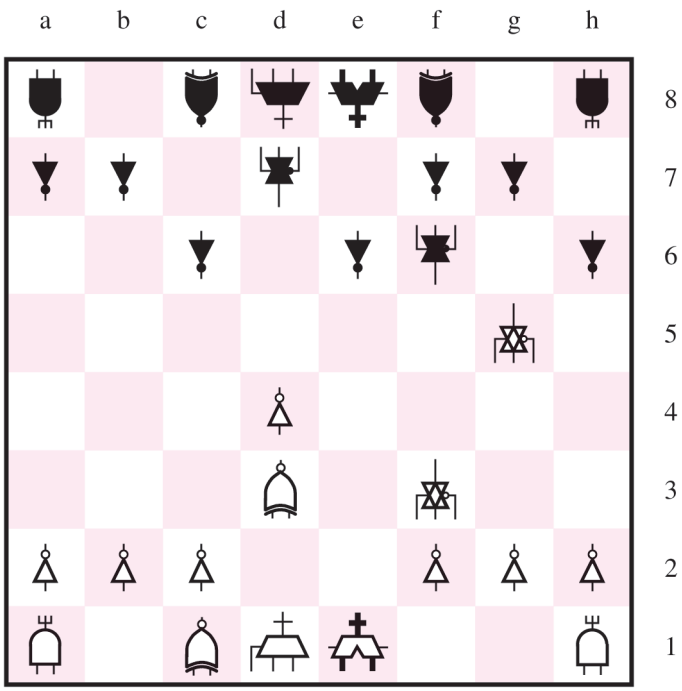
chapter

7

FLIP-FLOPS, REGISTERS, COUNTERS, AND A SIMPLE PROCESSOR



7. Ng1–f3, h7–h6

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

In this chapter we will introduce circuits that can be used as storage elements. But first, we will motivate the need for such circuits by means of a simple example. Suppose that we wish to control an alarm system, as shown in Figure 7.1. The alarm mechanism responds to the control input $On/\overline{Off}$. It is turned on when $On/\overline{Off} = 1$, and it is off when $On/\overline{Off} = 0$. The desired operation is that the alarm turns on when the sensor generates a positive voltage signal, *Set*, in response to some undesirable event. Once the alarm is triggered, it must remain active even if the sensor output goes back to zero. The alarm is turned off manually by means of a *Reset* input. The circuit requires a memory element to remember that the alarm has to be active until the *Reset* signal arrives.

Figure 7.2 gives a rudimentary memory element, consisting of a loop that has two inverters. If we assume that $A = 0$, then $B = 1$. The circuit will maintain these values indefinitely. We say that the circuit is in the *state* defined by these values. If we assume that $A = 1$, then $B = 0$, and the circuit will remain in this second state indefinitely. Thus the circuit has two possible states. This circuit is not useful, because it lacks some practical means for changing its state.

A more useful circuit is shown in Figure 7.3. It includes a mechanism for changing the state of the circuit in Figure 7.2, using two transmission gates of the type discussed in section 3.9. One transmission gate, *TG*1, is used to connect the *Data* input terminal to point

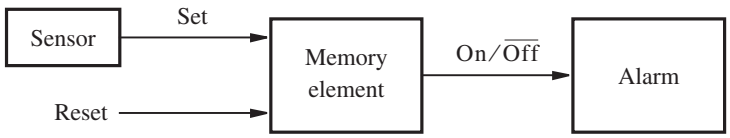


Figure 7.1 Control of an alarm system.

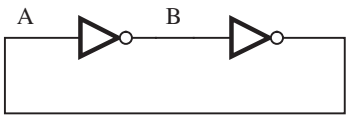


Figure 7.2 A simple memory element.

