.

clock signal usually appears in a real system. The diagram presents the effect of several combinations of signal values. Observe that we have labeled one output as  $Q$  and the other as its complement  $\bar{Q}$ , rather than  $Q_a$  and  $Q_b$  as in Figure 5.4. Since the undefined mode, where  $S = R = 1$ , must be avoided in practice, the normal operation of the latch will have the outputs as complements of each other. Moreover, we will often say that the latch is *set* when  $Q = 1$ , and it is *reset* when  $Q = 0$ . A graphical symbol for the gated SR latch is given in Figure 5.5d.

### 5.2.1 GATED SR LATCH WITH NAND GATES

So far we have implemented the basic latch with cross-coupled NOR gates. We can also construct the latch with NAND gates. Using this approach, we can implement the gated SR latch as depicted in Figure 5.6. The behavior of this circuit is described by the table in Figure 5.5b. Note that in this circuit, the clock is gated by NAND gates, rather than by AND gates. Note also that the  $S$  and  $R$  inputs are reversed in comparison with the circuit in Figure 5.5a.

## 5.3 GATED D LATCH

In Section 5.2 we presented the gated SR latch and showed how it can be used as the memory element in the alarm system of Figure 5.1. This latch is useful for many other applications. In this section we describe another gated latch that is even more useful in practice. It has a single data input, called  $D$ , and it stores the value on this input, under the control of a clock signal. It is called a *gated D latch*.

To motivate the need for a gated D latch, consider the adder/subtractor unit discussed in Chapter 3 (Figure 3.12). When we described how that circuit is used to add numbers, we did not discuss what is likely to happen with the sum bits that are produced by the adder. Adder/subtractor units are often used as part of a computer. The result of an addition or subtraction operation is often used as an operand in a subsequent operation. Therefore, it is necessary to be able to remember the values of the sum bits generated by the adder until

they are needed again. We might think of using the basic latches to remember these bits, one bit per latch. In this context, instead of saying that a latch remembers the value of a bit, it is more illuminating to say that the latch *stores* the value of the bit or simply “stores the bit.” We should think of the latch as a storage element.

But can we obtain the desired operation using the basic latches? We can certainly reset all latches before the addition operation begins. Then we would expect that by connecting a sum bit to the  $S$  input of a latch, the latch would be set to 1 if the sum bit has the value 1; otherwise, the latch would remain in the 0 state. This would work fine if all sum bits are 0 at the start of the addition operation and, after some propagation delay through the adder, some of these bits become equal to 1 to give the desired sum. Unfortunately, the propagation delays that exist in the adder circuit cause a big problem in this arrangement. Suppose that we use a ripple-carry adder. When the  $X$  and  $Y$  inputs are applied to the adder, the sum outputs may alternate between 0 and 1 a number of times as the carries ripple through the circuit. The problem is that if we connect a sum bit to the  $S$  input of a latch, then if the sum bit is temporarily a 1 and then settles to 0 in the final result, the latch will remain set to 1 erroneously.

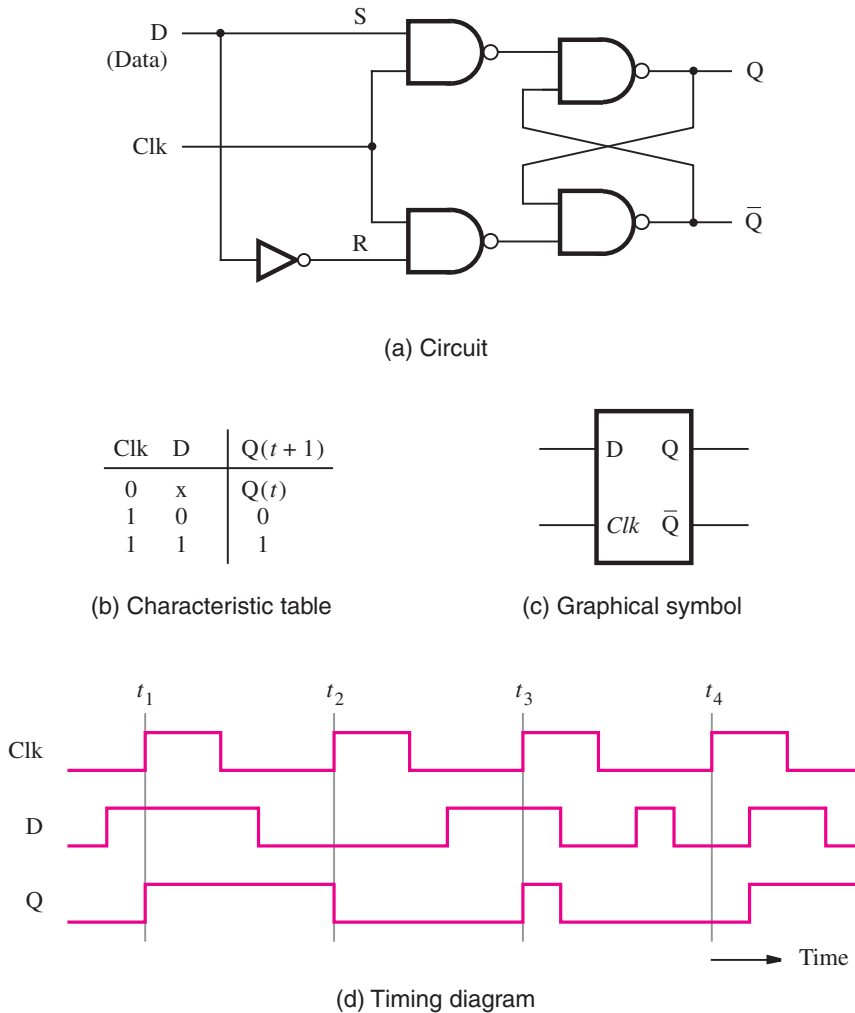
The problem caused by the alternating values of the sum bits in the adder could be solved by using the gated SR latches, instead of the basic latches. Then we could arrange that the clock signal is 0 during the time needed by the adder to produce a correct sum. After allowing for the maximum propagation delay in the adder circuit, the clock should go to 1 to store the values of the sum bits in the gated latches. As soon as the values have been stored, the clock can return to 0, which ensures that the stored values will be retained until the next time the clock goes to 1. To achieve the desired operation, we would also have to reset all latches to 0 prior to loading the sum-bit values into these latches. This is an awkward way of dealing with the problem, and it is preferable to use the gated D latches instead.

Figure 5.7a shows the circuit for a gated D latch. It is based on the gated SR latch, but instead of using the  $S$  and  $R$  inputs separately, it has just one data input,  $D$ . For convenience we have labeled the points in the circuit that are equivalent to the  $S$  and  $R$  inputs. If  $D = 1$ , then  $S = 1$  and  $R = 0$ , which forces the latch into the state  $Q = 1$ . If  $D = 0$ , then  $S = 0$  and  $R = 1$ , which causes  $Q = 0$ . Of course, the changes in state occur only when  $Clk = 1$ .

In this circuit it is impossible to have the troublesome situation where  $S = R = 1$ . In the gated D latch, the output  $Q$  merely tracks the value of the input  $D$  while  $Clk = 1$ . As soon as  $Clk$  goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the  $D$  input seen at the time the clock changes from 1 to 0. Figure 5.7 also gives the characteristic table, the graphical symbol, and a timing diagram for the gated D latch.

The timing diagram illustrates what happens if the  $D$  signal changes while  $Clk = 1$ . During the third clock pulse, starting at  $t_3$ , the output  $Q$  changes to 1 because  $D = 1$ . But midway through the pulse  $D$  goes to 0, which causes  $Q$  to go to 0. This value of  $Q$  is stored when  $Clk$  changes to 0. Now no further change in the state of the latch occurs until the next clock pulse, at  $t_4$ . The key point to observe is that as long as the clock has the value 1, the  $Q$  output follows the  $D$  input. But when the clock has the value 0, the  $Q$  output cannot change. The logic values are implemented as low and high voltage levels, as explained in detail in Appendix B. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be *level sensitive*. The circuits in Figures 5.5 through 5.7 are



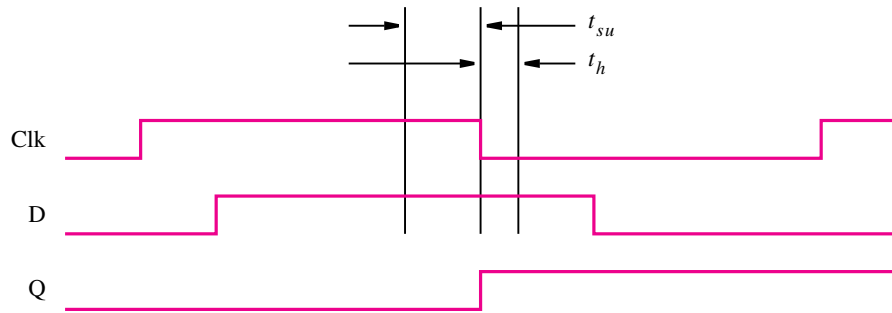


**Figure 5.7** Gated D latch.

level sensitive. We will show in Section 5.4 that it is possible to design storage elements for which the output changes only at the point in time when the clock changes from one value to the other. Such circuits are said to be *edge triggered*.

### 5.3.1 EFFECTS OF PROPAGATION DELAYS

In the previous discussion we ignored the effects of propagation delays. In practical circuits it is essential to take these delays into account. Consider the gated D latch in Figure 5.7a. It stores the value of the  $D$  input that is present at the time the clock signal changes from 1 to 0. It operates properly if the  $D$  signal is stable (that is, not changing) at the time  $Clk$



**Figure 5.8** Setup and hold times.

goes from 1 to 0. But it may lead to unpredictable results if the  $D$  signal also changes at this time. Therefore, the designer of a logic circuit that generates the  $D$  signal must ensure that this signal is stable when the critical change in the clock signal takes place.

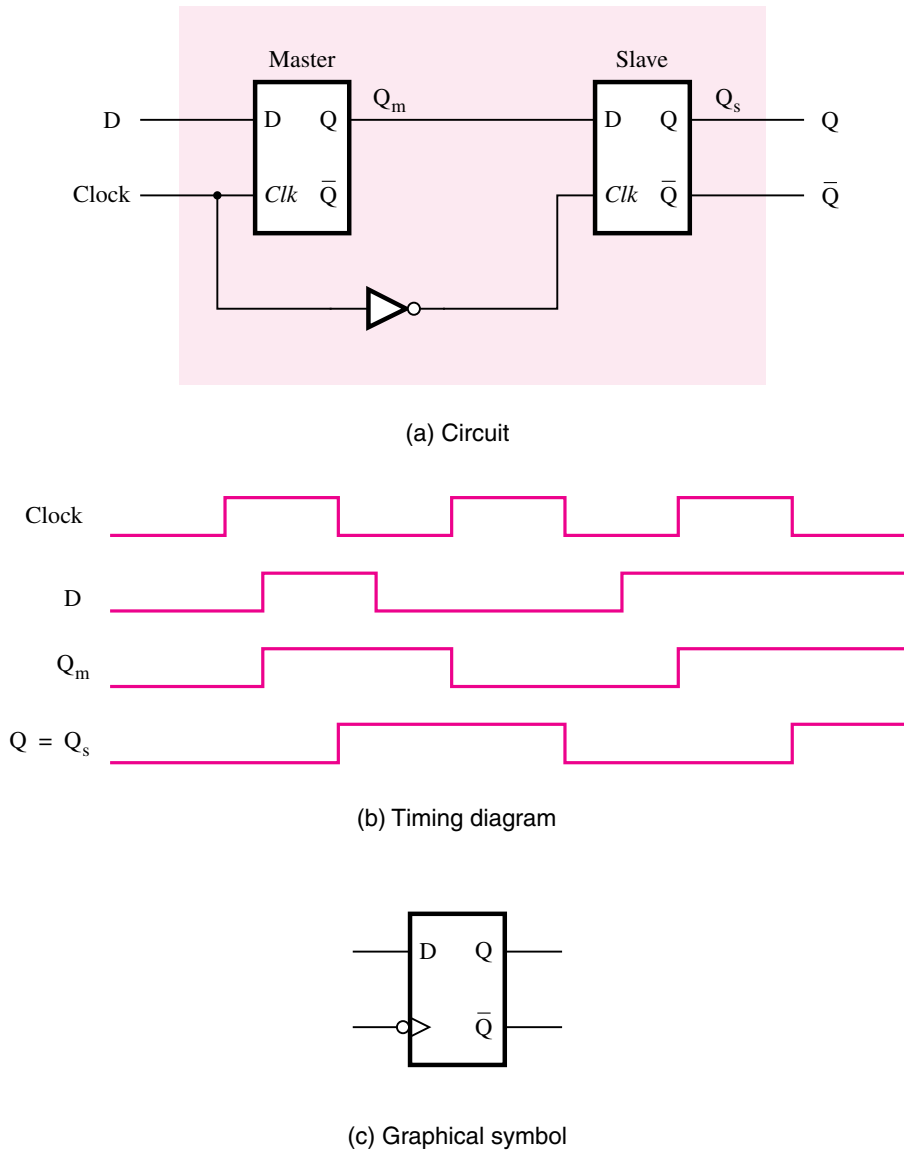
Figure 5.8 illustrates the critical timing region. The minimum time that the  $D$  signal must be stable prior to the negative edge of the  $Clk$  signal is called the *setup time*,  $t_{su}$ , of the latch. The minimum time that the  $D$  signal must remain stable after the negative edge of the  $Clk$  signal is called the *hold time*,  $t_h$ , of the latch. The values of  $t_{su}$  and  $t_h$  depend on the technology used. Manufacturers of integrated circuit chips provide this information on the data sheets that describe their chips. Typical values for a modern technology may be  $t_{su} = 0.3$  ns and  $t_h = 0.2$  ns. We will give examples of how setup and hold times affect the speed of operation of circuits in Section 5.15. The behavior of storage elements when setup or hold times are violated is discussed in Chapter 7.

## 5.4 EDGE-TRIGGERED D FLIP-FLOPS

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the clock signal is active (equal to 1 in our examples). As we will see in Sections 5.8 and 5.9, there is also a need for storage elements that can change their states no more than once during one clock cycle. We will now discuss circuits that exhibit such behavior.

### 5.4.1 MASTER-SLAVE D FLIP-FLOP

Consider the circuit given in Figure 5.9a, which consists of two gated D latches. The first, called *master*, changes its state while  $Clock = 1$ . The second, called *slave*, changes its state while  $Clock = 0$ . The operation of the circuit is such that when the clock is high, the master tracks the value of the  $D$  input signal and the slave does not change. Thus the value of  $Q_m$  follows any changes in  $D$ , and the value of  $Q_s$  remains constant. When the clock signal



**Figure 5.9** Master-slave D flip-flop.

changes to 0, the master stage stops following the changes in the  $D$  input. At the same time, the slave stage responds to the value of the signal  $Q_m$  and changes state accordingly. Since  $Q_m$  does not change while  $Clock = 0$ , the slave stage can undergo at most one change of state during a clock cycle. From the external observer's point of view, namely, the circuit connected to the output of the slave stage, the master-slave circuit changes its state at the negative-going edge of the clock. The *negative edge* is the edge where the clock signal

changes from 1 to 0. Regardless of the number of changes in the  $D$  input to the master stage during one clock cycle, the observer of the  $Q_s$  signal will see only the change that corresponds to the  $D$  input at the negative edge of the clock.

The circuit in Figure 5.9 is called a *master-slave D flip-flop*. The term *flip-flop* denotes a storage element that changes its output state at the edge of a controlling clock signal. The timing diagram for this flip-flop is shown in Figure 5.9b. A graphical symbol is given in Figure 5.9c. In the symbol we use the  $>$  mark to denote that the flip-flop responds to the “active edge” of the clock. We place a bubble on the clock input to indicate that the active edge for this particular circuit is the negative edge.

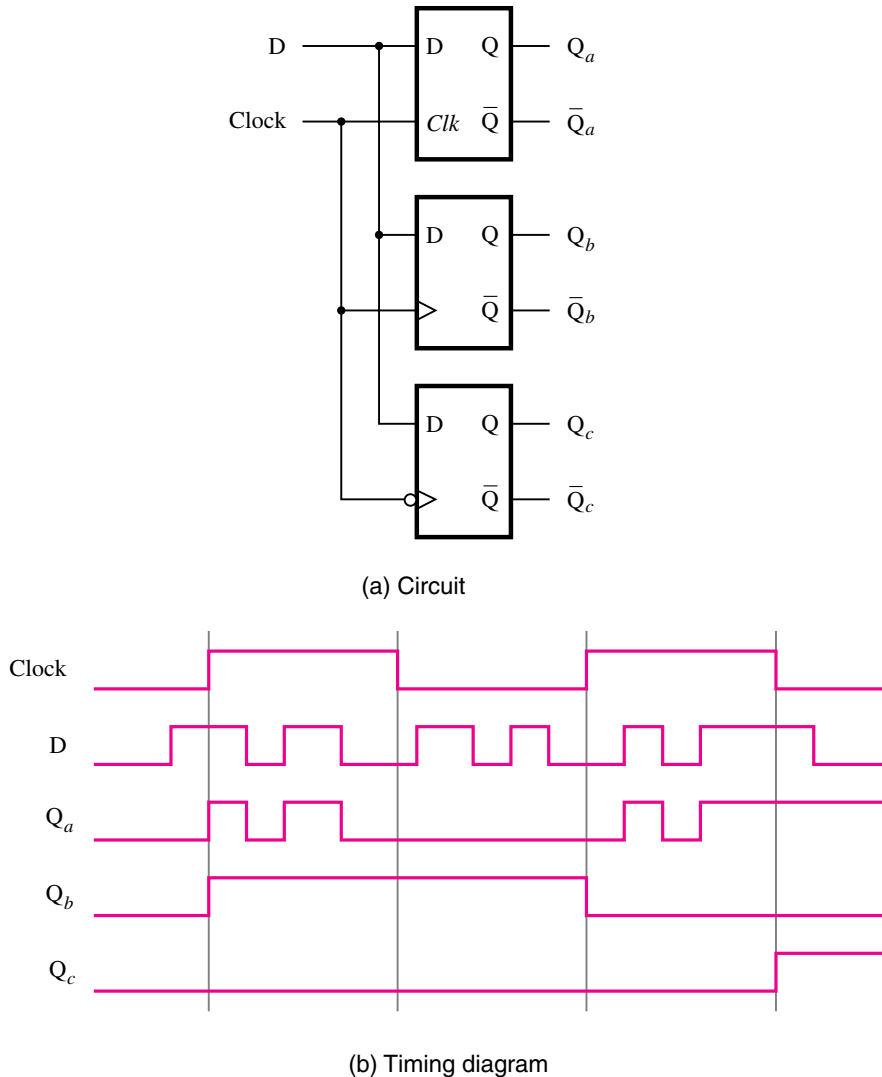
We can augment the circuit in Figure 5.9a by reversing the connections of the clock signal to the master and slave stages. That is, include an inverter in the clock input to the master stage and connect the uncomplemented clock signal to the slave stage. Now, the state of the master stage will be transferred into the slave stage when the clock signal changes from 0 to 1. This circuit behaves as a *positive-edge-triggered master-slave D flip-flop*. It can be represented by the graphical symbol in Figure 5.9c where the bubble on the clock input is removed.

### Level-Sensitive versus Edge-Triggered Storage Elements

At this point it is useful to compare the timing in the various storage elements that we have discussed. Figure 5.10 shows three different types of storage elements that are driven by the same data and clock inputs. The first element is a gated D latch, which is level sensitive. The second one is a positive-edge-triggered D flip-flop, and the third one is a negative-edge-triggered D flip-flop. To accentuate the differences between these storage elements, the  $D$  input changes its values more than once during each half of the clock cycle. Observe that the gated D latch follows the  $D$  input as long as the clock is high. The positive-edge-triggered flip-flop responds only to the value of  $D$  when the clock changes from 0 to 1. The negative-edge-triggered flip-flop responds only to the value of  $D$  when the clock changes from 1 to 0.

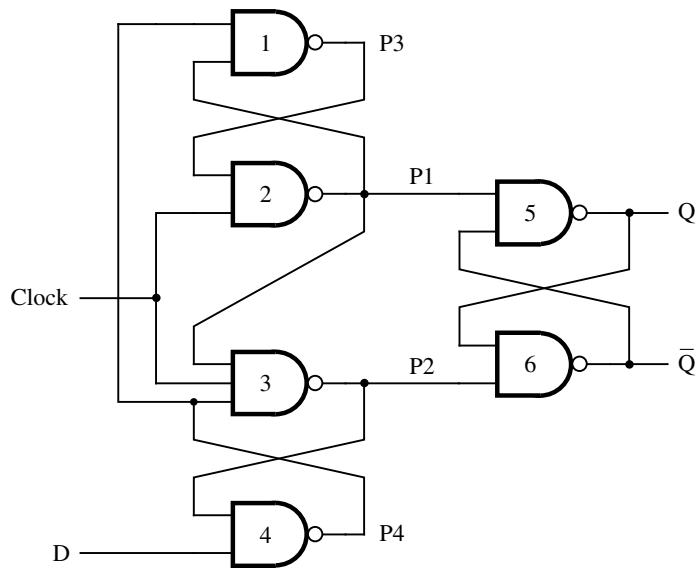
## 5.4.2 OTHER TYPES OF EDGE-TRIGGERED D FLIP-FLOPS

Master-slave flip-flops illustrate the concept of edge triggering very clearly. Other circuits have been used to accomplish the same task. Consider the circuit presented in Figure 5.11a. It requires only six NAND gates and, hence, fewer transistors than the master-slave circuit. The operation of the circuit is as follows. When  $Clock = 0$ , the outputs of gates 2 and 3 are high. Thus  $P1 = P2 = 1$ , which maintains the output latch, comprising gates 5 and 6, in its present state. At the same time, the signal  $P3$  is equal to  $D$ , and  $P4$  is equal to its complement  $\bar{D}$ . When  $Clock$  changes to 1, the following changes take place. The values of  $P3$  and  $P4$  are transmitted through gates 2 and 3 to cause  $P1 = \bar{D}$  and  $P2 = D$ , which sets  $Q = D$  and  $\bar{Q} = \bar{D}$ . To operate reliably,  $P3$  and  $P4$  must be stable when  $Clock$  changes from 0 to 1. Hence the setup time of the flip-flop is equal to the delay from the  $D$  input through gates 4 and 1 to  $P3$ . The hold time is given by the delay through gate 3 because once  $P2$  is stable, the changes in  $D$  no longer matter.