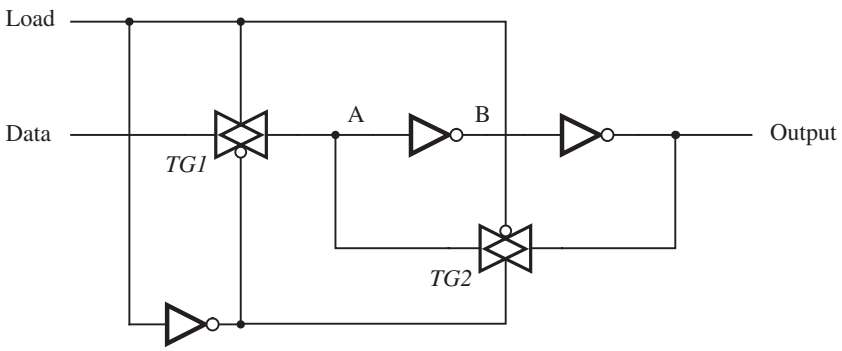


Figure 7.3 A controlled memory element.

A in the circuit. The second, *TG*2, is used as a switch in the *feedback loop* that maintains the state of the circuit. The transmission gates are controlled by the *Load* signal. If *Load* = 1, then *TG*1 is on and the point *A* will have the same value as the *Data* input. Since the value presently stored at *Output* may not be the same value as *Data*, the feedback loop is broken by having *TG*2 turned off when *Load* = 1. When *Load* changes to zero, then *TG*1 turns off and *TG*2 turns on. The feedback path is closed and the memory element will retain its state as long as *Load* = 0. This memory element cannot be applied directly to the system in Figure 7.1, but it is useful for many other applications, as we will see later.

7.1 BASIC LATCH

Instead of using the transmission gates, we can construct a similar circuit using ordinary logic gates. Figure 7.4 presents a memory element built with NOR gates. Its inputs, *Set* and *Reset*, provide the means for changing the state, Q, of the circuit. A more usual way of drawing this circuit is given in Figure 7.5*a*, where the two NOR gates are said to be connected in cross-coupled style. The circuit is referred to as a *basic latch*. Its behavior is described by the truth table in Figure 7.5*b*. When both inputs, *R* and *S*, are equal to 0 the latch maintains its existing state. This state may be either $Q_a = 0$ and $Q_b = 1$, or $Q_a = 1$ and $Q_b = 0$, which is indicated in the truth table by stating that the Q_a and Q_b outputs have values 0/1 and 1/0, respectively. Observe that Q_a and Q_b are complements of each other in this case. When $R = 0$ and $S = 1$, the latch is *set* into a state where $Q_a = 1$ and $Q_b = 0$. When $R = 1$ and $S = 0$, the latch is *reset* into a state where $Q_a = 0$ and $Q_b = 1$. The fourth possibility is to have $R = S = 1$. In this case both Q_a and Q_b will be 0.

Figure 7.5*c* gives a timing diagram for the latch, assuming that the propagation delay through the NOR gates is negligible. Of course, in a real circuit the changes in the waveforms would be delayed according to the propagation delays of the gates. We assume that initially $Q_a = 0$ and $Q_b = 1$. The state of the latch remains unchanged until time t_2,

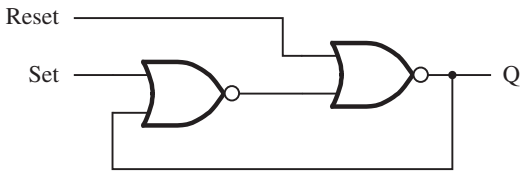
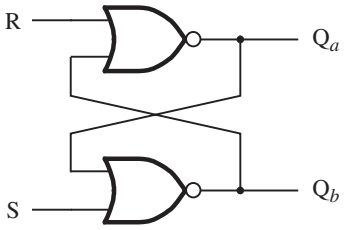