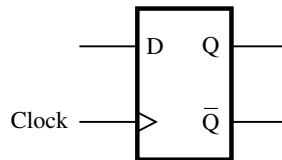


**Figure 5.10** Comparison of level-sensitive and edge-triggered D storage elements.

For proper operation it is necessary to show that after *Clock* changes to 1 any further changes in *D* will not affect the output latch as long as *Clock* = 1. We have to consider two cases. Suppose first that *D* = 0 at the positive edge of the clock. Then *P2* = 0, which will keep the output of gate 4 equal to 1 as long as *Clock* = 1, regardless of the value of the *D* input. The second case is if *D* = 1 at the positive edge of the clock. Then *P1* = 0, which forces the outputs of gates 1 and 3 to be equal to 1, regardless of the *D* input. Therefore, the flip-flop ignores changes in the *D* input while *Clock* = 1.



(a) Circuit



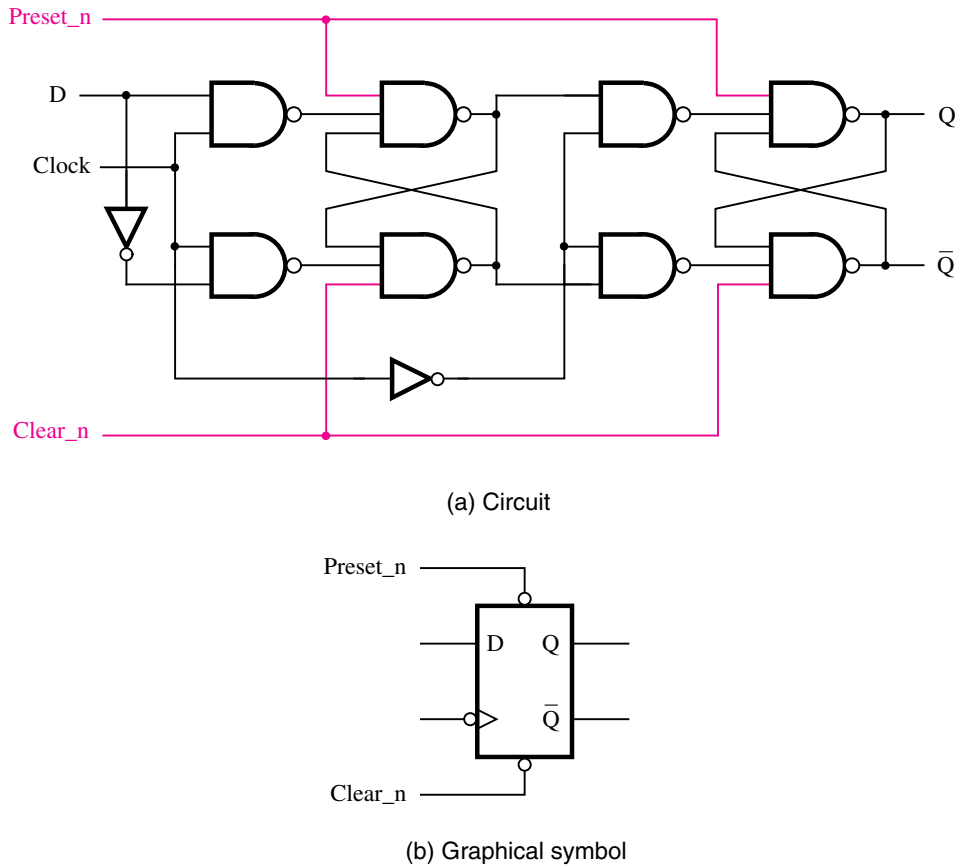
(b) Graphical symbol

**Figure 5.11** A positive-edge-triggered D flip-flop.

This circuit behaves as a positive-edge-triggered D flip-flop. A similar circuit, constructed with NOR gates, can be used as a negative-edge-triggered flip-flop.

### 5.4.3 D FLIP-FLOPS WITH CLEAR AND PRESET

Flip-flops are often used for implementation of circuits that can have many possible states, where the response of the circuit depends not only on the present values of the circuit's inputs but also on the particular state that the circuit is in at that time. We will discuss a general form of such circuits in Chapter 6. A simple example is a counter circuit that counts the number of occurrences of some event, perhaps passage of time. We will discuss counters in detail in Section 5.9. A counter comprises a number of flip-flops, whose outputs are interpreted as a number. The counter circuit has to be able to increment or decrement the number. It is also important to be able to force the counter into a known initial state (count).

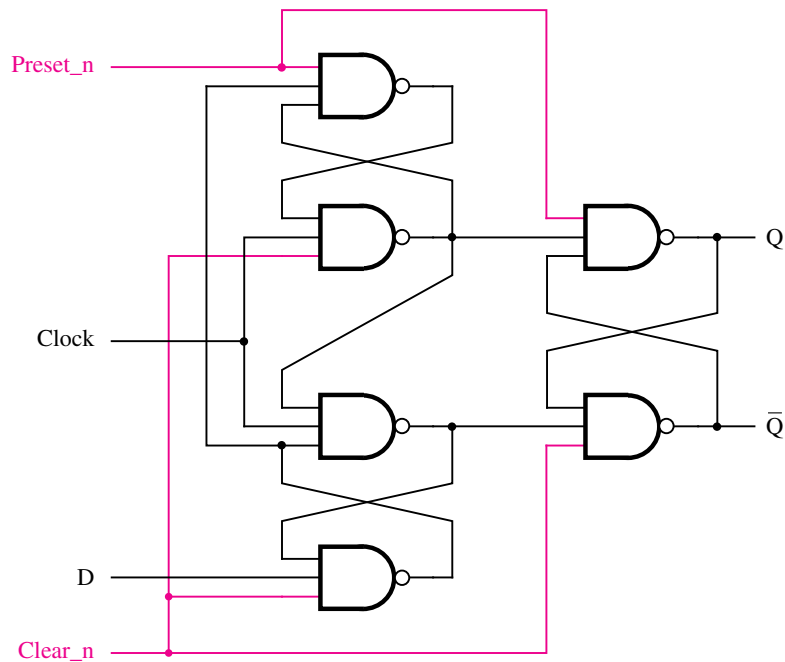


**Figure 5.12** Master-slave D flip-flop with *Clear* and *Preset*.

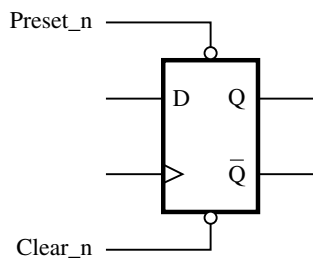
Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have  $Q = 0$ . It is equally useful to be able to preset each flip-flop to  $Q = 1$ , to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures 5.9 and 5.11 as follows.

Figure 5.12a shows an implementation of the circuit in Figure 5.9a using NAND gates. The master stage is just the gated D latch of Figure 5.7a. Instead of using another latch of the same type for the slave stage, we can use the slightly simpler gated SR latch of Figure 5.6. This eliminates one NOT gate from the circuit.

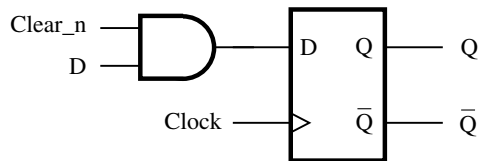
A simple way of providing the clear and preset capability is to add an extra input to each NAND gate in the cross-coupled latches, as indicated in blue. Placing a 0 on the *Clear<sub>n</sub>* input will force the flip-flop into the state  $Q = 0$ . If *Clear<sub>n</sub>* = 1, then this input will have no effect on the NAND gates. Similarly, *Preset<sub>n</sub>* = 0 forces the flip-flop into the state  $Q = 1$ , while *Preset<sub>n</sub>* = 1 has no effect. To denote that the *Clear<sub>n</sub>* and *Preset<sub>n</sub>* inputs are active when their value is 0, we appended the letter *n* (for “negative”) to these names. We should note that the circuit that uses this flip-flop should not try to force both



(a) Circuit



(b) Graphical symbol



(c) Adding a synchronous clear

**Figure 5.13** Positive-edge-triggered D flip-flop with *Clear* and *Preset*.

$\text{Clear}_n$  and  $\text{Preset}_n$  to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 5.12b.

A similar modification can be done on the edge-triggered flip-flop of Figure 5.11a, as indicated in Figure 5.13a. Again, both  $\text{Clear}_n$  and  $\text{Preset}_n$  inputs are active low. They do not disturb the flip-flop when they are equal to 1.

In the circuits in Figures 5.12a and 5.13a, the effect of a low signal on either the  $\text{Clear}_n$  or  $\text{Preset}_n$  input is immediate. For example, if  $\text{Clear}_n = 0$  then the flip-flop goes into



the state  $Q = 0$  immediately, regardless of the value of the clock signal. In such a circuit, where the *Clear\_n* signal is used to clear a flip-flop without regard to the clock signal, we say that the flip-flop has an *asynchronous clear*. In practice, it is often preferable to clear the flip-flops on the active edge of the clock. Such *synchronous clear* can be accomplished as shown in Figure 5.13c. The flip-flop operates normally when the *Clear\_n* input is equal to 1. But if *Clear\_n* goes to 0, then on the next positive edge of the clock the flip-flop will be cleared to 0. We will examine the clearing of flip-flops in more detail in Section 5.10.

#### 5.4.4 FLIP-FLOP TIMING PARAMETERS

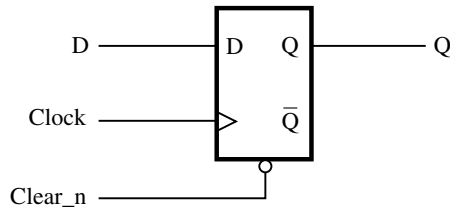
In Section 5.3.1 we discussed timing issues related to latch circuits. In practice such issues are equally important for circuits with flip-flops. Figure 5.14a shows a positive-edge triggered flip-flop with asynchronous clear, and part (b) of the figure illustrates some important timing parameters for this flip-flop. Data is loaded into the *D* input of the flip-flop on a positive clock edge, and this logic value must be stable during the setup time,  $t_{su}$ , before the clock edge occurs. The data must remain stable during the hold time,  $t_h$ , after the edge. If the setup or hold requirements are not adhered to in a circuit that uses this flip-flop, then it may enter an unstable condition known as *metastability*; we discuss this concept in Chapter 7.

As indicated in Figure 5.14, a clock-to-*Q* propagation delay,  $t_{cQ}$ , is incurred before the value of *Q* changes after a positive clock edge. In general, the delay may not be exactly the same for the cases when *Q* changes from 1 to 0 or 0 to 1, but we assume for simplicity that these delays are equal. For the flip-flops in a commercial chip, two values are usually specified for  $t_{cQ}$ , representing the maximum and minimum delays that may occur in practice. Specifying a range of values when estimating the delays in a chip is a common practice due to many sources of variation in delay that are caused by the chip manufacturing process. In Section 5.15 we provide some examples that illustrate the effects of flip-flop timing parameters on the operation of circuits.

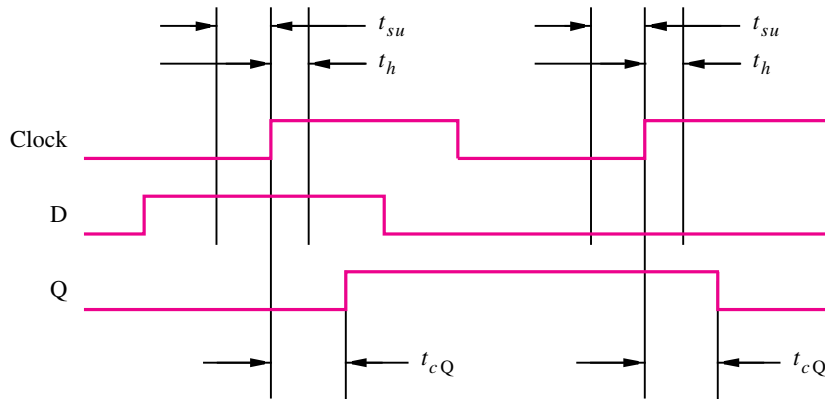
---

## 5.5 T FLIP-FLOP

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 5.15a. This circuit uses a positive-edge-triggered D flip-flop. The *feedback* connections make the input signal *D* equal to either the value of *Q* or  $\bar{Q}$  under the control of the signal that is labeled *T*. On each positive edge of the clock, the flip-flop may change its state  $Q(t)$ . If  $T = 0$ , then  $D = Q$  and the state will remain the same, that is,  $Q(t + 1) = Q(t)$ . But if  $T = 1$ , then  $D = \bar{Q}$  and the new state will be  $Q(t + 1) = \bar{Q}(t)$ . Therefore, the overall operation of the circuit is that it retains its present state if  $T = 0$ , and it reverses its present state if  $T = 1$ .



(a) D flip-flop with asynchronous clear



(b) Timing diagram

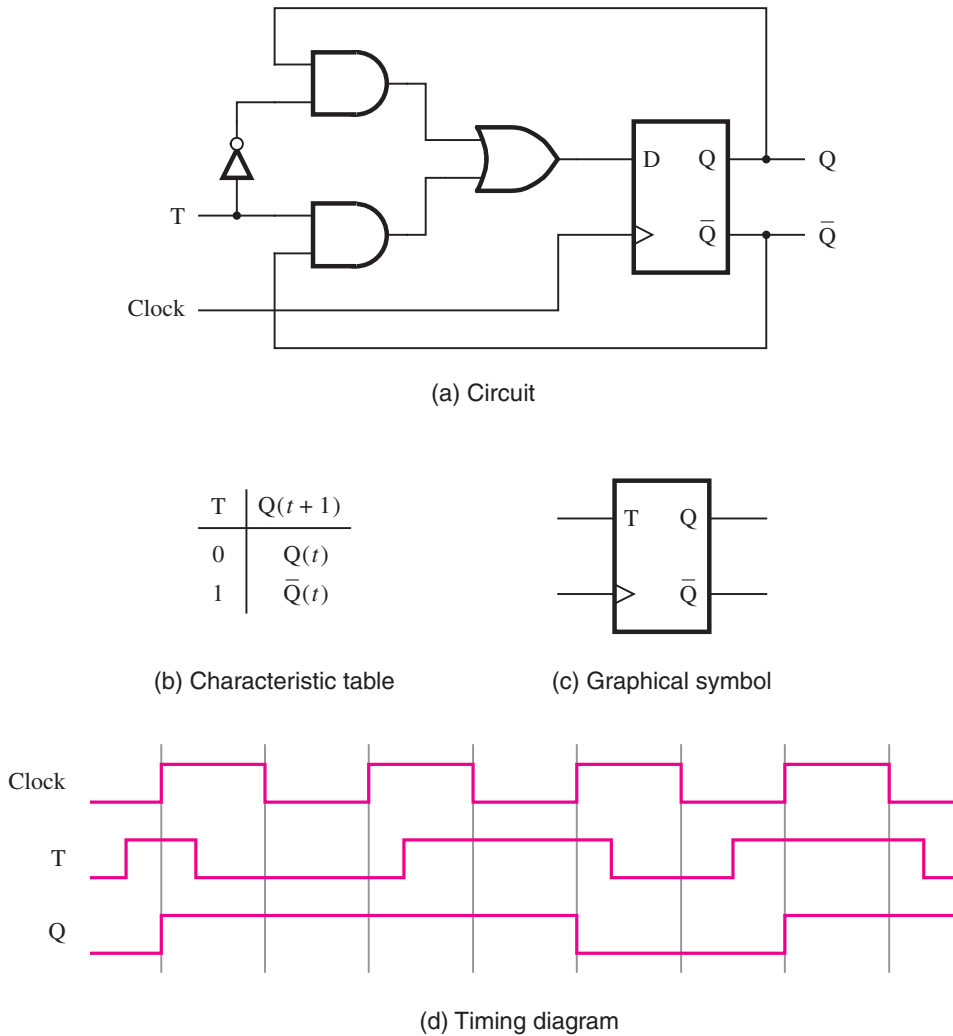
**Figure 5.14** Flip-flop timing parameters.

The operation of the circuit is specified in the form of a characteristic table in Figure 5.15b. Any circuit that implements this table is called a *T flip-flop*. The name T flip-flop derives from the behavior of the circuit, which “toggles” its state when  $T = 1$ . The toggle feature makes the T flip-flop a useful element for building counter circuits, as we will see in Section 5.9.

## 5.6 JK FLIP-FLOP

Another interesting circuit can be derived from Figure 5.15a. Instead of using a single control input,  $T$ , we can use two inputs,  $J$  and  $K$ , as indicated in Figure 5.16a. For this circuit the input  $D$  is defined as

$$D = J\bar{Q} + \bar{K}Q$$



**Figure 5.15** T flip-flop.

A corresponding characteristic table is given in Figure 5.16b. The circuit is called a *JK flip-flop*. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop, where  $J = S$  and  $K = R$ , for all input values except  $J = K = 1$ . For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop.

The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the  $J$  and  $K$  inputs together.

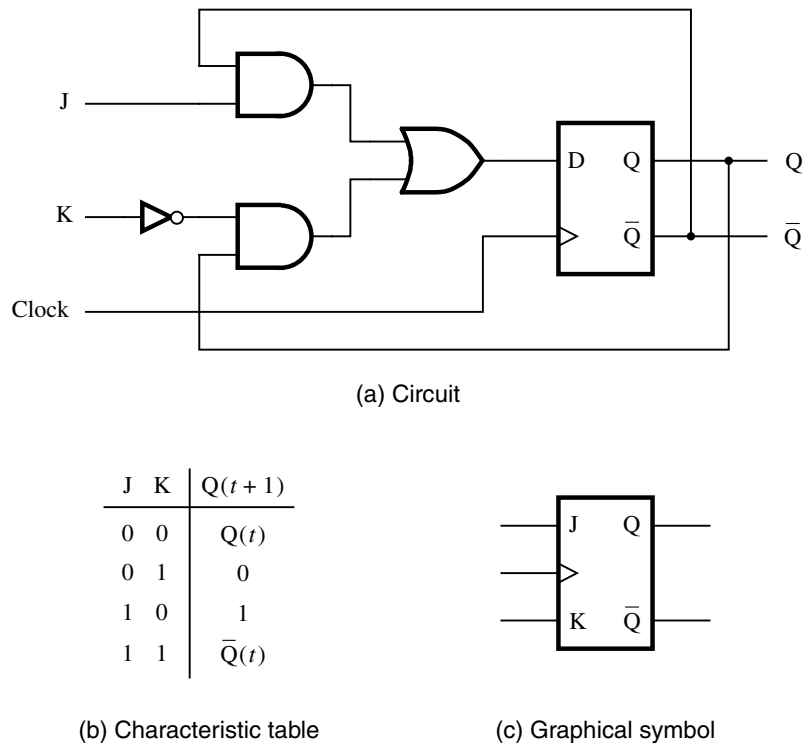


Figure 5.16 JK flip-flop.

5.7 SUMMARY OF TERMINOLOGY

We have used the terminology that is quite common. But the reader should be aware that different interpretations of the terms *latch* and *flip-flop* can be found in the literature. Our terminology can be summarized as follows:

- Basic latch** is a feedback connection of two NOR gates or two NAND gates, which can store one bit of information. It can be set to 1 using the *S* input and reset to 0 using the *R* input.
- Gated latch** is a basic latch that includes input gating and a control input signal. The latch retains its existing state when the control input is equal to 0. Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock. We considered two types of gated latches:
- **Gated SR latch** uses the *S* and *R* inputs to set the latch to 1 or reset it to 0, respectively.
  - **Gated D latch** uses the *D* input to force the latch into a state that has the same logic value as the *D* input.

A **flip-flop** is a storage element that can have its output state changed only on the edge of the controlling clock signal. If the state changes when the clock signal goes from 0 to 1, we say that the flip-flop is **positive-edge triggered**. If the state changes when the clock signal goes from 1 to 0, we say that the flip-flop is **negative-edge triggered**.

---

## 5.8 REGISTERS

A flip-flop stores one bit of information. When a set of  $n$  flip-flops is used to store  $n$  bits of information, such as an  $n$ -bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register, and each flip-flop operates as described in the previous sections. The term register is merely a convenience for referring to  $n$ -bit structures consisting of flip-flops.

### 5.8.1 SHIFT REGISTER

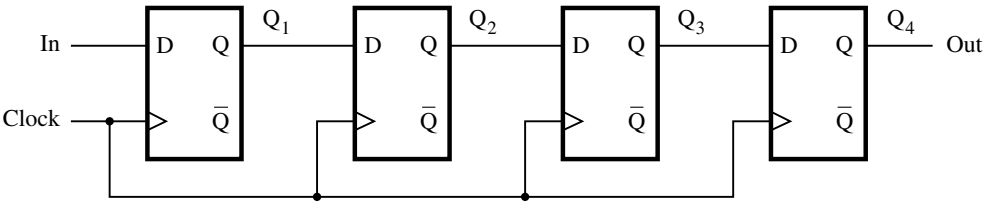
In Section 3.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to the right. A register that provides the ability to shift its contents is called a *shift register*.

Figure 5.17a shows a four-bit shift register that is used to shift its contents one bit-position to the right. The data bits are loaded into the shift register in a serial fashion using the *In* input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. An illustration of the transfer is given in Figure 5.17b, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.

To implement a shift register it is necessary to use flip-flops. The level-sensitive gated latches are not suitable, because a change in the value of *In* would propagate through more than one latch during the time when the clock is equal to 1.

### 5.8.2 PARALLEL-ACCESS SHIFT REGISTER

In computer systems it is often necessary to transfer  $n$ -bit data items. This may be done by transmitting all bits at once using  $n$  separate wires, in which case we say that the transfer is performed in *parallel*. But it is also possible to transfer all bits using a single wire, by performing the transfer one bit at a time, in  $n$  consecutive clock cycles. We refer to this scheme as *serial* transfer. To transfer an  $n$ -bit data item serially, we can use a shift register that can be loaded with all  $n$  bits in parallel (in one clock cycle). Then during the next  $n$  clock cycles, the contents of the register can be shifted out for serial transfer. The reverse operation is also needed. If bits are received serially, then after  $n$  clock cycles the contents of the register can be accessed in parallel as an  $n$ -bit item.



(a) Circuit

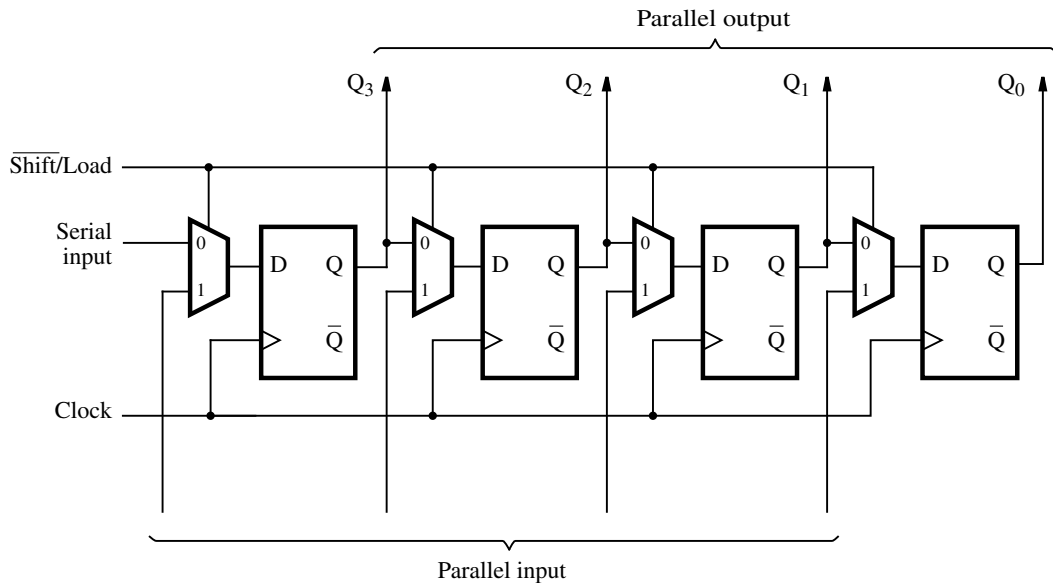
	In	Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub> = Out
$t_0$	1	0	0	0	0
$t_1$	0	1	0	0	0
$t_2$	1	0	1	0	0
$t_3$	1	1	0	1	0
$t_4$	1	1	1	0	1
$t_5$	0	1	1	1	0
$t_6$	0	0	1	1	1
$t_7$	0	0	0	1	1

(b) A sample sequence

**Figure 5.17** A simple shift register.

Figure 5.18 shows a four-bit shift register that provides the parallel access. A 2-to-1 multiplexer on its *D* input allows each flip-flop to be connected to two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal *Shift/Load* is used to select the mode of operation. If *Shift/Load* = 0, then the circuit operates as a shift register. If *Shift/Load* = 1, then the parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

In Figure 5.18 we have chosen to label the flip-flops' outputs as  $Q_3, \dots, Q_0$  because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of  $Q_0$  during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then accessed in parallel is called a series-to-parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 5.18 can perform both of these functions.



**Figure 5.18** Parallel-access shift register.

## 5.9 COUNTERS

In Chapter 3 we dealt with circuits that perform arithmetic operations. We showed how adder/subtractor circuits can be designed, either using a simple cascaded (ripple-carry) structure that is inexpensive but slow or using a more complex carry-lookahead structure that is both more expensive and faster. In this section we examine special types of addition and subtraction operations, which are used for the purpose of counting. In particular, we want to design circuits that can increment or decrement a count by 1. Counter circuits are used in digital systems for many purposes. They may count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, keep track of time elapsed between specific events, and so on.

Counters can be implemented using the adder/subtractor circuits discussed in Chapter 3 and the registers discussed in Section 5.8. However, since we only need to change the contents of a counter by 1, it is not necessary to use such elaborate circuits. Instead, we can use much simpler circuits that have a significantly lower cost. We will show how the counter circuits can be designed using T and D flip-flops.

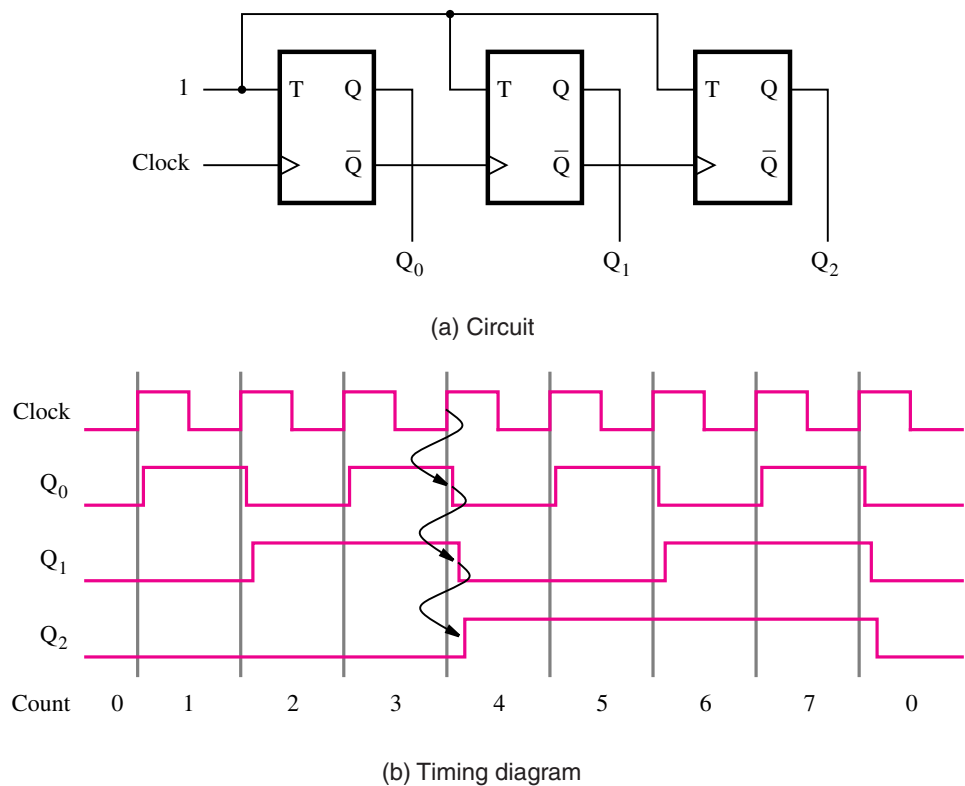
### 5.9.1 ASYNCHRONOUS COUNTERS

The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation.

### Up-Counter with T Flip-Flops

Figure 5.19a gives a three-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The  $T$  input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called *Clock*. Thus the clock input of the first flip-flop is connected to the *Clock* line. The other two flip-flops have their clock inputs driven by the  $\bar{Q}$  output of the preceding flip-flop. Therefore, they toggle their state whenever the preceding flip-flop changes its state from  $Q = 1$  to  $Q = 0$ , which results in a positive edge of the  $\bar{Q}$  signal.

Figure 5.19b shows a timing diagram for the counter. The value of  $Q_0$  toggles once each clock cycle. The change takes place shortly after the positive edge of the *Clock* signal. The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by  $\bar{Q}_0$ , the value of  $Q_1$  changes shortly after the negative edge of the  $Q_0$  signal. Similarly, the value of  $Q_2$  changes shortly after the negative edge of the  $Q_1$  signal. If we look at the values  $Q_2Q_1Q_0$  as the count, then the timing diagram indicates that the counting



**Figure 5.19** A three-bit up-counter.

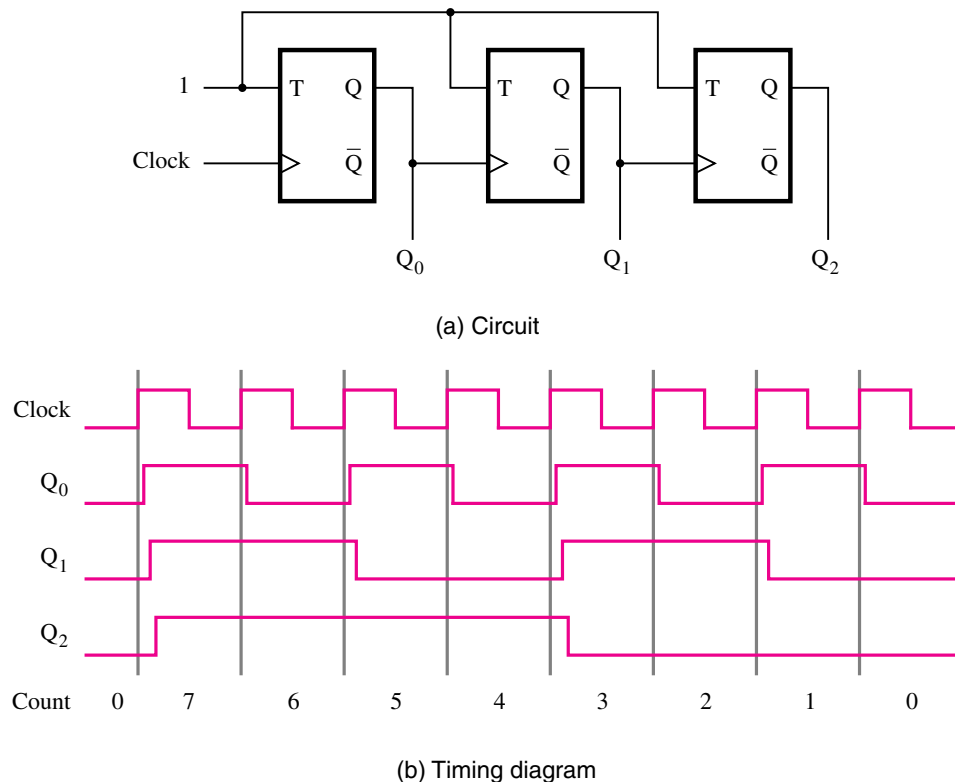


sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an *up-counter*.

The counter in Figure 5.19a has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 5.19b, this change requires the toggling of the states of all three flip-flops. The change in  $Q_0$  is observed only after a propagation delay from the positive edge of *Clock*. The  $Q_1$  and  $Q_2$  flip-flops have not yet changed; hence for a brief time the count is  $Q_2Q_1Q_0 = 010$ . The change in  $Q_1$  appears after a second propagation delay, at which point the count is 000. Finally, the change in  $Q_2$  occurs after a third delay, at which point the stable state of the circuit is reached and the count is 100. This behavior is similar to the rippling of carries in the ripple-carry adder circuit of Figure 3.5. The circuit in Figure 5.19a is an *asynchronous counter*, or a *ripple counter*.

### Down-Counter with T Flip-Flops

A slight modification of the circuit in Figure 5.19a is presented in Figure 5.20a. The only difference is that in Figure 5.20a the clock inputs of the second and third flip-flops are



**Figure 5.20** A three-bit down-counter.

driven by the  $Q$  outputs of the preceding stages, rather than by the  $\overline{Q}$  outputs. The timing diagram, given in Figure 5.20*b*, shows that this circuit counts in the sequence 0, 7, 6, 5, 4, 3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a *down-counter*.

It is possible to combine the functionality of the circuits in Figures 5.19*a* and 5.20*a* to form a counter that can count either up or down. Such a counter is called an *up/down-counter*. We leave the derivation of this counter as an exercise for the reader (Problem 5.15).

5.9.2 SYNCHRONOUS COUNTERS

The asynchronous counters in Figures 5.19*a* and 5.20*a* are simple, but not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the approach described below.

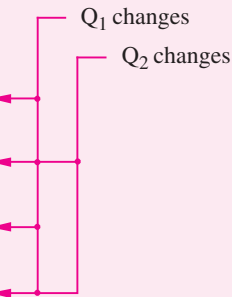
Synchronous Counter with T Flip-Flops

Table 5.1 shows the contents of a three-bit up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of the table, it is apparent that bit  $Q_0$  changes on each clock cycle. Bit  $Q_1$  changes only when  $Q_0 = 1$ . Bit  $Q_2$  changes only when both  $Q_1$  and  $Q_0$  are equal to 1. In general, for an  $n$ -bit up-counter, a given flip-flop changes its state only when all the preceding flip-flops are in the state  $Q = 1$ . Therefore, if we use T flip-flops to realize the counter, then the  $T$  inputs are defined as

$$T_0 = 1$$
$$T_1 = Q_0$$
$$T_2 = Q_0Q_1$$

Table 5.1 Derivation of the synchronous up-counter.

Clock cycle	$Q_2$	$Q_1$	$Q_0$
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0



$$T_3 = Q_0 Q_1 Q_2$$

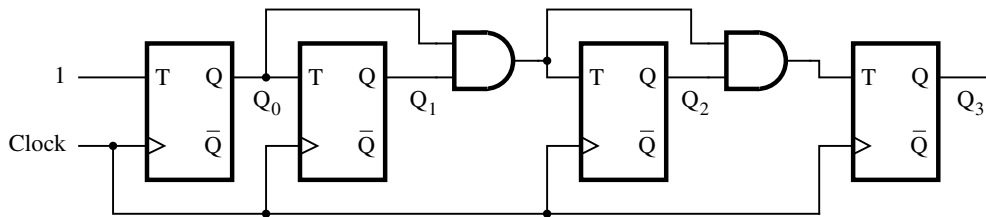
.

.

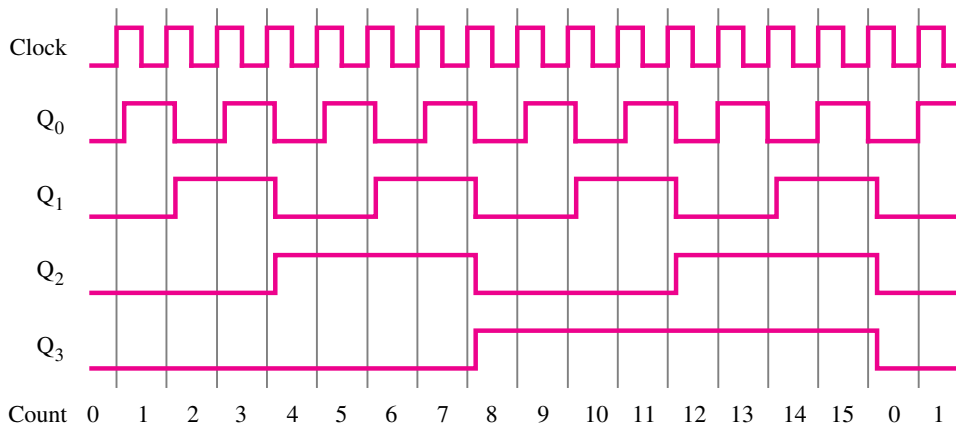
.

$$T_n = Q_0 Q_1 \cdots Q_{n-1}$$

An example of a four-bit counter based on these expressions is given in Figure 5.21a. Instead of using AND gates of increased size for each stage, we use a factored arrangement as shown in the figure. This arrangement does not slow down the response of the counter, because all flip-flops change their states after a propagation delay from the positive edge of the clock. Note that a change in the value of  $Q_0$  may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain

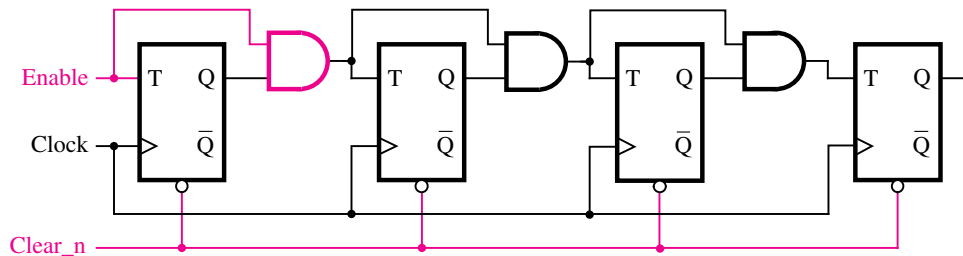


(a) Circuit



(b) Timing diagram

**Figure 5.21** A four-bit synchronous up-counter.



**Figure 5.22** Inclusion of Enable and Clear capability.

amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops.

Figure 5.21b gives a timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Because all changes take place with the same delay after the active edge of the *Clock* signal, the circuit is called a *synchronous counter*.

### Enable and Clear Capability

The counters in Figures 5.19 through 5.21 change their contents in response to each clock pulse. Often it is desirable to be able to inhibit counting, so that the count remains in its present state. This may be accomplished by including an *Enable* control signal, as indicated in Figure 5.22. The circuit is the counter of Figure 5.21, where the *Enable* signal controls directly the *T* input of the first flip-flop. Connecting the *Enable* also to the AND-gate chain means that if *Enable* = 0, then all *T* inputs will be equal to 0. If *Enable* = 1, then the counter operates as explained previously.

In many applications it is necessary to start with the count equal to zero. This is easily achieved if the flip-flops can be cleared, as explained in Section 5.4.3. The clear inputs on all flip-flops can be tied together and driven by a *Clear\_n* control input.

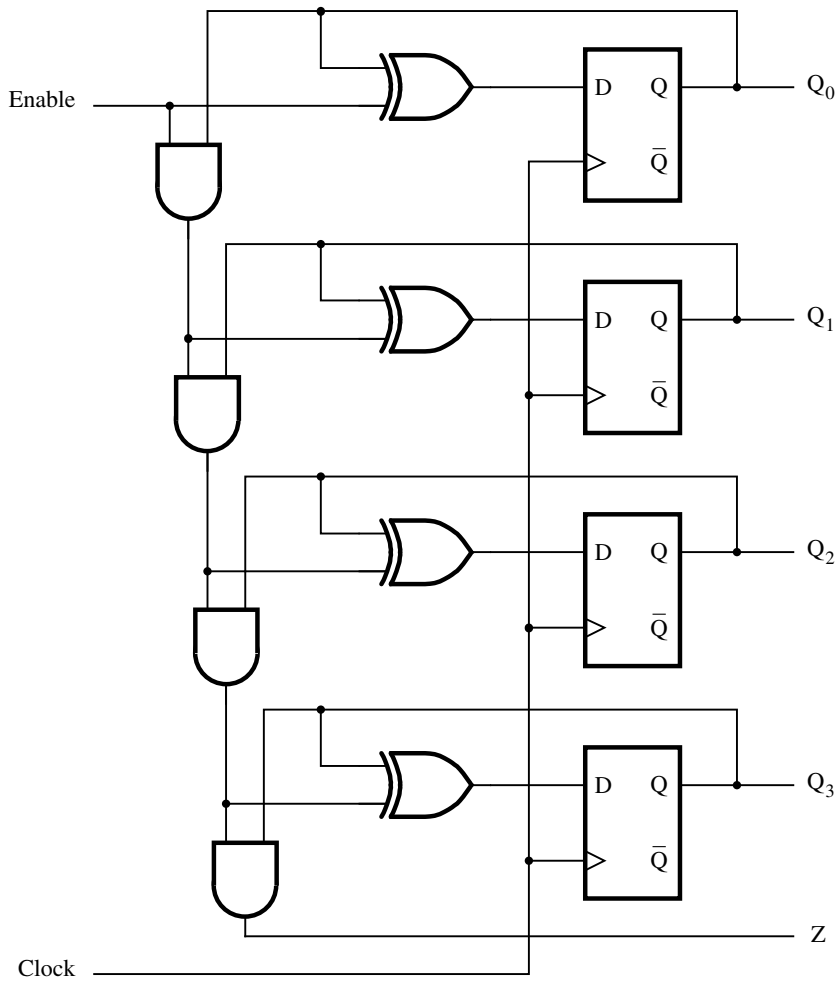
### Synchronous Counter with D Flip-Flops

While the toggle feature makes T flip-flops a natural choice for the implementation of counters, it is also possible to build counters using other types of flip-flops. The JK flip-flops can be used in exactly the same way as the T flip-flops because if the *J* and *K* inputs are tied together, a JK flip-flop becomes a T flip-flop. We will now consider using D flip-flops for this purpose.

It is not obvious how D flip-flops can be used to implement a counter. We will present a formal method for deriving such circuits in Chapter 6. Here we will present a circuit structure that meets the requirements but will leave the derivation for Chapter 6. Figure 5.23 gives a four-bit up-counter that counts in the sequence 0, 1, 2, . . . , 14, 15, 0, 1, and so on. The count is indicated by the flip-flop outputs  $Q_3Q_2Q_1Q_0$ . If we assume that *Enable* = 1, then the *D* inputs of the flip-flops are defined by the expressions

$$D_0 = Q_0 \oplus 1 = \overline{Q_0}$$

$$D_1 = Q_1 \oplus Q_0$$



**Figure 5.23** A four-bit counter with D flip-flops.

$$D_2 = Q_2 \oplus Q_1 Q_0$$

$$D_3 = Q_3 \oplus Q_2 Q_1 Q_0$$

For a larger counter the  $i$ th stage is defined by

$$D_i = Q_i \oplus Q_{i-1} Q_{i-2} \cdots Q_1 Q_0$$

We will show how to derive these equations in Chapter 6.

We have included the *Enable* control signal in Figure 5.23 so that the counter counts the clock pulses only if *Enable* = 1. In effect, the above equations are modified to implement the circuit in the figure as follows

$$\begin{aligned} D_0 &= Q_0 \oplus \text{Enable} \\ D_1 &= Q_1 \oplus Q_0 \cdot \text{Enable} \\ D_2 &= Q_2 \oplus Q_1 \cdot Q_0 \cdot \text{Enable} \\ D_3 &= Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot \text{Enable} \end{aligned}$$

The operation of the counter is based on our observation for Table 5.1 that the state of the flip-flop in stage *i* changes only if all preceding flip-flops are in the state *Q* = 1. This makes the output of the AND gate that feeds stage *i* equal to 1, which causes the output of the XOR gate connected to *D<sub>i</sub>* to be equal to  $\overline{Q}_i$ . Otherwise, the output of the XOR gate provides *D<sub>i</sub>* = *Q<sub>i</sub>*, and the flip-flop remains in the same state.

We have included an extra AND gate that produces the output *Z*. This signal makes it easy to concatenate two such counters to create a larger counter. It is also useful in applications where it is necessary to detect the state where the count has reached its maximum value (all 1s) and will go to 0 in the next clock cycle.