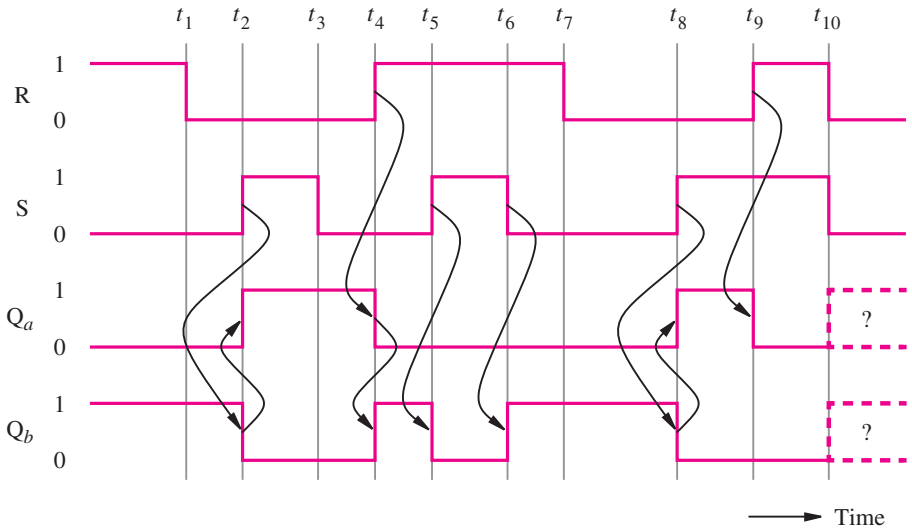
Figure 7.4 A memory element with NOR gates.



(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(No change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Truth table



(c) Timing diagram

Figure 7.5 A basic latch built with NOR gates.

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

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

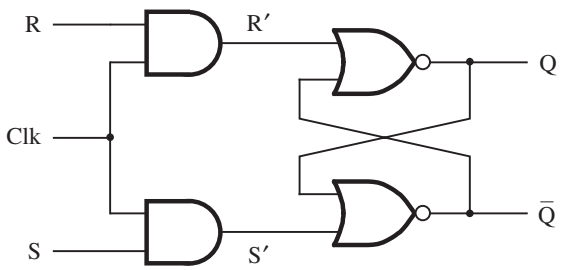
The latch in Figure 7.5*a* can perform the functions needed for the memory element in Figure 7.1, by connecting the *Set* signal to the S input and *Reset* to the R input. The Q_a output provides the desired $On/\overline{Off}$ signal. To initialize the operation of the alarm system, the latch is reset. Thus the alarm is off. When the sensor generates the logic value 1, the latch is set and Q_a becomes equal to 1. This turns on the alarm mechanism. If the sensor output returns to 0, the latch retains its state where $Q_a = 1$; hence the alarm remains turned on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the *Reset* input equal to 1.

7.2 GATED SR LATCH

In section 7.1 we saw that the basic SR latch can serve as a useful memory element. It remembers its state when both the S and R inputs are 0. It changes its state in response to changes in the signals on these inputs. The state changes occur at the time when the changes in the signals occur. If we cannot control the time of such changes, then we don't know when the latch may change its state.

In the alarm system of Figure 7.1, it may be desirable to be able to enable or disable the entire system by means of a control input, *Enable*. Thus when enabled, the system would function as described above. In the disabled mode, changing the *Set* input from 0 to 1 would not cause the alarm to turn on. The latch in Figure 7.5*a* cannot provide the desired operation. But the latch circuit can be modified to respond to the input signals S and R only when $Enable = 1$. Otherwise, it would maintain its state.

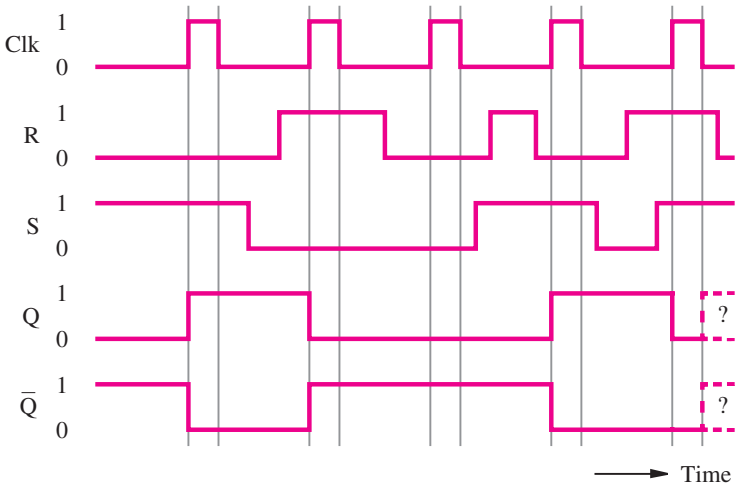
The modified circuit is depicted in Figure 7.6*a*. It includes two AND gates that provide the desired control. When the control signal *Clk* is equal to 0, the S' and R' inputs to the