The counter in Figure 5.23 is essentially the same as the circuit in Figure 5.22. We showed in Figure 5.15a that a T flip-flop can be formed from a D flip-flop by providing the extra gating that gives

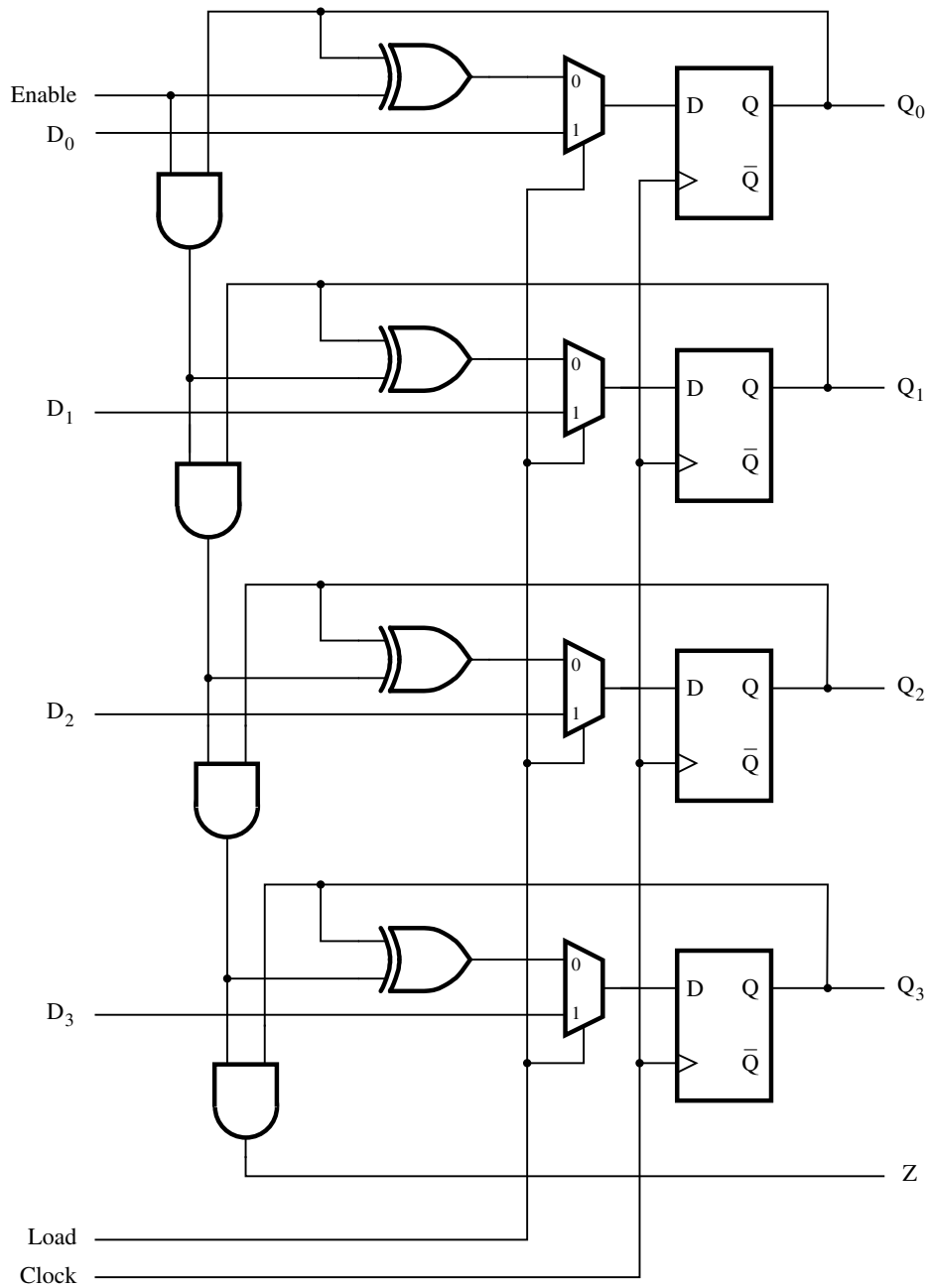
$$\begin{aligned} D &= Q\overline{T} + \overline{Q}T \\ &= Q \oplus T \end{aligned}$$

Thus in each stage in Figure 5.23, the D flip-flop and the associated XOR gate implement the functionality of a T flip-flop.

### 5.9.3 COUNTERS WITH PARALLEL LOAD

Often it is necessary to start counting with the initial count being equal to 0. This state can be achieved by using the capability to clear the flip-flops as indicated in Figure 5.22. But sometimes it is desirable to start with a different count. To allow this mode of operation, a counter circuit must have some inputs through which the initial count can be loaded. Using the *Clear* and *Preset* inputs for this purpose is a possibility, but a better approach is discussed below.

The circuit of Figure 5.23 can be modified to provide the parallel-load capability as shown in Figure 5.24. A two-input multiplexer is inserted before each *D* input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when *Load* = 0. A new initial value, *D<sub>3</sub>D<sub>2</sub>D<sub>1</sub>D<sub>0</sub>*, is loaded into the counter when *Load* = 1.

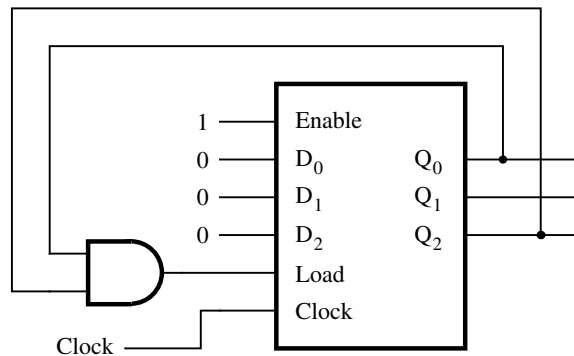


**Figure 5.24** A counter with parallel-load capability.

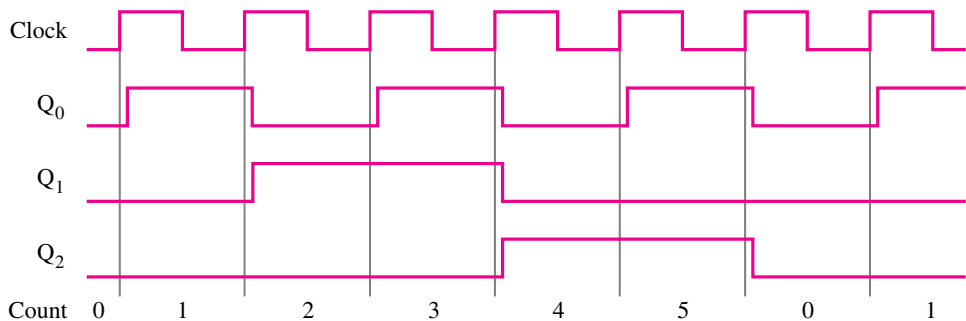
## 5.10 RESET SYNCHRONIZATION

We have already mentioned that it is important to be able to clear, or *reset*, the contents of a counter prior to commencing a counting operation. This can be done using the clear capability of the individual flip-flops. But we may also be interested in resetting the count to 0 during the normal counting process. An  $n$ -bit up-counter functions naturally as a modulo- $2^n$  counter. Suppose that we wish to have a counter that counts modulo some base that is not a power of 2. For example, we may want to design a modulo-6 counter, for which the counting sequence is 0, 1, 2, 3, 4, 5, 0, 1, and so on.

The most straightforward approach is to recognize when the count reaches 5 and then reset the counter. An AND gate can be used to detect the occurrence of the count of 5. Actually, it is sufficient to ascertain that  $Q_2 = Q_0 = 1$ , which is true only for 5 in our desired counting sequence. A circuit based on this approach is given in Figure 5.25a. It



(a) Circuit



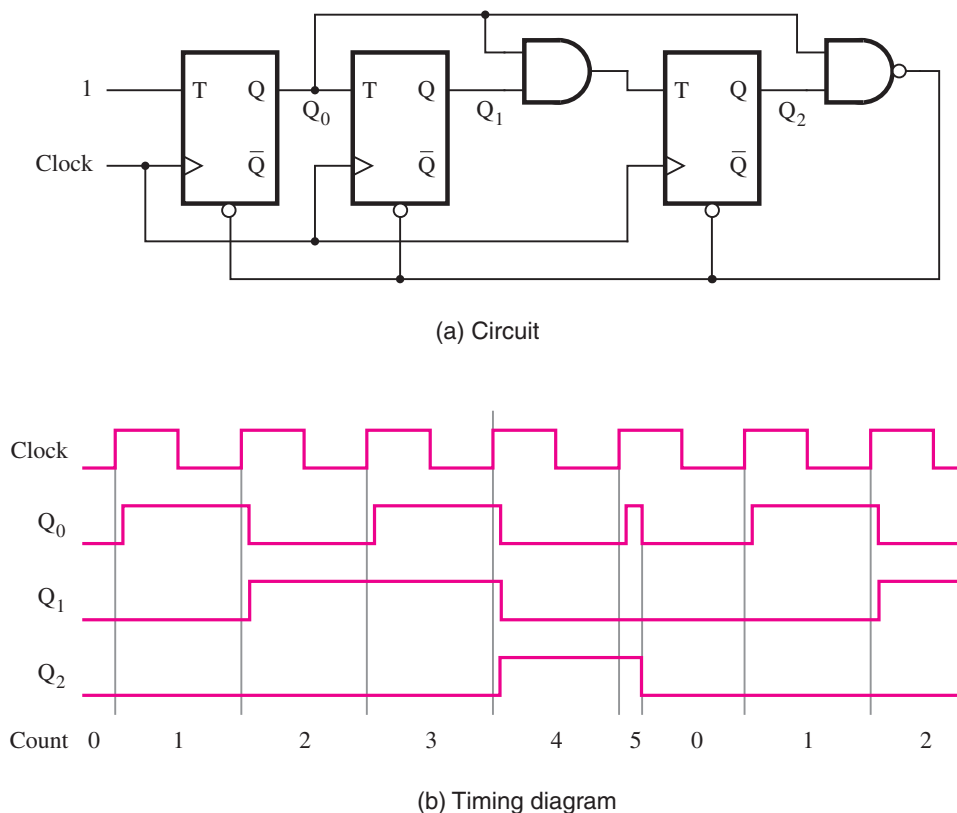
(b) Timing diagram

**Figure 5.25** A modulo-6 counter with synchronous reset.



uses a three-bit synchronous counter of the type depicted in Figure 5.24. The parallel-load feature of the counter is used to reset its contents when the count reaches 5. The resetting action takes place at the positive clock edge after the count has reached 5. It involves loading  $D_2D_1D_0 = 000$  into the flip-flops. As seen in the timing diagram in Figure 5.25b, the desired counting sequence is achieved, with each value of the count being established for one full clock cycle. Because the counter is reset on the active edge of the clock, we say that this type of counter has a *synchronous reset*.

Consider now the possibility of using the clear feature of individual flip-flops, rather than the parallel-load approach. The circuit in Figure 5.26a illustrates one possibility. It uses the counter structure of Figure 5.21a. Since the clear inputs are active when low, a NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops. Conceptually, this seems to work fine, but closer examination reveals a potential problem. The timing diagram for this circuit is given in Figure 5.26b. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the circuit, but not on the clock. Therefore, signal values  $Q_2Q_1Q_0 = 101$  are maintained for a



**Figure 5.26** A modulo-6 counter with asynchronous reset.

time that is much less than a clock cycle. Depending on a particular application of such a counter, this may be adequate, but it may also be completely unacceptable. For example, if the counter is used in a digital system where all operations in the system are synchronized by the same clock, then this narrow pulse denoting  $Count = 5$  would not be seen by the rest of the system. This is not a good way of designing circuits, because pulses of short length often cause unforeseen difficulties in practice. The approach employed in Figure 5.26a is said to use *asynchronous reset*.

The timing diagrams in Figures 5.25b and 5.26b suggest that synchronous reset is a better choice than asynchronous reset. The same observation is true if the natural counting sequence has to be broken by loading some value other than zero. The new value of the count can be established cleanly using the parallel-load feature. The alternative of using the clear and preset capability of individual flip-flops to set their states to reflect the desired count has the same problems as discussed in conjunction with the asynchronous reset.

---

## 5.11 OTHER TYPES OF COUNTERS

In this section we discuss three other types of counters that can be found in practical applications. The first uses the decimal counting sequence, and the other two generate sequences of codes that do not represent binary numbers.

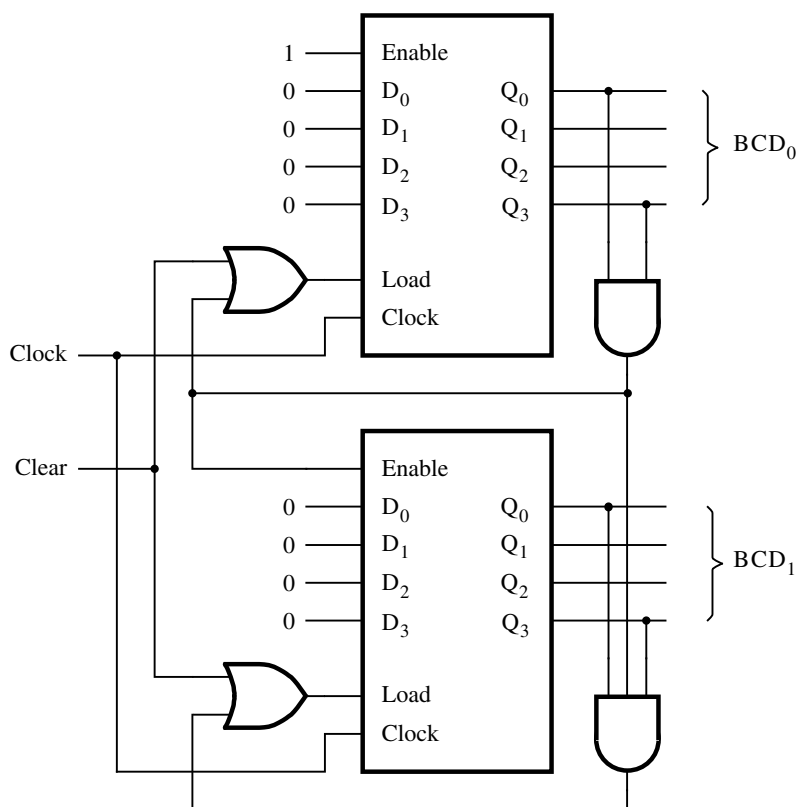
### 5.11.1 BCD COUNTER

Binary-coded-decimal (BCD) counters can be designed using the approach explained in Section 5.10. A two-digit BCD counter is presented in Figure 5.27. It consists of two modulo-10 counters, one for each BCD digit, which we implemented using the parallel-load four-bit counter of Figure 5.24. Note that in a modulo-10 counter it is necessary to reset the four flip-flops after the count of 9 has been obtained. Thus the *Load* input to each stage is equal to 1 when  $Q_3 = Q_0 = 1$ , which causes 0s to be loaded into the flip-flops at the next positive edge of the clock signal. Whenever the count in stage 0,  $BCD_0$ , reaches 9 it is necessary to enable the second stage so that it will be incremented when the next clock pulse arrives. This is accomplished by keeping the *Enable* signal for  $BCD_1$  low at all times except when  $BCD_0 = 9$ .

In practice, it has to be possible to clear the contents of the counter by activating some control signal. Two OR gates are included in the circuit for this purpose. The control input *Clear* can be used to load 0s into the counter. Observe that in this case *Clear* is active when high.

### 5.11.2 RING COUNTER

In the preceding counters the count is indicated by the state of the flip-flops in the counter. In all cases the count is a binary number. Using such counters, if an action is to be taken

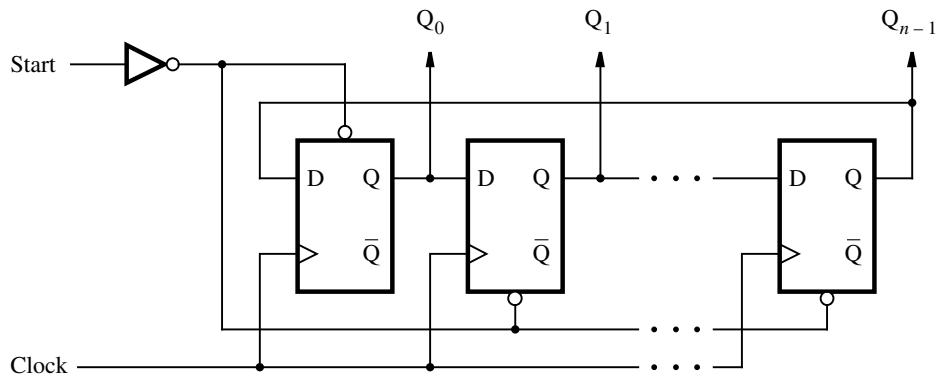
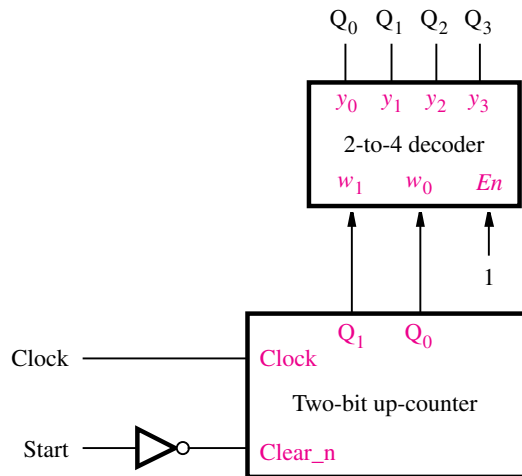


**Figure 5.27** A two-digit BCD counter.

as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 5.25 through 5.27.

It is possible to devise a counterlike circuit in which each flip-flop reaches the state  $Q_i = 1$  for exactly one count, while for all other counts  $Q_i = 0$ . Then  $Q_i$  indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flip-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 5.28a. The  $Q$  output of the last stage in the shift register is fed back as the input to the first stage, which creates a ringlike structure. If a single 1 is injected into the ring, this 1 will be shifted through the ring at successive clock cycles. For example, in a four-bit structure, the possible codes  $Q_0Q_1Q_2Q_3$  will be 1000, 0100, 0010, and 0001. As we said in Section 4.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.

The circuit in Figure 5.28a is referred to as a *ring counter*. Its operation has to be initialized by injecting a 1 into the first stage. This is achieved by using the *Start* control signal, which presets the left-most flip-flop to 1 and clears the others to 0. We assume that

(a) An  $n$ -bit ring counter

(b) A four-bit ring counter

**Figure 5.28** Ring counter.

all changes in the value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.

The circuit in Figure 5.28a can be used to build a ring counter with any number of bits,  $n$ . For the specific case of  $n = 4$ , part (b) of the figure shows how a ring counter can be constructed using a two-bit up-counter and a decoder. When *Start* is set to 1, the counter is reset to 00. After *Start* changes back to 0, the counter increments its value in the

normal way. The 2-to-4 decoder, described in Section 4.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces  $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$ , and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two.

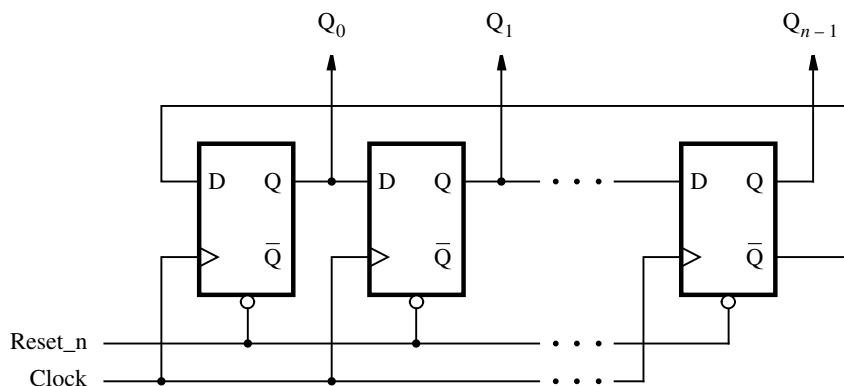
### 5.11.3 JOHNSON COUNTER

An interesting variation of the ring counter is obtained if, instead of the  $Q$  output, we take the  $\bar{Q}$  output of the last stage and feed it back to the first stage, as shown in Figure 5.29. This circuit is known as a *Johnson counter*. An  $n$ -bit counter of this type generates a counting sequence of length  $2n$ . For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on. Note that in this sequence, only a single bit has a different value for two consecutive codes.

To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops, as shown in the figure. Observe that neither the Johnson nor the ring counter will generate the desired counting sequence if not initialized properly.

### 5.11.4 REMARKS ON COUNTER DESIGN

The sequential circuits presented in this chapter, namely, registers and counters, have a regular structure that allows the circuits to be designed using an intuitive approach. In Chapter 6 we will present a more formal approach to design of sequential circuits and show how the circuits presented in this chapter can be derived using this approach.



**Figure 5.29** Johnson counter.

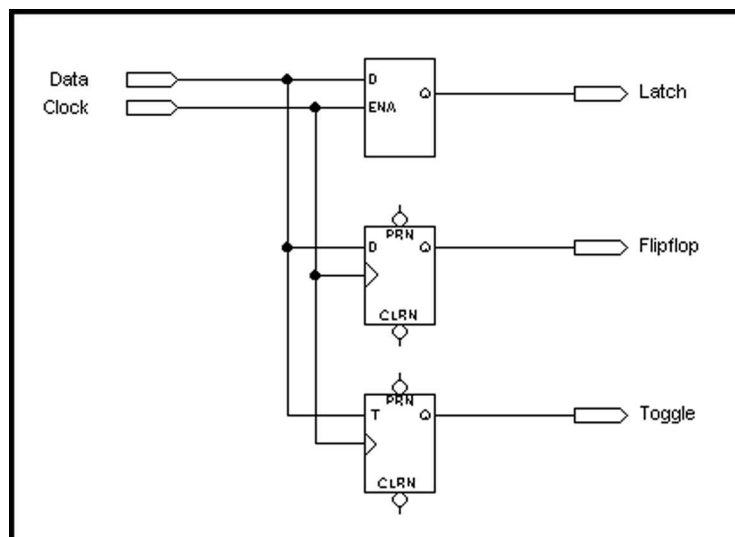
## 5.12 USING STORAGE ELEMENTS WITH CAD TOOLS

This section shows how circuits with storage elements can be designed using either schematic capture or Verilog code.

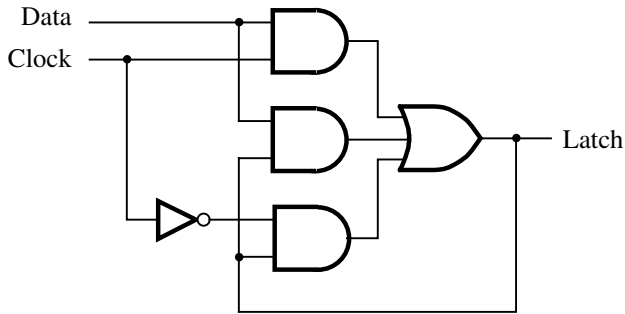
### 5.12.1 INCLUDING STORAGE ELEMENTS IN SCHEMATICS

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 5.30 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them.

When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates shown in Section 5.2. In some chips the AND-OR circuit depicted in Figure 5.31 may be preferable. This circuit is functionally equivalent to the cross-coupled version in Section 5.2. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs *Data* and *Latch*. Without this gate, the top



**Figure 5.30** Three types of storage elements in a schematic.



**Figure 5.31** Gated D latch generated by CAD tools.

AND gate sets the value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a *static hazard*. A detailed explanation of hazards will be given in Chapter 11.

The circuit in Figure 5.30 can be implemented in a CPLD as shown in Figure 5.32. The D and T flip-flops are realized using the flip-flops on the chip that are configurable as either D or T types. The figure depicts in blue the gates and wires needed to implement the circuit in Figure 5.30.