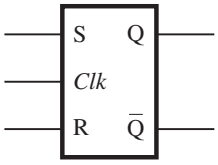
(a) Circuit

Clk	S	R	Q(t + 1)
0	x	x	Q(t) (No change)
1	0	0	Q(t) (No change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Truth table



(c) Timing diagram



(d) Graphical symbol

Figure 7.6 Gated SR latch.

latch will be 0, regardless of the values of signals S and R. Hence the latch will maintain its existing state as long as $Clk = 0$. When Clk changes to 1, the S' and R' signals will be the same as the S and R signals, respectively. Therefore, in this mode the latch will behave as we described in section 7.1. Note that we have used the name Clk for the control signal that allows the latch to be set or reset, rather than call it the *Enable* signal. The reason is that

such circuits are often used in digital systems where it is desirable to allow the changes in the states of memory elements to occur only at well-defined time intervals, as if they were controlled by a clock. The control signal that defines these time intervals is usually called the *clock* signal. The name *Clk* is meant to reflect this nature of the signal.

Circuits of this type, which use a control signal, are called *gated latches*. Because our circuit exhibits set and reset capability, it is called a *gated SR latch*. Figure 7.6*b* describes its behavior. It defines the state of the Q output at time $t+1$, namely, $Q(t+1)$, as a function of the inputs S, R, and *Clk*. When $Clk = 0$, the latch will remain in the state it is in at time t, that is, $Q(t)$, regardless of the values of inputs S and R. This is indicated by specifying $S = x$ and $R = x$, where x means that the signal value can be either 0 or 1. (Recall that we already used this notation in Chapter 4.) When $Clk = 1$, the circuit behaves as the basic latch in Figure 7.5. It is set by $S = 1$ and reset by $R = 1$. The last row of the truth table, where $S = R = 1$, shows that the state $Q(t + 1)$ is undefined because we don't know whether it will be 0 or 1. This corresponds to the situation described in section 7.1 in conjunction with the timing diagram in Figure 7.5 at time t_{10}. At this time both S and R inputs go from 1 to 0, which causes the oscillatory behavior that we discussed. If $S = R = 1$, this situation will occur as soon as *Clk* goes from 1 to 0. To ensure a meaningful operation of the gated SR latch, it is essential to avoid the possibility of having both the S and R inputs equal to 1 when *Clk* changes from 1 to 0.

A timing diagram for the gated SR latch is given in Figure 7.6*c*. It shows *Clk* as a periodic signal that is equal to 1 at regular time intervals to suggest that this is how the clock signal usually appears in a real system. The diagram presents the effect of several combinations of signal values. Observe that we have labeled one output as Q and the other as its complement $\overline{Q}$, rather than Q_a and Q_b as in Figure 7.5. Since the undefined mode, where $S = R = 1$, must be avoided in practice, the normal operation of the latch will have the outputs as complements of each other. Moreover, we will often say that the latch is *set* when $Q = 1$, and it is *reset* when $Q = 0$. A graphical symbol for the gated SR latch is given in Figure 7.6*d*.

7.2.1 GATED SR LATCH WITH NAND GATES

So far we have implemented the basic latch with cross-coupled NOR gates. We can also construct the latch with NAND gates. Using this approach, we can implement the gated SR latch as depicted in Figure 7.7. The behavior of this circuit is described by the truth table

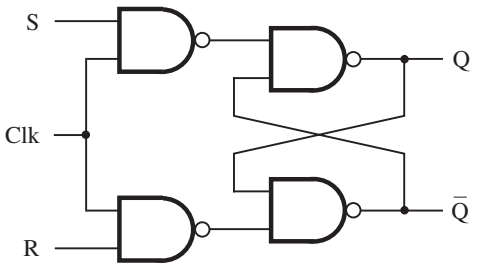


Figure 7.7 Gated SR latch with NAND gates.

in Figure 7.6*b*. Note that in this circuit, the clock is gated by NAND gates, rather than by AND gates. Note also that the *S* and *R* inputs are reversed in comparison with the circuit in Figure 7.6*a*. The circuit with NAND gates requires fewer transistors than the circuit with AND gates. We will use the circuit in Figure 7.7, in preference to the circuit in Figure 7.6*a*.

7.3 GATED D LATCH

In section 7.2 we presented the gated SR latch and showed how it can be used as the memory element in the alarm system of Figure 7.1. This latch is useful for many other applications. In this section we describe another gated latch that is even more useful in practice. It has a single data input, called *D*, and it stores the value on this input, under the control of a clock signal. It is called a *gated D latch*.

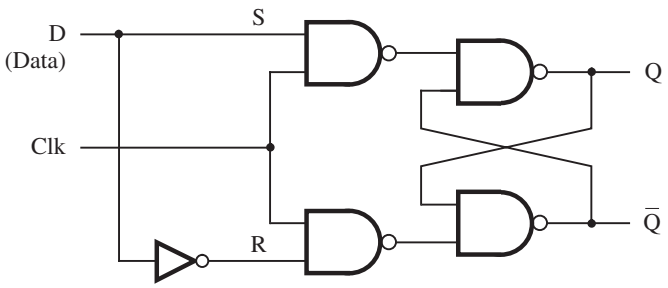
To motivate the need for a gated D latch, consider the adder/subtractor unit discussed in Chapter 5 (Figure 5.13). When we described how that circuit is used to add numbers, we did not discuss what is likely to happen with the sum bits that are produced by the adder. Adder/subtractor units are often used as part of a computer. The result of an addition or subtraction operation is often used as an operand in a subsequent operation. Therefore, it is necessary to be able to remember the values of the sum bits generated by the adder until they are needed again. We might think of using the basic latches to remember these bits, one bit per latch. In this context, instead of saying that a latch remembers the value of a bit, it is more illuminating to say that the latch *stores* the value of the bit or simply "stores the bit." We should think of the latch as a storage element.

But can we obtain the desired operation using the basic latches? We can certainly reset all latches before the addition operation begins. Then we would expect that by connecting a sum bit to the *S* input of a latch, the latch would be set to 1 if the sum bit has the value 1; otherwise, the latch would remain in the 0 state. This would work fine if all sum bits are 0 at the start of the addition operation and, after some propagation delay through the adder, some of these bits become equal to 1 to give the desired sum. Unfortunately, the propagation delays that exist in the adder circuit cause a big problem in this arrangement. Suppose that we use a ripple-carry adder. When the *X* and *Y* inputs are applied to the adder, the sum outputs may alternate between 0 and 1 a number of times as the carries ripple through the circuit. This situation was illustrated in the timing diagram in Figure 5.21. The problem is that if we connect a sum bit to the *S* input of a latch, then if the sum bit is temporarily a 1 and then settles to 0 in the final result, the latch will remain set to 1 erroneously.

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

an awkward way of dealing with the problem, and it is preferable to use the gated D latches instead.

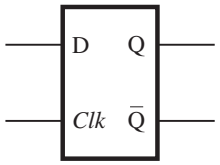
Figure 7.8*a* shows the circuit for a gated D latch. It is based on the gated SR latch, but instead of using the S and R inputs separately, it has just one data input, D. For convenience we have labeled the points in the circuit that are equivalent to the S and R inputs. If $D = 1$, then $S = 1$ and $R = 0$, which forces the latch into the state $Q = 1$. If $D = 0$, then $S = 0$ and $R = 1$, which causes $Q = 0$. Of course, the changes in state occur only when $Clk = 1$.