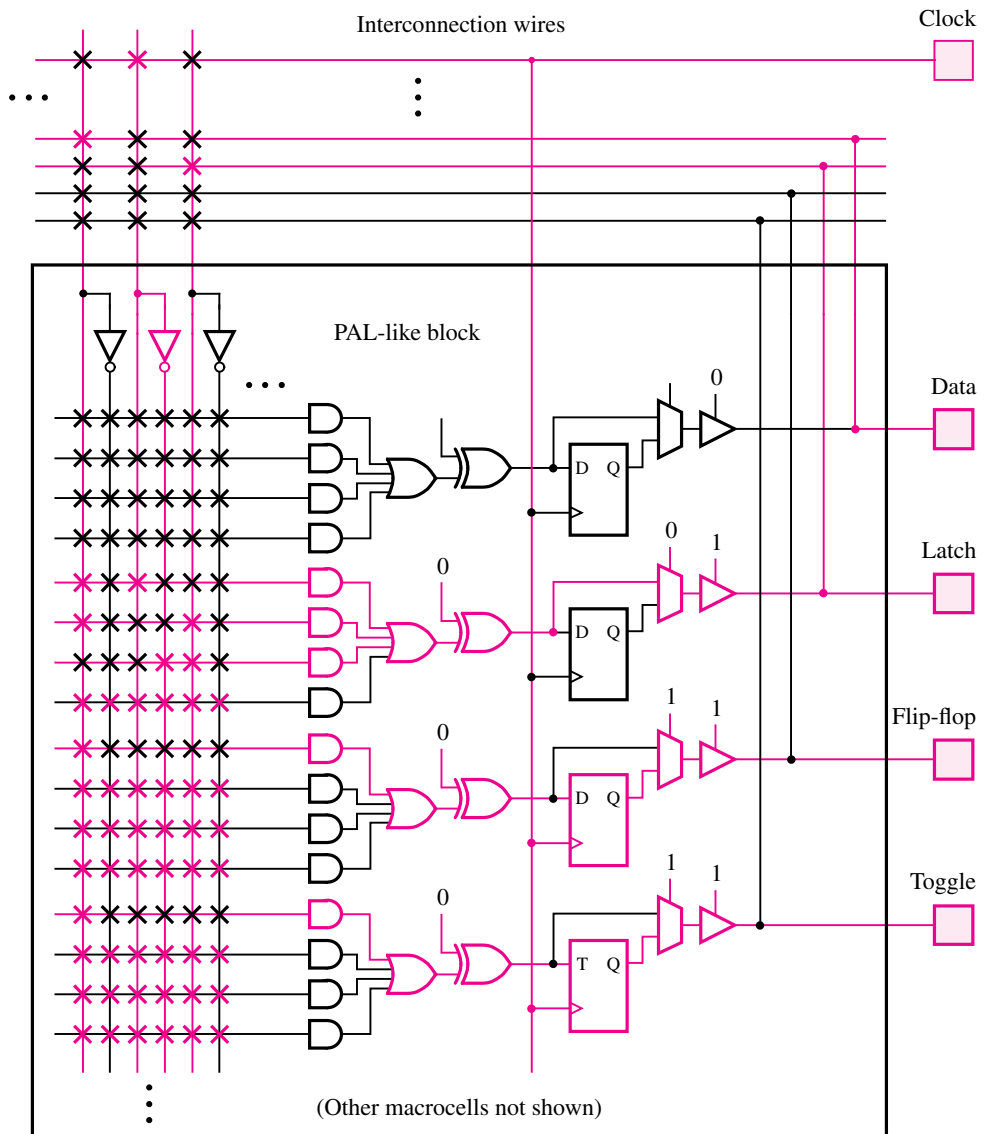
The results of a timing simulation for the implementation in Figure 5.32 are given in Figure 5.33. The *Latch* signal, which is the output of the gated D latch implemented as indicated in Figure 5.31, follows the *Data* input whenever the *Clock* signal is 1. Because of propagation delays in the chip, the *Latch* signal is delayed in time with respect to the *Data* signal. Since the *Flipflop* signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called *Toggle* in the figure, toggles when *Data* = 1 and a positive clock edge occurs. The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the *clock-to-output time*,  $t_{co}$ .

### 5.12.2 USING VERILOG CONSTRUCTS FOR STORAGE ELEMENTS

In Section 4.6 we described a number of Verilog constructs. We now show how these constructs can be used to describe storage elements.

A simple way of specifying a storage element is by using the **if-else** statement to describe the desired behavior responding to changes in the levels of data and clock inputs. Consider the **always** block

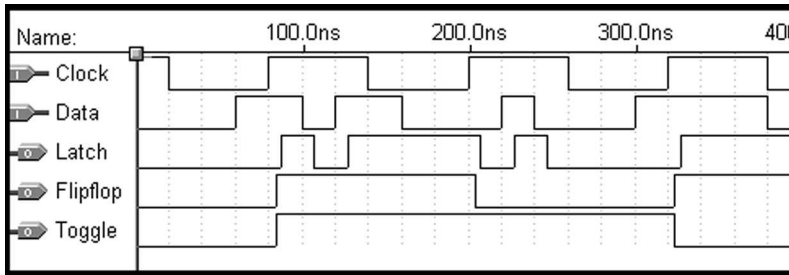
```
always @(Control, B)
if (Control)
    A = B;
```



**Figure 5.32** Implementation of the schematic in Figure 5.30 in a CPLD.

where  $A$  is a variable of **reg** type. This code specifies that the value of  $A$  should be made equal to the value of  $B$  when  $Control = 1$ . But the statement does not indicate an action that should occur when  $Control = 0$ . In the absence of an assigned value, the Verilog compiler assumes that the value of  $A$  caused by the **if** statement must be maintained when  $Control$  is not equal to 1. This notion of *implied memory* is realized by instantiating a latch in the circuit.





**Figure 5.33** Timing simulation for the storage elements in Figure 5.30.

```

module D_latch (D, Clk, Q);
    input D, Clk;
    output reg Q;

    always @(D, Clk)
        if (Clk)
            Q = D;

endmodule

```

**Figure 5.34** Code for a gated D latch.

**CODE FOR A GATED D LATCH** The code in Figure 5.34 defines a module named *D\_latch*, which has the inputs *D* and *Clk* and the output *Q*. The **if** clause defines that the *Q* output must take the value of *D* when *Clk* = 1. Since no **else** clause is given, a latch will be synthesized to maintain the value of *Q* when *Clk* = 0. Therefore, the code describes a gated D latch. The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the *Q* output.

### Example 5.1

An **always** construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list. While in the examples presented so far the **always** blocks are sensitive to the *levels* of signals, it is also possible to specify that a response should take place only at a particular edge of a signal. The desired edge is specified by using the Verilog keywords **posedge** and **negedge**, which are used to implement edge-triggered circuits.

**CODE FOR A D FLIP-FLOP** Figure 5.35 defines a module named *flipflop*, which is a positive-edge-triggered D flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the *Q* output. The keyword **posedge** specifies

### Example 5.2

```
module flipflop (D, Clock, Q);  
  input D, Clock;  
  output reg Q;  
  
  always @(posedge Clock)  
    Q = D;  
  
endmodule
```

**Figure 5.35** Code for a D flip-flop.

that a change may occur only on the positive edge of *Clock*. At this time the output *Q* is set to the value of the input *D*. Since **posedge** appears in the sensitivity list, *Q* will be implemented as the output of a flip-flop.

---

### 5.12.3 BLOCKING AND NON-BLOCKING ASSIGNMENTS

In all our Verilog examples presented so far we have used the equal sign for assignments, as in

$$f = x1 \& x2;$$

or

$$C = A + B;$$

or

$$Q = D;$$

This notation is called a *blocking* assignment. A Verilog compiler evaluates the statements in an **always** block in the order in which they are written. If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.

---

**Example 5.3** Consider the code in Figure 5.36. Since the **always** block is sensitive to the positive clock edge, both *Q1* and *Q2* will be implemented as the outputs of D flip-flops. However, because blocking assignments are involved, these two flip-flops will not be connected in cascade, as the reader might expect. The first statement

$$Q1 = D;$$

```

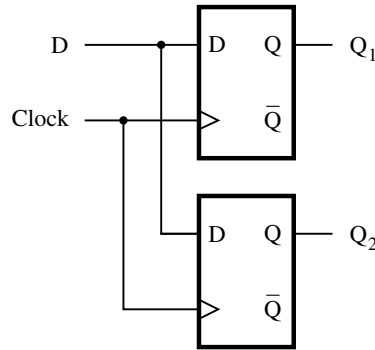
module example5_3 (D, Clock, Q1, Q2);
  input D, Clock;
  output reg Q1, Q2;

  always @(posedge Clock)
  begin
    Q1 = D;
    Q2 = Q1;
  end

endmodule

```

**Figure 5.36** Incorrect code for two cascaded flip-flops.



**Figure 5.37** Circuit for Example 5.3.

sets Q1 to the value of *D*. This new value is used in evaluating the subsequent statement

$$Q2 = Q1;$$

which results in  $Q2 = Q1 = D$ . The synthesized circuit has two parallel flip-flops, as illustrated in Figure 5.37. A synthesis tool will likely delete one of these redundant flip-flops as an optimization step.

Verilog also provides a *non-blocking* assignment, denoted with `<=`. All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered. Thus, a given variable has the same value for all statements in the block. The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.

---

**Example 5.4** Figure 5.38 gives the same code as in Figure 5.36, but using non-blocking assignments. In the two statements

```
Q1 <= D;
Q2 <= Q1;
```

the variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block. This code generates a cascaded connection between flip-flops, which implements the shift register depicted in Figure 5.39.

The differences between blocking and non-blocking assignments are illustrated further by the following two examples.

---

**Example 5.5** Code that involves some gates in addition to flip-flops is defined in Figure 5.40 using blocking assignment statements. The resulting circuit is given in Figure 5.41. Both *f* and *g* are implemented as the outputs of D flip-flops, because the sensitivity list of the **always** block specifies the event **posedge** Clock. Since blocking assignments are used, the updated value of *f* generated by the statement *f* = *x1* & *x2* has to be seen immediately by the following statement *g* = *f* | *x3*. Thus, the AND gate that produces *x1* & *x2* is connected to the OR gate that feeds the *g* flip-flop, as shown in Figure 5.41.

---

**Example 5.6** If non-blocking assignments are used, as given in Figure 5.42, then both *f* and *g* are updated simultaneously. Hence, the previous value of *f* is used in updating the value of *g*, which means that the output of the flip-flop that generates *f* is connected to the OR gate that feeds the *g* flip-flop. This gives rise to the circuit in Figure 5.43.

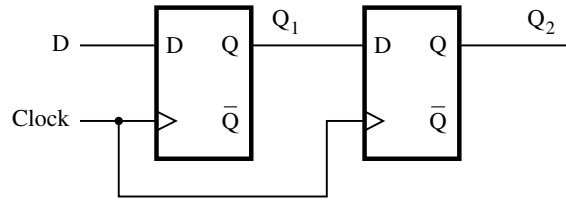
---

```
module example5_4 (D, Clock, Q1, Q2);
    input D, Clock;
    output reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 <= D;
        Q2 <= Q1;
    end

endmodule
```

**Figure 5.38** Code for two cascaded flip-flops.



**Figure 5.39** Circuit defined in Figure 5.38.

```

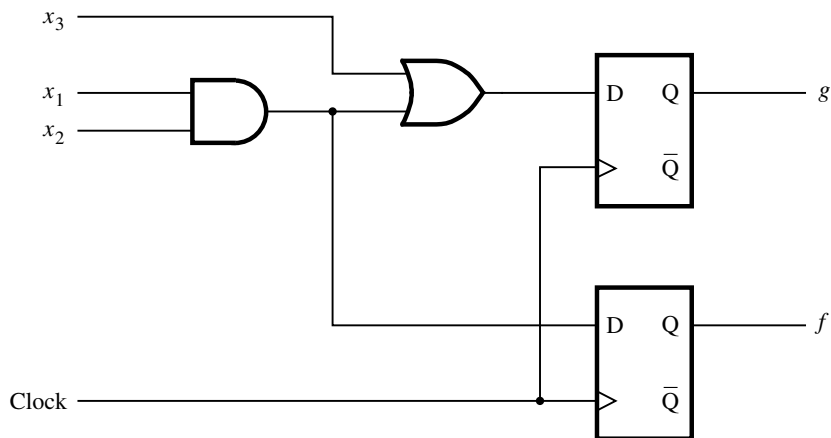
module example5_5 (x1, x2, x3, Clock, f, g);
  input x1, x2, x3, Clock;
  output reg f, g;

  always @(posedge Clock)
  begin
    f = x1 & x2;
    g = f | x3;
  end

endmodule

```

**Figure 5.40** Code for Example 5.5.



**Figure 5.41** Circuit for Example 5.5.

```

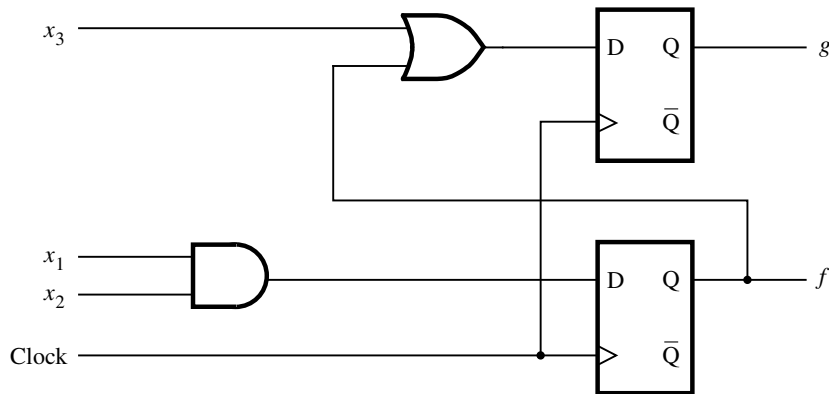
module example5_6 (x1, x2, x3, Clock, f, g);
  input x1, x2, x3, Clock;
  output reg f, g;

  always @(posedge Clock)
  begin
    f <= x1 & x2;
    g <= f | x3;
  end

endmodule

```

**Figure 5.42** Code for Example 5.6.



**Figure 5.43** Circuit for Example 5.6.

It is interesting to consider what circuit would be synthesized if the statements that specify  $f$  and  $g$  were reversed. For the code in Figure 5.40 the impact would be significant. If  $g$  is evaluated first, then the second statement does not depend on the first one, because  $f$  does not depend on  $g$ . The resulting circuit would be the same as the one in Figure 5.43. In contrast, reversing the statement order would make no difference for the code in Figure 5.42, in which the non-blocking assignment is used.

The use of blocking assignments for sequential circuits can easily lead to wrong results, as demonstrated in Figure 5.37. The dependence on ordering of blocking assignments is dangerous, as shown in the previous example. For this reason, only non-blocking assignments should be used to describe sequential circuits.

### 5.12.4 NON-BLOCKING ASSIGNMENTS FOR COMBINATIONAL CIRCUITS

A natural question at this point is whether non-blocking assignments can be used for combinational circuits. The answer is that they can be used in most situations, but when subsequent assignments in an **always** block depend on the results of previous assignments, the non-blocking assignments can generate nonsensical circuits. As an example, assume that we have a three-bit vector  $A = a_2a_1a_0$ , and we wish to generate a combinational function  $f$  that is equal to 1 when there are two adjacent bits in  $A$  that have the value 1. One way to specify this function with blocking assignments is

```
always @(A)
begin
    f = A[1] & A[0];
    f = f | (A[2] & A[1]);
end
```

These statements produce the desired logic function, which is  $f = a_1a_0 + a_2a_1$ . Consider now changing the code to use the non-blocking assignments

```
f <= A[1] & A[0];
f <= f | (A[2] & A[1]);
```

There are two key aspects of the Verilog semantics relevant to this code:

1. The results of non-blocking assignments are visible only after all of the statements in the **always** block have been evaluated.
2. When there are multiple assignments to the same variable inside an **always** block, the result of the last assignment is maintained.

In this example,  $f$  has an unspecified initial value when we enter the **always** block. The first statement assigns  $f = a_1a_0$ , but this result is not visible to the second statement. It still sees the original unspecified value of  $f$ . The second assignment overrides (deletes!) the first assignment and produces the logic function  $f = f + a_2a_1$ . This expression does not correspond to a combinational circuit, because it represents an AND-OR circuit in which the OR-gate is fed back to itself. It is best to use blocking assignments when describing combinational circuits, so as to avoid accidentally creating a sequential circuit.

### 5.12.5 FLIP-FLOPS WITH CLEAR CAPABILITY

By using a particular sensitivity list and a specific style of **if-else** statement, it is possible to include clear (or preset) signals on flip-flops.

---

**Example 5.7 ASYNCHRONOUS CLEAR** Figure 5.44 gives a module that defines a D flip-flop with an asynchronous active-low reset (clear) input. When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0. Note that the sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock. We cannot omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level-sensitive signals.

---

**Example 5.8 SYNCHRONOUS CLEAR** Figure 5.45 shows how a D flip-flop with a synchronous reset input can be described. In this case the reset signal is acted upon only when a positive clock edge arrives. This code generates the circuit in Figure 5.13c, which has an AND gate connected to the flip-flop's D input.

```

module flipflop (D, Clock, Resetn, Q);
  input D, Clock, Resetn;
  output reg Q;

  always @(negedge Resetn, posedge Clock)
    if (!Resetn)
      Q <= 0;
    else
      Q <= D;

endmodule

```

**Figure 5.44** D flip-flop with asynchronous reset.

```

module flipflop (D, Clock, Resetn, Q);
  input D, Clock, Resetn;
  output reg Q;

  always @(posedge Clock)
    if (!Resetn)
      Q <= 0;
    else
      Q <= D;

endmodule

```

**Figure 5.45** D flip-flop with synchronous reset.



## 5.13 USING VERILOG CONSTRUCTS FOR REGISTERS AND COUNTERS

In this section we show how registers and counters can be specified in Verilog code. Figure 5.44 gives code for a D flip-flop. One way to describe an  $n$ -bit register is to write hierarchical code that includes  $n$  instances of the D flip-flop subcircuit. A simpler approach is to use the same code as in Figure 5.44 and define the  $D$  input and  $Q$  output as multibit signals.

**AN  $N$ -BIT REGISTER** Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed. The code for an  $n$ -bit register is given in Figure 5.46. The parameter  $n$  specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

**Example 5.9**

**A FOUR-BIT SHIFT REGISTER** Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Figure 5.18. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the  $D$  input. Figure 5.47 defines the module named *muxdff*, which represents this subcircuit. The two data inputs are named  $D_0$  and  $D_1$ , and they are selected using the  $Sel$  input. The **if-else** statement specifies that on the positive clock edge if  $Sel = 0$ , then  $Q$  is assigned the value of  $D_0$ ; otherwise,  $Q$  is assigned the value of  $D_1$ . An alternative way of defining the same circuit is given in Figure 5.48. In this code, the conditional assignment statement specifies a 2-to-1 multiplexer with the output  $D$ , which is then connected to the flip-flop in the **always** block.

**Example 5.10**

Figure 5.49 defines the four-bit shift register. The module *Stage3* instantiates the left-most flip-flop, which has the output  $Q_3$ , and the module *Stage0* instantiates the right-most flip-flop,  $Q_0$ . When  $L = 1$ , the register is loaded in parallel from the  $R$  input; and when  $L = 0$ , shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit,  $Q_3$ , from the  $w$  input.

**ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER** A different style of code for the four-bit shift register is given in Figure 5.50. Instead of using subcircuits, the shift register is defined using the approach presented in Example 5.4. All actions take place at the positive edge of the clock. If  $L = 1$ , the register is loaded in parallel with the four bits of input  $R$ . If  $L = 0$ , the contents of the register are shifted to the right and the value of the input  $w$  is loaded into the most-significant bit  $Q_3$ .

**Example 5.11**

```

module regn (D, Clock, Resetn, Q);
  parameter n = 16;
  input [n-1:0] D;
  input Clock, Resetn;
  output reg [n-1:0] Q;

  always @(negedge Resetn, posedge Clock)
    if (!Resetn)
      Q <= 0;
    else
      Q <= D;

endmodule

```

**Figure 5.46** Code for an  $n$ -bit register with asynchronous clear.

```

module muxdff (D0, D1, Sel, Clock, Q);
  input D0, D1, Sel, Clock;
  output reg Q;

  always @(posedge Clock)
    if (!Sel)
      Q <= D0;
    else
      Q <= D1;

endmodule

```

**Figure 5.47** Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```

module muxdff (D0, D1, Sel, Clock, Q);
  input D0, D1, Sel, Clock;
  output reg Q;

  wire D;
  assign D = Sel ? D1 : D0;

  always @(posedge Clock)
    Q <= D;

endmodule

```

**Figure 5.48** Alternative code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```

module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output wire [3:0] Q;

    muxdff Stage3 (w, R[3], L, Clock, Q[3]);
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule

```

**Figure 5.49** Hierarchical code for a four-bit shift register.

```

module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output reg [3:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
            begin
                Q[0] <= Q[1];
                Q[1] <= Q[2];
                Q[2] <= Q[3];
                Q[3] <= w;
            end

endmodule

```

**Figure 5.50** Alternative code for a four-bit shift register.

---

**AN N-BIT SHIFT REGISTER** Figure 5.51 shows the code that can be used to represent shift registers of any size. The parameter  $n$ , which has the default value 16 in the figure, sets the number of flip-flops. The code is identical to that in Figure 5.50 with two exceptions. First,  $R$  and  $Q$  are defined in terms of  $n$ . Second, the **else** clause that describes the shifting operation is generalized to work for any number of flip-flops by using a **for** loop.

---

**Example 5.12**

```

module shiftn (R, L, w, Clock, Q);
  parameter n = 16;
  input [n-1:0] R;
  input L, w, Clock;
  output reg [n-1:0] Q;
  integer k;

  always @(posedge Clock)
    if (L)
      Q <= R;
    else
      begin
        for (k = 0; k < n-1; k = k+1)
          Q[k] <= Q[k+1];
        Q[n-1] <= w;
      end
endmodule

```

**Figure 5.51** An  $n$ -bit shift register.

```

module upcount (Resetn, Clock, E, Q);
  input Resetn, Clock, E;
  output reg [3:0] Q;

  always @(negedge Resetn, posedge Clock)
    if (!Resetn)
      Q <= 0;
    else if (E)
      Q <= Q + 1;
endmodule

```

**Figure 5.52** Code for a four-bit up-counter.

---

**Example 5.13 UP-COUNTER** Figure 5.52 represents a four-bit up-counter with a reset input, *Resetn*, and an enable input, *E*. The outputs of the flip-flops in the counter are represented by the vector named *Q*. The **if** statement specifies an asynchronous reset of the counter if *Resetn* = 0. The **else if** clause specifies that if *E* = 1 the count is incremented on the positive clock edge.

---

---

**UP-COUNTER WITH PARALLEL LOAD** The code in Figure 5.53 defines an up-counter that has a parallel-load input in addition to a reset input. The parallel data is provided as the input vector *R*. The first **if** statement provides the same asynchronous reset as in Figure 5.52. The **else if** clause specifies that if  $L = 1$  the flip-flops in the counter are loaded in parallel from the *R* inputs on the positive clock edge. If  $L = 0$ , the count is incremented, under control of the enable input *E*. **Example 5.14**

---

**DOWN-COUNTER WITH PARALLEL LOAD** Figure 5.54 shows the code for a down-counter named *downcount*. A down-counter is normally used by loading it with some starting count and then decrementing its contents. The starting count is represented in the code by the vector *R*. On the positive clock edge, if  $L = 1$  the counter is loaded with the input *R*, and if  $L = 0$  the count is decremented, under control of the enable input *E*. **Example 5.15**

---

**UP/DOWN COUNTER** Verilog code for an up/down counter is given in Figure 5.55. This module combines the capabilities of the counters defined in Figures 5.53 and 5.54. It includes a control signal *up\_down* that governs the direction of counting. **Example 5.16**

---

```

module upcount (R, Resetn, Clock, E, L, Q);
  input [3:0] R;
  input Resetn, Clock, E, L;
  output reg [3:0] Q;

  always @(negedge Resetn, posedge Clock)
    if (!Resetn)
      Q <= 0;
    else if (L)
      Q <= R;
    else if (E)
      Q <= Q + 1;

endmodule

```

**Figure 5.53** A four-bit up-counter with a parallel load.

```

module downcount (R, Clock, E, L, Q);
    parameter n = 8;
    input [n-1:0] R;
    input Clock, L, E;
    output reg [n-1:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else if (E)
            Q <= Q - 1;

endmodule

```

**Figure 5.54** A down-counter with a parallel load.

```

module updowncount (R, Clock, L, E, up_down, Q);
    parameter n = 8;
    input [n-1:0] R;
    input Clock, L, E, up_down;
    output reg [n-1:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else if (E)
            Q <= Q + (up_down ? 1 : -1);

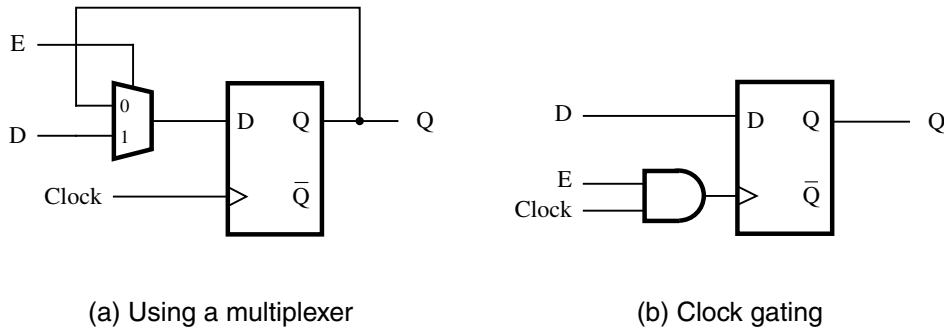
endmodule

```

**Figure 5.55** Code for an up/down counter.

### 5.13.1 FLIP-FLOPS AND REGISTERS WITH ENABLE INPUTS

We showed in Figures 5.22 and 5.23 how an enable input can be used in counter circuits to be able to prevent the flip-flops from toggling when an active clock edge occurs. It is also useful in many other types of circuits to be able to prevent the data stored in flip-flops from changing when an active clock edge occurs. For D flip-flops, this capability can be provided by adding a multiplexer to the flip-flop, as shown in Figure 5.56a. When  $E = 0$ , the flip-flop output cannot change, because the multiplexer connects  $Q$  to the input of the



**Figure 5.56** Providing an enable input for a D flip-flop.

```

module rege (D, Clock, Resetn, E, Q);
  input D, Clock, Resetn, E;
  output reg Q;

  always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
      Q <= 0;
    else if (E)
      Q <= D;

endmodule

```

**Figure 5.57** Code for a D flip-flop with enable.

flip-flop. But if  $E = 1$ , then the multiplexer allows new data to be loaded into the flip-flop from the  $D$  input. Instead of using the multiplexer shown in the figure, another way to implement the enable feature is to use a two-input AND gate as illustrated in Figure 5.56b. Then setting  $E = 0$  prevents the clock signal from reaching the flip-flop's clock input. This method seems simpler than the multiplexer approach, but we will show in Section 5.15 that it can cause problems in practical operation. We will prefer the multiplexer-based approach over gating the clock with an AND gate.

Verilog code for a D flip-flop with an asynchronous reset input and an enable input is given in Figure 5.57. After first specifying the reset condition, the **always** block uses an **else if** clause to ensure that the data stored in the flip-flop can change only when  $E = 1$ . We can extend the enable capability to registers with  $n$  bits by using  $n$  2-to-1 multiplexers controlled by  $E$ , as shown in Figure 5.58. The multiplexer for each flip-flop,  $i$ , selects either the external data bit,  $R_i$ , or the flip-flop's output,  $Q_i$ .

```

module regne (R, Clock, Resetn, E, Q);
  parameter n = 8;
  input [n-1:0] R;
  input Clock, Resetn, E;
  output reg [n-1:0] Q;

  always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
      Q <= 0;
    else if (E)
      Q <= R;

endmodule

```

**Figure 5.58** An  $n$ -bit register with an enable input.

### 5.13.2 SHIFT REGISTERS WITH ENABLE INPUTS

It is useful to be able to inhibit the shifting operation in a shift register by using an enable input,  $E$ . We showed in Figure 5.18 that shift registers can be constructed with a parallel-load capability, which is implemented using a multiplexer. Figure 5.59 shows how the enable feature can be added by using an additional multiplexer. If the parallel-load control input,  $L$ , is 1, the flip-flops are loaded in parallel. If  $L = 0$ , the additional multiplexer selects new data to be loaded into the flip-flops only if the enable  $E$  is 1.

Verilog code that represents the circuit in Figure 5.59 is given in Figure 5.60. When  $L = 1$ , the register is loaded in parallel from the  $R$  input. When  $L = 0$  and  $E = 1$ , the data in the shift register is shifted in a left-to-right direction.

---

## 5.14 DESIGN EXAMPLE

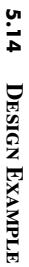
This section presents an example of a digital system that makes use of some of the building blocks described in this chapter and in Chapter 4.

### 5.14.1 REACTION TIMER

Electronic devices operate at remarkably fast speeds, with the typical delay through a logic gate being less than 1 ns. In this example we use a logic circuit to measure the speed of a much slower type of device—a person.

We will design a circuit that can be used to measure the reaction time of a person to a specific event. The circuit turns on a small light, called a *light-emitting diode (LED)*. In response to the LED being turned on, the person attempts to press a switch as quickly as





303

```

module shiftrne (R, L, E, w, Clock, Q);
  parameter n = 4;
  input [n-1:0] R;
  input L, E, w, Clock;
  output reg [n-1:0] Q;
  integer k;

  always @(posedge Clock)
  begin
    if (L)
      Q <= R;
    else if (E)
      begin
        Q[n-1] <= w;
        for (k = n-2; k >= 0; k = k-1)
          Q[k] <= Q[k+1];
      end
    end
  end

endmodule

```

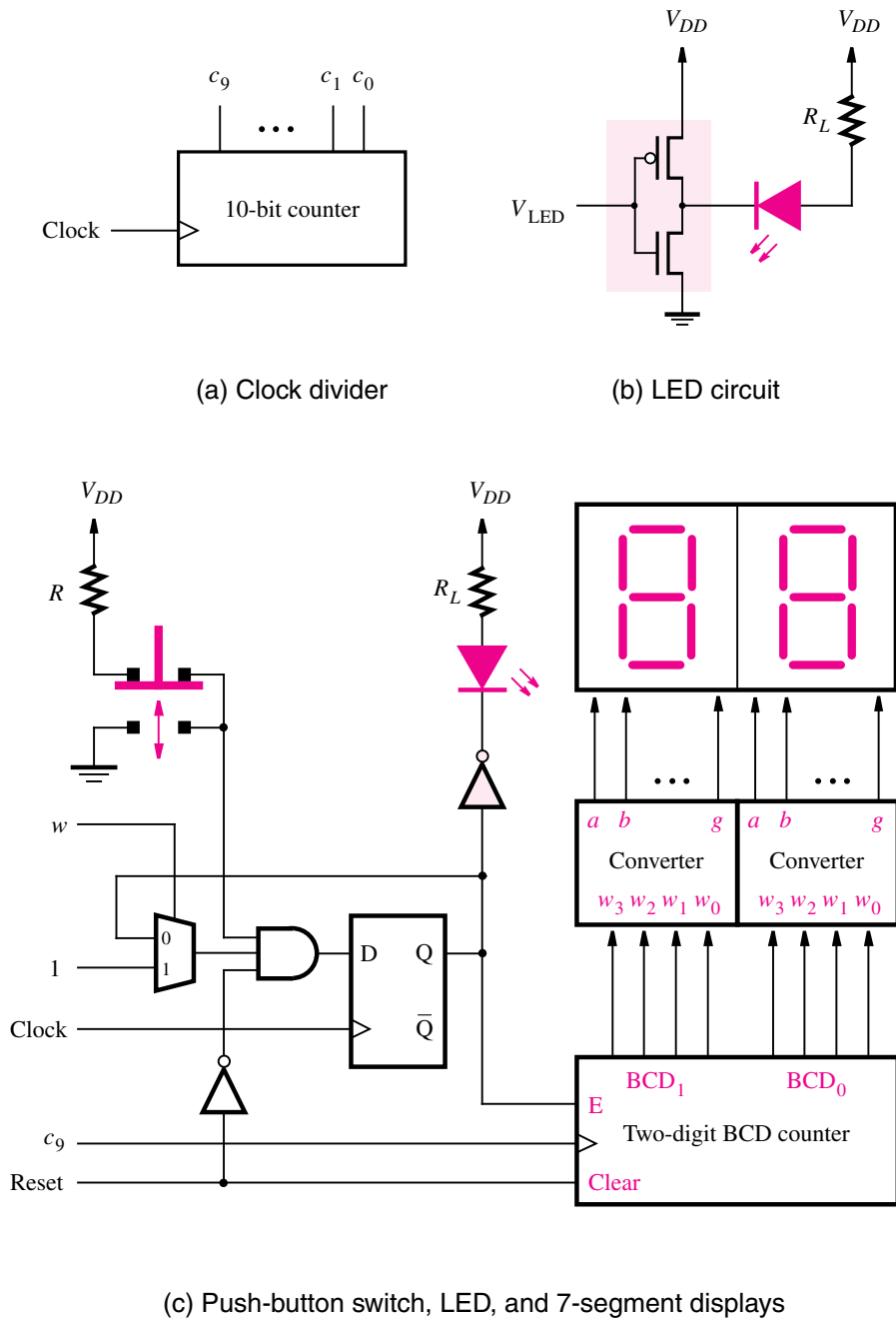
**Figure 5.60** A left-to-right shift register with an enable input.

possible. The circuit measures the elapsed time from when the LED is turned on until the switch is pressed.

To measure the reaction time, a clock signal with an appropriate frequency is needed. In this example we use a 100 Hz clock, which measures time at a resolution of 1/100 of a second. The reaction time can then be displayed using two digits that represent fractions of a second from 00/100 to 99/100.

Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a four-bit counter is given in Figure 5.21. It shows that the least-significant bit output,  $Q_0$ , of the counter is a periodic signal with half the frequency of the clock input. Hence we can view  $Q_0$  as dividing the clock frequency by two. Similarly, the  $Q_1$  output divides the clock frequency by four. In general, output  $Q_i$  in an  $n$ -bit counter divides the clock frequency by  $2^{i+1}$ . In the case of our 102.4 kHz clock signal, we can use a 10-bit counter, as shown in Figure 5.61b. The counter output  $c_9$  has the required 100 Hz frequency because  $102400 \text{ Hz}/1024 = 100 \text{ Hz}$ .

The reaction timer circuit has to be able to turn an LED on and off. The graphical symbol for an LED is shown in blue in Figure 5.61b. Small blue arrows in the symbol represent the light that is emitted when the LED is turned on. The LED has two terminals: the one on the left in the figure is the *cathode*, and the terminal on the right is the *anode*.



**Figure 5.61** A reaction-timer circuit.

To turn the LED on, the cathode has to be set to a sufficiently lower voltage than the anode, which causes a current to flow through the LED. If the voltages on its two terminals are equal, the LED is off.

Figure 5.61*b* shows one way to control the LED, using an inverter. If the input voltage  $V_{LED} = 0$ , then the voltage at the cathode is equal to  $V_{DD}$ ; hence the LED is off. But if  $V_{LED} = V_{DD}$ , the cathode voltage is 0 V and the LED is on. The amount of current that flows is limited by the value of the resistor  $R_L$ . This current flows through the LED and the inverter. Since the current flows *into* the inverter, we say that the inverter *sinks* the current. The maximum current that a logic gate can sink without sustaining permanent damage is usually called  $I_{OL}$ , which stands for the “maximum current when the output is low.” The value of  $R_L$  is chosen such that the current is less than  $I_{OL}$ . As an example assume that the inverter is implemented inside a chip for which the value of  $I_{OL}$  specified in the data sheet for the chip is 12 mA. For  $V_{DD} = 5$  V, this leads to  $R_L \approx 450 \, \Omega$  because  $5 \text{ V} / 450 \, \Omega = 11 \text{ mA}$  (there is actually a small voltage drop across the LED when it is turned on, but we ignore this for simplicity). The amount of light emitted by the LED is proportional to the current flow. If 11 mA is insufficient, then the inverter should be implemented in a driver chip, like those described in Appendix B, because drivers provide a higher value of  $I_{OL}$ .

The complete reaction-timer circuit is illustrated in Figure 5.61*c*, with the inverter from part (*b*) shaded in grey. The graphical symbol for a push-button switch is shown in the top left of the diagram. The switch normally makes contact with the top terminals, as depicted in the figure. When depressed, the switch makes contact with the bottom terminals; when released, it automatically springs back to the top position. In the figure the switch is connected such that it normally produces a logic value of 1, and it produces a 0 pulse when pressed.

When depressed, the push-button switch causes the D flip-flop to be synchronously reset. The output of this flip-flop determines whether the LED is on or off, and it also provides the count enable input to a two-digit BCD counter. As discussed in Section 5.11, each digit in a BCD counter has four bits that take the values 0000 to 1001. Thus the counting sequence can be viewed as decimal numbers from 00 to 99. A circuit for the BCD counter is given in Figure 5.27. Each digit in the counter is connected through a code converter to a 7-segment display, which we described in the discussion for Figure 2.63. In Figure 5.61*c* the counter is clocked by the  $c_9$  output of the clock divider in part (*a*) of the figure. The intended use of the reaction-timer circuit is to first assert the *Reset* input to clear the flip-flop and thus turn off the LED and disable the counter. Asserting the *Reset* input also clears the contents of the counter to 00. The input  $w$  normally has the value 0, which keeps the flip-flop cleared and prevents the count value from changing. The reaction test is initiated by setting  $w = 1$  for one  $c_9$  clock cycle. After the next positive edge of the clock, the flip-flop output becomes a 1, which turns on the LED. We assume that  $w$  returns to 0 after one  $c_9$  clock cycle, but the flip-flop output remains at 1 because of the 2-to-1 multiplexer connected to the  $D$  input. The counter is then incremented every 1/100 of a second. When the user depresses the switch, the flip-flop is cleared, which turns off the LED and stops the counter. The two-digit display shows the elapsed time to the nearest 1/100 of a second from when the LED was turned on until the user was able to respond by depressing the switch.

### Verilog Code

To describe the circuit in Figure 5.61c using Verilog code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. Code for the BCD counter, which represents the circuit in Figure 5.27, is shown in Figure 5.62. The two-digit BCD output is represented by the 2 four-bit signals *BCD1* and *BCD0*. The *Clear* input provides a synchronous reset for both digits in the counter. If  $E = 1$ , the count value is incremented on the positive clock edge; and if  $E = 0$ , the count value is unchanged. Each digit can take the values from 0000 to 1001. Figure 5.63 gives the code for the BCD-to-7-segment decoder.

Figure 5.64 shows the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit1* and *Digit0*.

The flip-flop in Figure 5.61c is loaded with the value 1 if  $w = 1$ , but if  $w = 0$  the stored value in the flip-flop is not changed. This circuit is described by the **always** block in Figure 5.64, which also includes a synchronous reset input; the reset is activated if either

```

module BCDcount (Clock, Clear, E, BCD1, BCD0);
  input Clock, Clear, E;
  output reg [3:0] BCD1, BCD0;

  always @(posedge Clock)
  begin
    if (Clear)
    begin
      BCD1 <= 0;
      BCD0 <= 0;
    end
    else if (E)
      if (BCD0 == 4'b1001)
      begin
        BCD0 <= 0;
        if (BCD1 == 4'b1001)
          BCD1 <= 0;
        else
          BCD1 <= BCD1 + 1;
        end
      else
        BCD0 <= BCD0 + 1;
      end
    end

endmodule

```

**Figure 5.62** Code for the two-digit BCD counter in Figure 5.27.

```

module seg7 (bcd, leds);
  input [3:0] bcd;
  output reg [1:7] leds;

  always @(bcd)
    case (bcd) //abcdefg
      0: leds = 7'b1111110;
      1: leds = 7'b0110000;
      2: leds = 7'b1101101;
      3: leds = 7'b1111001;
      4: leds = 7'b0110011;
      5: leds = 7'b1011011;
      6: leds = 7'b1011111;
      7: leds = 7'b1110000;
      8: leds = 7'b1111111;
      9: leds = 7'b1111011;
      default: leds = 7'bx;
    endcase

endmodule

```

**Figure 5.63** Code for the BCD-to-7-segment decoder.

```

module reaction (Clock, Reset, c9, w, Pushn, LEDn, Digit1, Digit0);
  input Clock, Reset, c9, w, Pushn;
  output wire LEDn;
  output wire [1:7] Digit1, Digit0;
  reg LED;
  wire [3:0] BCD1, BCD0;

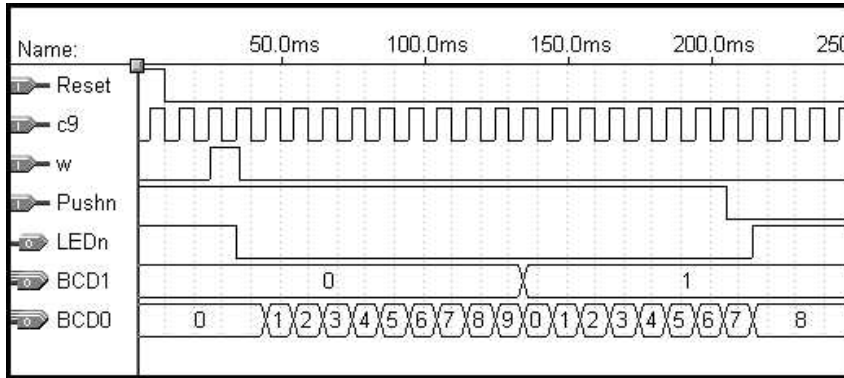
  always @(posedge Clock)
    begin
      if (!Pushn || Reset)
        LED <= 0;
      else if (w)
        LED <= 1;
    end

  assign LEDn = ~LED;
  BCDcount counter (c9, Reset, LED, BCD1, BCD0);
  seg7 seg1 (BCD1, Digit1);
  seg7 seg0 (BCD0, Digit0);

endmodule

```

**Figure 5.64** Code for the reaction timer.



**Figure 5.65** Simulation of the reaction timer circuit.

$Pushn = 0$  or  $Reset = 1$ . We have chosen to use a synchronous reset because the flip-flop output is connected to the enable input  $E$  on the BCD counter. As we know from the discussion in Section 5.3, it is important that all signals connected to flip-flops meet the required setup and hold times. The push-button switch can be pressed at any time and is not synchronized to the  $c_9$  clock signal. By using a synchronous reset for the flip-flop in Figure 5.61c, we avoid possible timing problems in the counter. Of course, the setup time of the flip-flop itself may become violated due to the asynchronous operation of the push-button switch. We show in Chapter 7 how this type of problem can be alleviated by adding extra flip-flops that are used to synchronize signals.

A simulation of the reaction-timer circuit implemented in a chip is shown in Figure 5.65. Initially,  $Reset$  is asserted to clear the the flip-flop and counter. When  $w$  changes to 1, the circuit sets  $LEDn$  to 0, which represents the LED being turned on. After some amount of time, the switch will be depressed. In the simulation we arbitrarily set  $Pushn$  to 0 after 18  $c_9$  clock cycles. Thus this choice represents the case when the person's reaction time is about 0.18 seconds. In human terms this duration is a very short time; for electronic circuits it is a very long time. An inexpensive personal computer can perform tens of millions of operations in 0.18 seconds!

### 5.14.2 REGISTER TRANSFER LEVEL (RTL) CODE

At this point, we have introduced most of the Verilog constructs that are needed for synthesis. Most of our examples give behavioral code, utilizing **if-else** statements, **case** statements, **for** loops, and other procedural statements. It is possible to write behavioral code in a style that resembles a computer program, in which there is a complex flow of control with many loops and branches. With such code, sometimes called *high-level* behavioral code, it is difficult to relate the code to the final hardware implementation; it may even be difficult to predict what circuit a high-level synthesis tool will produce. In this book we do not use the high-level style of code. Instead, we present Verilog code in such a way that the code can be easily related to the circuit that is being described. Most design modules presented are fairly small, to facilitate simple descriptions. Larger designs are built by interconnecting the smaller

modules. This approach is usually referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. RTL code is characterized by a straightforward flow of control through the code; it comprises well-understood subcircuits that are connected together in a simple way.

## 5.15 TIMING ANALYSIS OF FLIP-FLOP CIRCUITS

In Figure 5.14 we showed the timing parameters associated with a D flip-flop. A simple circuit that uses this flip-flop is given in Figure 5.66. We wish to calculate the maximum clock frequency,  $F_{max}$ , for which this circuit will operate properly, and also determine if the circuit suffers from any hold time violations. In the literature, this type of analysis of circuits is usually called *timing analysis*. We will assume that the flip-flop timing parameters have the values  $t_{su} = 0.6$  ns,  $t_h = 0.4$  ns, and  $0.8 \text{ ns} \leq t_{cQ} \leq 1.0$  ns. A range of minimum and maximum values is given for  $t_{cQ}$  because, as we mentioned in Section 5.4.4, this is the usual way of dealing with variations in delay that exist in integrated circuit chips.

To calculate the minimum period of the clock signal,  $T_{min} = 1/F_{max}$ , we need to consider all paths in the circuit that start and end at flip-flops. In this simple circuit there is only one such path, which starts when data is loaded into the flip-flop by a positive clock edge, propagates to the Q output after the  $t_{cQ}$  delay, propagates through the NOT gate, and finally must meet the setup requirement at the  $D$  input. Therefore

$$T_{min} = t_{cQ} + t_{NOT} + t_{su}$$

Since we are interested in the longest delay for this calculation, the maximum value of  $t_{cQ}$  should be used. For the calculation of  $t_{NOT}$  we will assume that the delay through any logic gate can be calculated as  $1 + 0.1k$ , where  $k$  is the number of inputs to the gate. For a NOT gate this gives 1.1 ns, which leads to

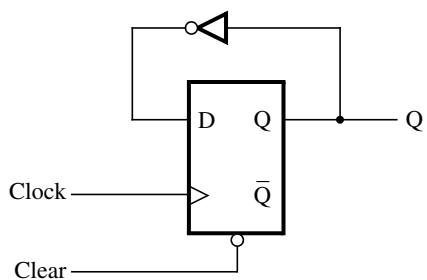
$$\begin{aligned} T_{min} &= 1.0 + 1.1 + 0.6 = 2.7 \text{ ns} \\ F_{max} &= 1/2.7 \text{ ns} = 370.37 \text{ MHz} \end{aligned}$$

It is also necessary to check if there are any hold time violations in the circuit. In this case we need to examine the shortest possible delay from a positive clock edge to a change in the value of the  $D$  input. The delay is given by  $t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9$  ns. Since  $1.9 \text{ ns} > t_h = 0.4$  ns there is no hold time violation.

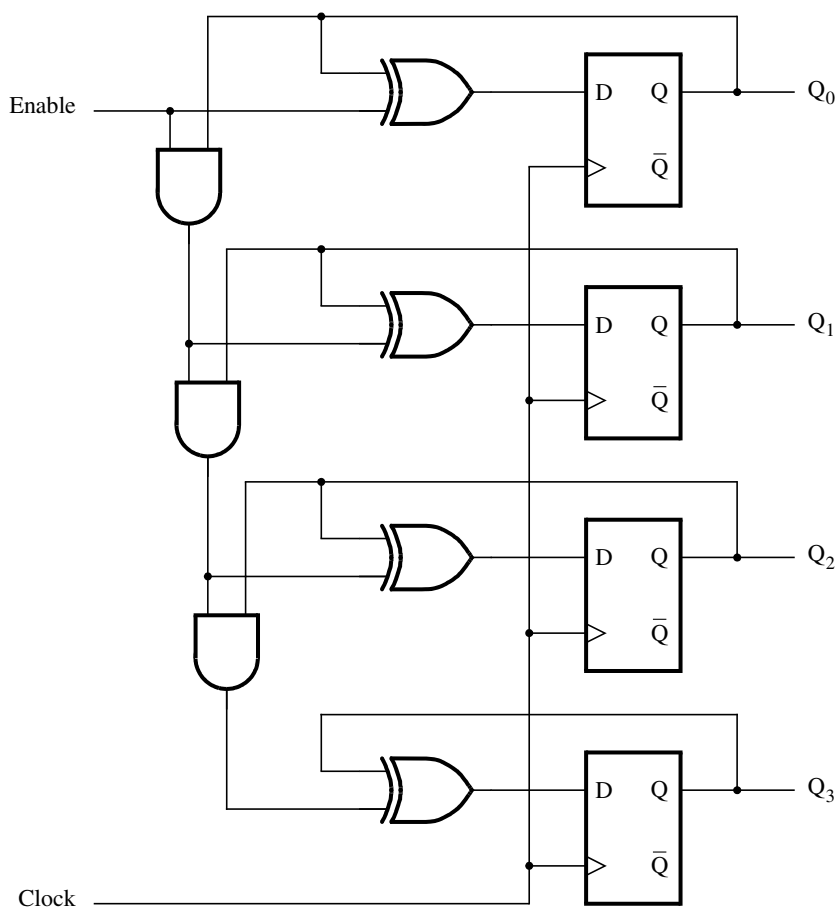
As another example of timing analysis of flip-flop circuits, consider the counter circuit shown in Figure 5.67. We wish to calculate the maximum clock frequency for which this circuit will operate properly assuming the same flip-flop timing parameters as we did for Figure 5.66. We will again assume that the propagation delay through a logic gate can be calculated as  $1 + 0.1k$ .

There are many paths in this circuit that start and end at flip-flops. The longest such path starts at flip-flop  $Q_0$  and ends at flip-flop  $Q_3$ . The longest path in a circuit is often called a *critical* path. The delay of the critical path includes the clock-to-Q delay of flip-flop  $Q_0$ , the propagation delay through three AND gates, and one XOR-gate delay. We must also





**Figure 5.66** A simple flip-flop circuit.



**Figure 5.67** A 4-bit counter.

account for the setup time of flip-flop Q<sub>3</sub>. This gives

$$T_{min} = t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su}$$

Using the maximum value of  $t_{cQ}$  gives

$$\begin{aligned} T_{min} &= 1.0 + 3(1.2) + 1.2 + 0.6 \text{ ns} = 6.4 \text{ ns} \\ F_{max} &= 1/6.4 \text{ ns} = 156.25 \text{ MHz} \end{aligned}$$

The shortest paths through the circuit are from each flip-flop to itself, through an XOR gate. The minimum delay along each such path is  $t_{cQ} + t_{XOR} = 0.8 + 1.2 = 2.0 \text{ ns}$ . Since  $2.0 \text{ ns} > t_h = 0.4 \text{ ns}$  there are no hold time violations.

### 5.15.1 TIMING ANALYSIS WITH CLOCK SKEW

In the above analysis we assumed that the clock signal arrived at exactly the same time at all four flip-flops. We will now repeat this analysis assuming that the clock signal still arrives at flip-flops Q<sub>0</sub>, Q<sub>1</sub>, and Q<sub>2</sub> simultaneously, but that there is a delay in the arrival of the clock signal at flip-flop Q<sub>3</sub>. Such a variation in the arrival time of a clock signal at different flip-flops is called *clock skew*,  $t_{skew}$ , and can be caused by a number of factors.

In Figure 5.67 the critical path through the circuit is from flip-flop Q<sub>0</sub> to Q<sub>3</sub>. However, the clock skew at Q<sub>3</sub> has the effect of reducing this delay, because it provides additional time before data is loaded into this flip-flop. Taking a clock skew of 1.5 ns into account, the delay of the path from flip-flop Q<sub>0</sub> to Q<sub>3</sub> is given by  $t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su} - t_{skew} = 6.4 - 1.5 \text{ ns} = 4.9 \text{ ns}$ . There is now a different critical path through the circuit, which starts at flip-flop Q<sub>0</sub> and ends at Q<sub>2</sub>. The delay of this path gives

$$\begin{aligned} T_{min} &= t_{cQ} + 2(t_{AND}) + t_{XOR} + t_{su} \\ &= 1.0 + 2(1.2) + 1.2 + 0.6 \text{ ns} \\ &= 5.2 \text{ ns} \\ F_{max} &= 1/5.2 \text{ ns} = 192.31 \text{ MHz} \end{aligned}$$

In this case the clock skew results in an increase in the circuit's maximum clock frequency. But if the clock skew had been negative, which would be the case if the clock signal arrived earlier at flip-flop Q<sub>3</sub> than at other flip-flops, then the result would have been a reduced  $F_{max}$ .

Since the loading of data into flip-flop Q<sub>3</sub> is delayed by the clock skew, it has the effect of increasing the hold time requirement of this flip-flop to  $t_h + t_{skew}$ , for all paths that end at Q<sub>3</sub> but start at Q<sub>0</sub>, Q<sub>1</sub>, or Q<sub>2</sub>. The shortest such path in the circuit is from flip-flop Q<sub>2</sub> to Q<sub>3</sub> and has the delay  $t_{cQ} + t_{AND} + t_{XOR} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns}$ . Since  $3.2 \text{ ns} > t_h + t_{skew} = 1.9 \text{ ns}$  there is no hold time violation.

If we repeat the above hold time analysis for clock skew values  $t_{skew} \geq 3.2 - t_h = 2.8 \text{ ns}$ , then hold time violations will exist. Thus, if  $t_{skew} \geq 2.8 \text{ ns}$  the circuit will not work reliably at any clock frequency.

Consider the circuit in Figure 5.68. In this circuit, there is a path that starts at flip-flop  $Q_1$ , passes through some network of logic gates, and ends at the  $D$  input of flip-flop  $Q_2$ . As indicated in the figure, different delays may be incurred before the clock signal reaches the flip-flops. Let  $\Delta_1$  and  $\Delta_2$  be the clock-signal delays for flip-flops  $Q_1$  and  $Q_2$ , respectively. The clock skew between these two flip-flops is then defined as **Example 5.17**

$$t_{skew} = \Delta_2 - \Delta_1$$

Let the longest delay along the paths through the logic gates in the circuit be  $t_L$ . Then, the minimum allowable clock period for these two flip-flops is

$$T_{min} = t_{cQ} + t_L + t_{su} - t_{skew}$$

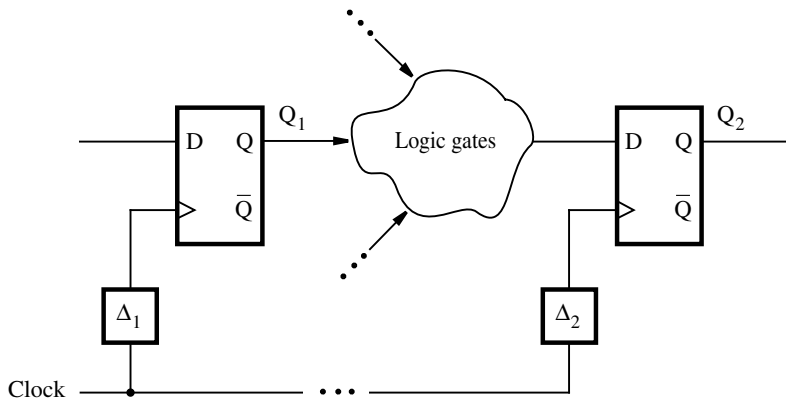
Thus, if  $\Delta_2 > \Delta_1$ , then  $t_{skew}$  allows for an increased value of  $F_{max}$ , but if  $\Delta_2 < \Delta_1$ , then the clock skew requires a decrease in  $F_{max}$ .

To calculate whether a hold time violation exists at flip-flop  $Q_2$  we need to determine the delay along the shortest path between the flip-flops. If the minimum delay through the logic gates in the circuit is  $t_l$ , then a hold time violation occurs if

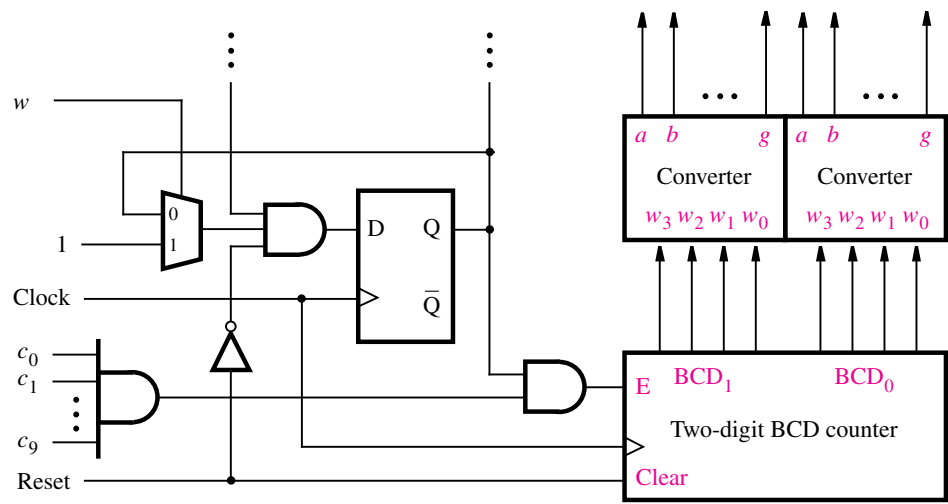
$$t_{cQ} + t_l < t_h + t_{skew}$$

Here, the hold-time constraint is more difficult to meet if  $\Delta_2 - \Delta_1 > 0$ , and it is less difficult to meet if  $\Delta_2 - \Delta_1 < 0$ .

The techniques described above for  $F_{max}$  and hold time analysis can be applied to any circuit in which the same, or related, clock signals are connected to all flip-flops. Consider again the reaction-timer circuit in Figure 5.61. The clock divider in part (a) of the figure generates the  $c_9$  signal, which drives the clock inputs of the flip-flops in the BCD counter.



**Figure 5.68** A general example of clock skew.



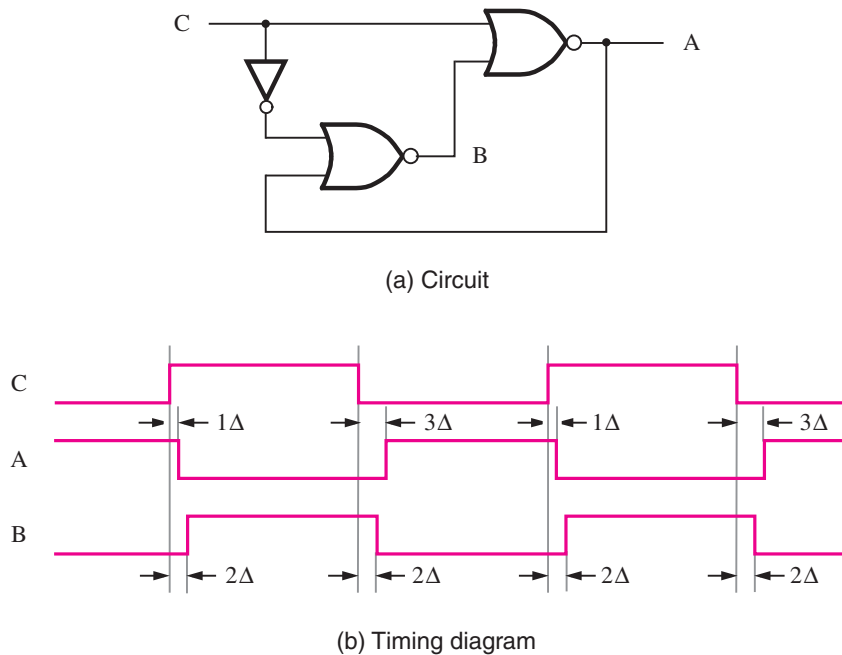
**Figure 5.69** A modified version of the reaction-timer circuit.

Since these paths start at the Q output of the  $c_9$  flip-flop but end at the clock inputs of other flip-flops, rather than at D inputs, the above timing analysis method cannot be applied. However, we could restructure the reaction-timer circuit as indicated in Figure 5.69. Instead of using  $c_9$  directly as a clock for the BCD counter, a ten-input AND gate is used to generate a signal that has the value 1 for only one of the 1024 count values from the clock divider. This signal is then ANDed with the control flip-flop output to cause the BCD counter to be incremented at the desired rate of 100 times per second.

In the circuit of Figure 5.69 all flip-flops are clocked directly by the *Clock* signal. Therefore, the  $F_{max}$  and hold time analysis can now be applied. In general, it is a good design approach for sequential circuits to connect the clock inputs of all flip-flops to a common clock. We discuss such issues in detail in Chapter 7.

## 5.16 CONCLUDING REMARKS

In this chapter we have presented circuits that serve as basic storage elements in digital systems. These elements are used to build larger units such as registers, shift registers, and counters. Many other texts that deal with this material are available [3–10]. We have illustrated how circuits with flip-flops can be described using Verilog code. More information on Verilog can be found in [11–18]. In the next chapter a more formal method for designing circuits with flip-flops will be presented.



**Figure 5.70** Circuit for Example 5.18.

## 5.17 EXAMPLES OF SOLVED PROBLEMS

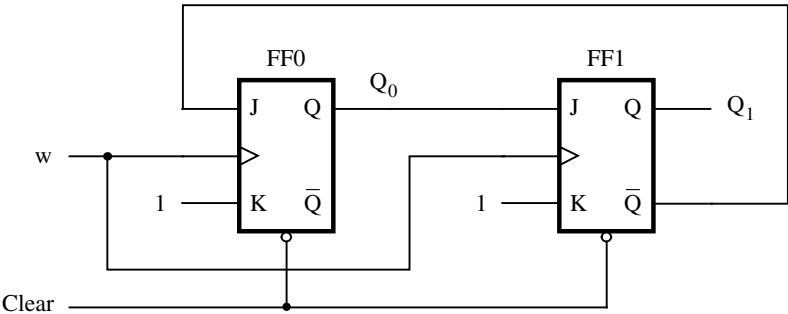
This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

**Problem:** Consider the circuit in Figure 5.70a. Assume that the input  $C$  is driven by a square wave signal with a 50% duty cycle. Draw a timing diagram that shows the waveforms at points  $A$  and  $B$ . Assume that the propagation delay through each gate is  $\Delta$  seconds. **Example 5.18**

**Solution:** The timing diagram is shown in Figure 5.70b.

**Problem:** Determine the functional behavior of the circuit in Figure 5.71. Assume that input  $w$  is driven by a square wave signal. **Example 5.19**

**Solution:** When both flip-flops are cleared, their outputs are  $Q_0 = Q_1 = 0$ . After the *Clear* input goes high, each pulse on the  $w$  input will cause a change in the flip-flops as indicated



**Figure 5.71**     Circuit for Example 5.19.

Time interval	FF0			FF1		
	$J_0$	$K_0$	$Q_0$	$J_1$	$K_1$	$Q_1$
Clear	1	1	0	0	1	0
$t_1$	1	1	1	1	1	0
$t_2$	0	1	0	0	1	1
$t_3$	1	1	0	0	1	0
$t_4$	1	1	1	1	1	0

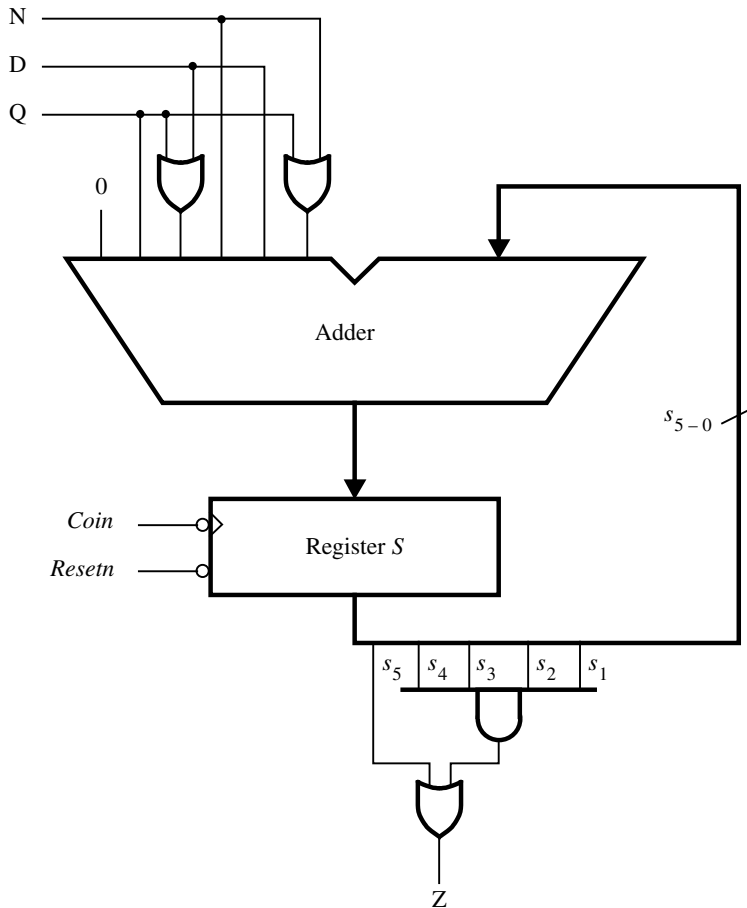
**Figure 5.72**     Summary of the behavior of the circuit in Figure 5.71.

in Figure 5.72. Note that the figure shows the state of the signals after the changes caused by the rising edge of a pulse have taken place.

In consecutive time intervals the values of  $Q_1 Q_0$  are 00, 01, 10, 00, 01, and so on. Therefore, the circuit generates the counting sequence: 0, 1, 2, 0, 1, and so on. Hence, the circuit is a modulo-3 counter.

**Example 5.20 Problem:** Design a circuit that can be used to control a vending machine. The circuit has five inputs: Q (quarter), D (dime), N (nickel), *Coin*, and *Resetrn*. When a coin is deposited in the machine, a coin-sensing mechanism generates a pulse on the appropriate input (Q, D, or N). To signify the occurrence of the event, the mechanism also generates a pulse on the line *Coin*. The circuit is reset by using the *Resetrn* signal (active low). When at least 30 cents has been deposited, the circuit activates its output, Z. No change is given if the amount exceeds 30 cents.

Design the required circuit by using the following components: a six-bit adder, a six-bit register, and any number of AND, OR, and NOT gates.



**Figure 5.73** Circuit for Example 5.20.

**Solution:** Figure 5.73 gives a possible circuit. The value of each coin is represented by a corresponding five-bit number. It is added to the current total, which is held in register  $S$ . The required output is

$$Z = s_5 + s_4 s_3 s_2 s_1$$

The register is clocked by the negative edge of the *Coin* signal. This allows for a propagation delay through the adder, and ensures that a correct sum will be placed into the register.

In Chapter 9 we will show how this type of control circuit can be designed using a more structured approach.

```

module vend (N, D, Q, Resetn, Coin, Z);
  input N, D, Q, Resetn, Coin;
  output Z;
  wire [4:0] X;
  reg [5:0] S;

  assign X[0] = N | Q;
  assign X[1] = D;
  assign X[2] = N;
  assign X[3] = D | Q;
  assign X[4] = Q;
  assign Z = S[5] | (S[4] & S[3] & S[2] & S[1]);

  always @(negedge Coin, negedge Resetn)
    if (Resetn == 1'b0)
      S <= 5'b00000;
    else
      S <= {1'b0, X} + S;

endmodule

```

**Figure 5.74** Code for Example 5.21.

---

**Example 5.21 Problem:** Write Verilog code to implement the circuit in Figure 5.73.

**Solution:** Figure 5.74 gives the desired code.

---

**Example 5.22 Problem:** In Section 5.15 we presented a timing analysis for the counter circuit in Figure 5.67. Redesign this circuit to reduce the logic delay between flip-flops, so that the circuit can operate at a higher maximum clock frequency.

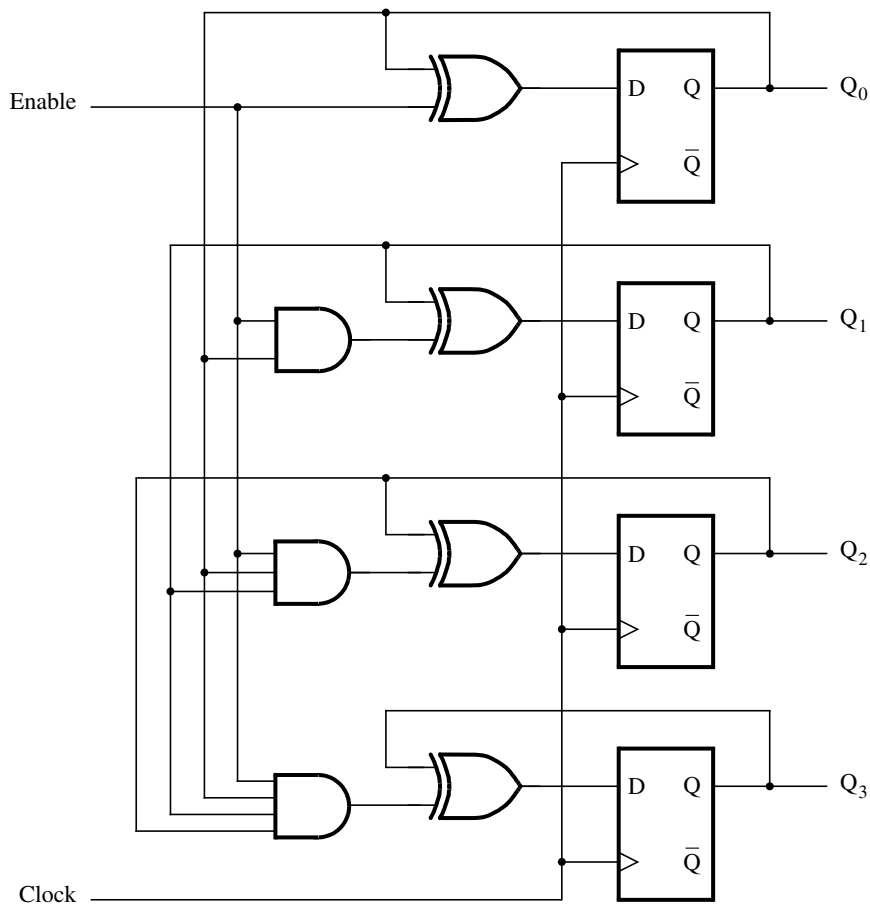
**Solution:** As we showed in Section 5.15, the performance of the counter circuit is limited by the delay through its cascaded AND gates. To increase the circuit's performance we can refactor the AND gates as illustrated in Figure 5.75. The longest delay path in this redesigned circuit, which starts at flip-flop  $Q_0$  and ends at  $Q_3$ , provides the minimum clock period

$$\begin{aligned}
 T_{min} &= t_{cQ} + t_{AND} + t_{XOR} + t_{su} \\
 &= 1.0 + 1.4 + 1.2 + 0.6 \text{ ns} = 4.2 \text{ ns}
 \end{aligned}$$

The redesigned counter has a maximum clock frequency of  $F_{max} = 1/4.2 \text{ ns} = 238.1 \text{ MHz}$ , compared to the result for the original counter, which was 156.25 MHz.

---





**Figure 5.75** A faster 4-bit counter.

**Problem:** In Example 5.17 we showed how to perform timing analysis when there may be different delays associated with the clock signal at each of the flip-flops in a circuit. The circuit in Figure 5.76 includes three flip-flops,  $Q_1$ ,  $Q_2$ , and  $Q_3$ , with corresponding clock delays  $\Delta_1$ ,  $\Delta_2$ , and  $\Delta_3$ . The flip-flop timing parameters are  $t_{su} = 0.6$  ns,  $t_h = 0.4$  ns, and  $0.8 \leq t_{cQ} \leq 1$  ns. Also, the delay through a logic gate is given by  $1 + 0.1k$ , where  $k$  is the number of inputs to the gate.

**Example 5.23**

Calculate the  $F_{max}$  for the circuit for the following sets of clock delays:  $\Delta_1 = \Delta_2 = \Delta_3 = 0$  ns;  $\Delta_1 = \Delta_3 = 0$  ns and  $\Delta_2 = 0.7$  ns;  $\Delta_1 = 1$  ns,  $\Delta_2 = 0$ , and  $\Delta_3 = 0.5$  ns. Also, determine if there are any hold time violations in the circuit for the sets of clock delays  $\Delta_1 = \Delta_2 = \Delta_3 = 0$  ns, and  $\Delta_1 = 1$  ns,  $\Delta_2 = 0$ ,  $\Delta_3 = 0.5$  ns.



The critical path delay is from flip-flop  $Q_1$  to flip-flop  $Q_2$ . Since  $T_{Q_1 \rightarrow Q_2} = 5$  ns, then  $F_{max} = 1/(5 \times 10^{-9}) = 200$  MHz.

To determine whether any hold time violations exist, we need to calculate the shortest path delays, using the minimum value of  $t_{cQ}$ . For  $\Delta_1 = \Delta_2 = \Delta_3 = 0$ , we have

$$T_{Q_1 \rightarrow Q_2} = t_{cQ} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns}$$

$$T_{Q_2 \rightarrow Q_2} = t_{cQ} + t_{AND} = 0.8 + 1.2 = 2 \text{ ns}$$

$$T_{Q_2 \rightarrow Q_3} = t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9 \text{ ns}$$

$$T_{Q_3 \rightarrow Q_1} = t_{cQ} = 0.8 \text{ ns}$$

$$T_{Q_3 \rightarrow Q_2} = t_{cQ} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns}$$

Since the shortest path delay  $T_{Q_3 \rightarrow Q_1} = 0.8 \text{ ns} > t_h$ , there is no hold time violation in the circuit.

We showed in Section 5.15 that if the shortest path delay through a circuit is called  $T_l$ , then a hold time violation occurs if  $T_l - t_{skew} < t_h$ . Adjusting the shortest path delays calculated above for the values  $\Delta_1 = 1$ ,  $\Delta_2 = 0$ , and  $\Delta_3 = 0.5$  gives

$$T_{Q_1 \rightarrow Q_2} = 3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_1) = 3.2 - (0 - 1) = 4.2 \text{ ns}$$

$$T_{Q_2 \rightarrow Q_3} = 1.9 - t_{skew} = 1.9 - (\Delta_3 - \Delta_2) = 1.9 - 0.5 = 1.4 \text{ ns}$$

$$T_{Q_3 \rightarrow Q_1} = 0.8 - t_{skew} = 0.8 - (\Delta_1 - \Delta_3) = 0.8 - (1 - 0.5) = 0.3 \text{ ns}$$

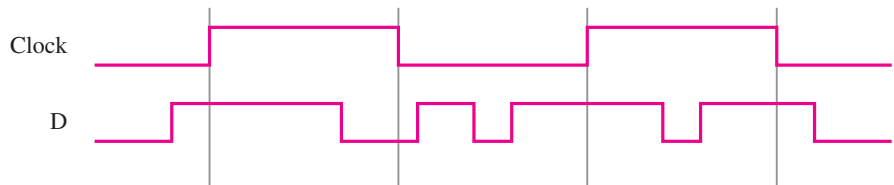
$$T_{Q_3 \rightarrow Q_2} = 3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_3) = 3.2 - (0 - 0.5) = 3.7 \text{ ns}$$

The shortest path delay is  $T_{Q_3 \rightarrow Q_1} = 0.3 \text{ ns} < t_h$ , which represents a hold time violation. Hence, the circuit may not function reliably regardless of the frequency of the clock signal.

## PROBLEMS

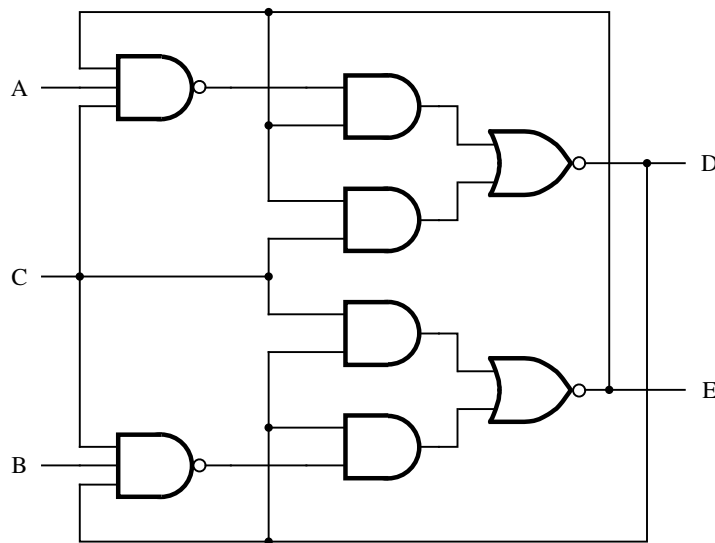
Answers to problems marked by an asterisk are given at the back of the book.

- 5.1** Consider the timing diagram in Figure P5.1. Assuming that the  $D$  and  $Clock$  inputs shown are applied to the circuit in Figure 5.10, draw waveforms for the  $Q_a$ ,  $Q_b$ , and  $Q_c$  signals.



**Figure P5.1** Timing diagram for Problem 5.1.

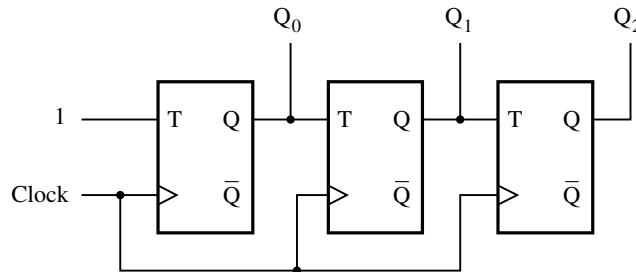
- 5.2** Figure 5.4 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its characteristic table and show its timing diagram.
- \*5.3** Show a circuit that implements the gated SR latch using NAND gates only.
- 5.4** Given a 100-MHz clock signal, derive a circuit using D flip-flops to generate 50-MHz and 25-MHz clock signals. Draw a timing diagram for all three clock signals, assuming reasonable delays.
- \*5.5** An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.
- 5.6** The gated SR latch in Figure 5.5a has unpredictable behavior if the  $S$  and  $R$  inputs are both equal to 1 when the  $Clk$  changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition  $S = R = 1$  causes the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.
- 5.7** Show how a JK flip-flop can be constructed using a T flip-flop and other logic gates.
- \*5.8** Consider the circuit in Figure P5.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?



**Figure P5.2** Circuit for Problem 5.8.

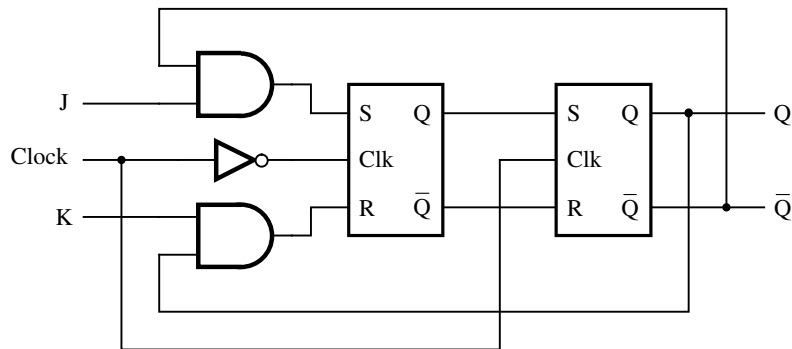
- 5.9** Write Verilog code that represents a T flip-flop with an asynchronous clear input. Use behavioral code, rather than structural code.
- 5.10** Write Verilog code that represents a JK flip-flop. Use behavioral code, rather than structural code.

- 5.11** Synthesize a circuit for the code written for Problem 5.10 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.
- 5.12** A universal shift register can shift in both the left-to-right and right-to-left directions, and it has parallel-load capability. Draw a circuit for such a shift register.
- 5.13** Write Verilog code for a universal shift register with  $n$  bits.
- 5.14** Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in Section 5.9.3.
- \*5.15** Design a three-bit up/down counter using T flip-flops. It should include a control input called  $\overline{Up}/Down$ . If  $\overline{Up}/Down = 0$ , then the circuit should behave as an up-counter. If  $\overline{Up}/Down = 1$ , then the circuit should behave as a down-counter.
- 5.16** Repeat Problem 5.15 using D flip-flops.
- \*5.17** The circuit in Figure P5.3 looks like a counter. What is the counting sequence of this circuit?



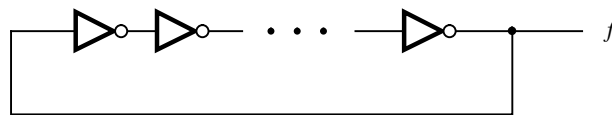
**Figure P5.3** The circuit for Problem 5.17.

- 5.18** Consider the circuit in Figure P5.4. How does this circuit compare with the circuit in Figure 5.16? Can the circuits be used for the same purposes? If not, what is the key difference between them?



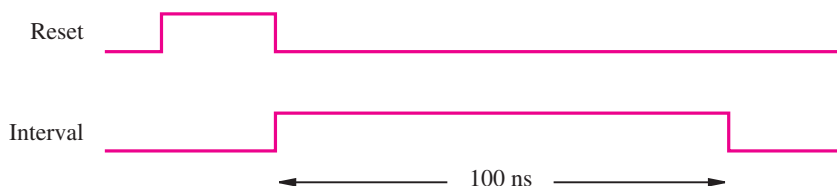
**Figure P5.4** Circuit for Problem 5.18.

- 5.19** Construct a NOR-gate circuit, similar to the one in Figure 5.11a, which implements a negative-edge-triggered D flip-flop.
- 5.20** Write Verilog code that represents a modulo-12 up-counter with synchronous reset.
- \*5.21** For the flip-flops in the counter in Figure 5.24, assume that  $t_{su} = 3$  ns,  $t_h = 1$  ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?
- 5.22** Write Verilog code that represents an eight-bit Johnson counter. Synthesize the code with your CAD tools and give a timing simulation that shows the counting sequence.
- 5.23** Write Verilog code in the style shown in Figure 5.51 that represents a ring counter. Your code should have a parameter  $n$  that sets the number of flip-flops in the counter.
- 5.24** A ring oscillator is a circuit that has an odd number,  $n$ , of inverters connected in a ringlike structure, as shown in Figure P5.5. The output of each inverter is a periodic signal with a certain period.



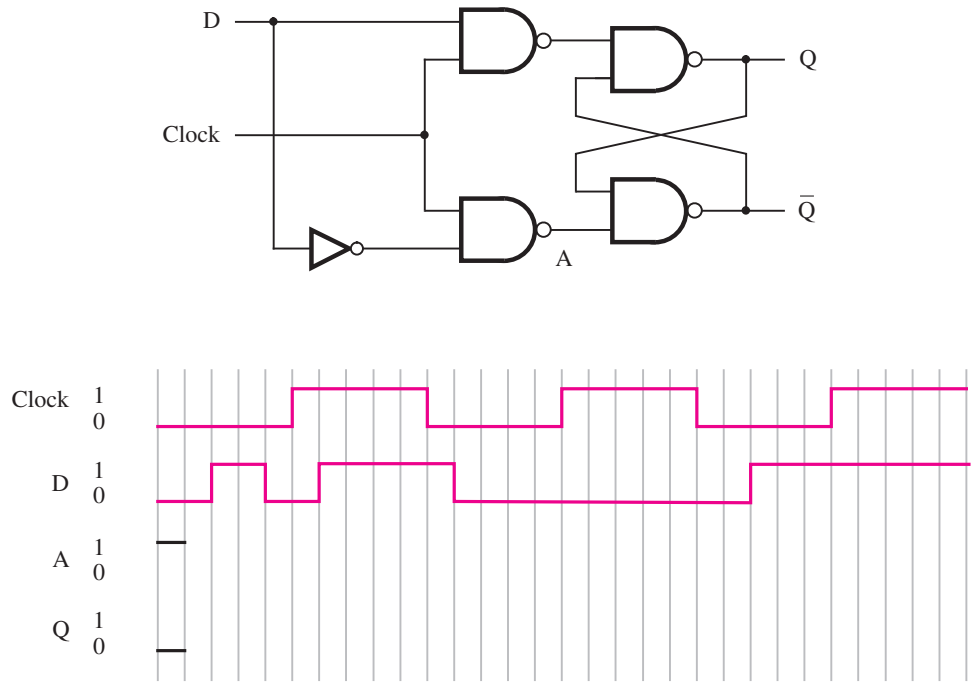
**Figure P5.5** A ring oscillator.

- (a) Assume that all the inverters are identical; hence they all have the same delay, called  $t_p$ . Let the output of one of the inverters be named  $f$ . Give an equation that expresses the period of the signal  $f$  in terms of  $n$  and  $t_p$ .
- (b) For this part you are to design a circuit that can be used to experimentally measure the delay  $t_p$  through one of the inverters in the ring oscillator. Assume the existence of an input called *Reset* and another called *Interval*. The timing of these two signals is shown in Figure P5.6. The length of time for which *Interval* has the value 1 is known. Assume that this length of time is 100 ns. Design a circuit that uses the *Reset* and *Interval* signals and the signal  $f$  from part (a) to experimentally measure  $t_p$ . In your design you may use logic gates and subcircuits such as adders, flip-flops, counters, registers, and so on.



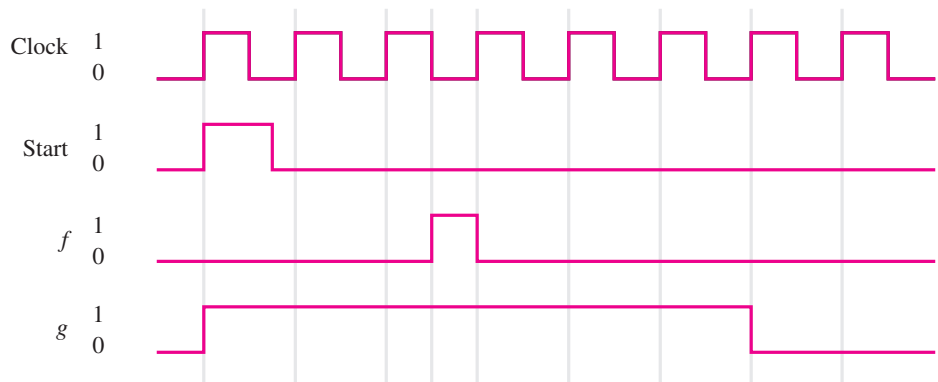
**Figure P5.6** Timing of signals for Problem 5.24.

- 5.25** A circuit for a gated D latch is shown in Figure P5.7. Assume that the propagation delay through either a NAND gate or an inverter is 1 ns. Complete the timing diagram given in the figure, which shows the signal values with 1 ns resolution.



**Figure P5.7** Circuit and timing diagram for Problem 5.25.

- \*5.26** A logic circuit has two inputs, *Clock* and *Start*, and two outputs, *f* and *g*. The behavior of the circuit is described by the timing diagram in Figure P5.8. When a pulse is received



**Figure P5.8** Timing diagram for Problem 5.26.

on the *Start* input, the circuit produces pulses on the *f* and *g* outputs as shown in the timing diagram. Design a suitable circuit using only the following components: a three-bit resettable positive-edge-triggered synchronous counter and basic logic gates. For your answer assume that the delays through all logic gates and the counter are negligible.

**5.27** The following code checks for adjacent ones in an  $n$ -bit vector.

```
always @(A)
begin
    f = A[1] & A[0];
    for (k = 2; k < n; k = k+1)
        f = f | (A[k] & A[k-1]);
end
```

With blocking assignments this code produces the desired logic function, which is  $f = a_1a_0 + \dots + a_{n-1}a_{n-2}$ . What logic function is produced if we change the code to use non-blocking assignments?

**5.28** The Verilog code in Figure P5.9 represents a 3-bit *linear-feedback shift register* (LFSR). This type of circuit generates a counting sequence of pseudo-random numbers that repeats after  $2^n - 1$  clock cycles, where  $n$  is the number of flip-flops in the LFSR. Synthesize a circuit to implement the LFSR in a chip. Draw a diagram of the circuit. Simulate the circuit's behavior by loading the pattern 001 into the LFSR and then enabling the register to count. What is the counting sequence?

```
module lfsr (R, L, Clock, Q);
    input [0:2] R;
    input L, Clock;
    output reg [0:2] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
            Q <= {Q[2], Q[0] ^ Q[2], Q[1]};

endmodule
```

**Figure P5.9** Code for a linear-feedback shift register.



**5.29** Repeat Problem 5.28 for the Verilog code in Figure P5.10.

```

module lfsr (R, L, Clock, Q);
  input [0:2] R;
  input L, Clock;
  output reg [0:2] Q;

  always @(posedge Clock)
    if (L)
      Q <= R;
    else
      Q <= {Q[2], Q[0], Q[1] ^ Q[2]};

endmodule

```

**Figure P5.10** Code for a linear-feedback shift register.

**5.30** The Verilog code in Figure P5.11 is equivalent to the code in Figure P5.9, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

```

module lfsr (R, L, Clock, Q);
  input [0:2] R;
  input L, Clock;
  output reg [0:2] Q;

  always @(posedge Clock)
    if (L)
      Q <= R;
    else
      begin
        Q[0] = Q[2];
        Q[1] = Q[0] ^ Q[2];
        Q[2] = Q[1];
      end

endmodule

```

**Figure P5.11** Code for Problem 5.30.

- 5.31** The Verilog code in Figure P5.12 is equivalent to the code in Figure P5.10, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

```
module lfsr (R, L, Clock, Q);  
    input [0:2] R;  
    input L, Clock;  
    output reg [0:2] Q;  
  
    always @(posedge Clock)  
        if (L)  
            Q <= R;  
        else  
            begin  
                Q[0] = Q[2];  
                Q[1] = Q[0];  
                Q[2] = Q[1] ^ Q[2];  
            end  
end  
  
endmodule
```

**Figure P5.12** Code for Problem 5.31.

- 5.32** The circuit in Figure 5.59 gives a shift register in which the parallel-load control input is independent of the enable input. Show a different shift register circuit in which the parallel-load operation can be performed only when the enable input is also asserted.

---

## REFERENCES

1. C. Hamacher, Z. Vranesic, S. Zaky, and N. Manjikian, *Computer Organization and Embedded Systems*, 6th ed., (McGraw-Hill: New York, 2011).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed., (Morgan Kaufmann: San Francisco, Ca., 2004).
3. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
4. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
5. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
6. M. M. Mano, *Digital Design*, 3rd ed. (Prentice-Hall: Upper Saddle River, N.J., 2002).
7. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, N.J., 1997).
8. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
9. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, N.J., 1995).
10. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, Ma., 1993).
11. Institute of Electrical and Electronics Engineers, *IEEE Standard Verilog Hardware Description Language Reference Manual*, (IEEE: Piscataway, NJ, 2001).
12. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
13. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
14. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
15. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
16. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
17. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
18. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

*This page intentionally left blank*