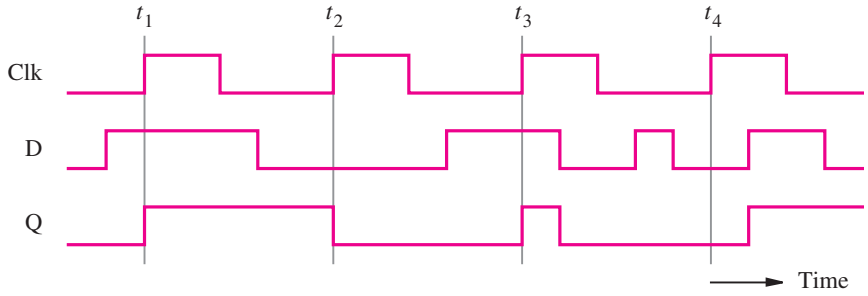
(a) Circuit

Clk	D	Q($t+1$)
0	x	Q(t)
1	0	0
1	1	1

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure 7.8 Gated D latch.

It is important to observe that in this circuit it is impossible to have the troublesome situation where $S = R = 1$. In the gated D latch, the output Q merely tracks the value of the input D while $Clk = 1$. As soon as Clk goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the D input seen at the time the clock changes from 1 to 0. Figure 7.8 also gives the truth table, the graphical symbol, and the timing diagram for the gated D latch.

The timing diagram illustrates what happens if the D signal changes while $Clk = 1$. During the third clock pulse, starting at t_3, the output Q changes to 1 because $D = 1$. But midway through the pulse D goes to 0, which causes Q to go to 0. This value of Q is stored when Clk changes to 0. Now no further change in the state of the latch occurs until the next clock pulse, at t_4. The key point to observe is that as long as the clock has the value 1, the Q output follows the D input. But when the clock has the value 0, the Q output cannot change. In Chapter 3 we saw that the logic values are implemented as low and high voltage levels. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be *level sensitive*. The circuits in Figures 7.6 through 7.8 are level sensitive. We will show in section 7.4 that it is possible to design storage elements for which the output changes only at the point in time when the clock changes from one value to the other. Such circuits are said to be *edge triggered*.

At this point we should reconsider the circuit in Figure 7.3. Careful examination of that circuit shows that it behaves in exactly the same way as the circuit in Figure 7.8*a*. The *Data* and *Load* inputs correspond to the D and Clk inputs, respectively. The *Output*, which has the same signal value as point A, corresponds to the Q output. Point B corresponds to $\overline{Q}$. Therefore, the circuit in Figure 7.3 is also a gated D latch. An advantage of this circuit is that it can be implemented using fewer transistors than the circuit in Figure 7.8*a*.

7.3.1 EFFECTS OF PROPAGATION DELAYS

In the previous discussion we ignored the effects of propagation delays. In practical circuits it is essential to take these delays into account. Consider the gated D latch in Figure 7.8*a*. It stores the value of the D input that is present at the time the clock signal changes from 1 to 0. It operates properly if the D signal is stable (that is, not changing) at the time Clk goes from 1 to 0. But it may lead to unpredictable results if the D signal also changes at this time. Therefore, the designer of a logic circuit that generates the D signal must ensure that this signal is stable when the critical change in the clock signal takes place.

Figure 7.9 illustrates the critical timing region. The minimum time that the D signal must be stable prior to the negative edge of the Clk signal is called the *setup time*, t_{su}, of the latch. The minimum time that the D signal must remain stable after the negative edge of the Clk signal is called the *hold time*, t_h, of the latch. The values of t_{su} and t_h depend on the technology used. Manufacturers of integrated circuit chips provide this information on the data sheets that describe their chips. Typical values for CMOS technology are $t_{su} = 3$ ns and $t_h = 2$ ns. We will give examples of how setup and hold times affect the speed of operation of circuits in section 7.13. The behavior of storage elements when setup or hold times are violated is discussed in section 10.3.3.

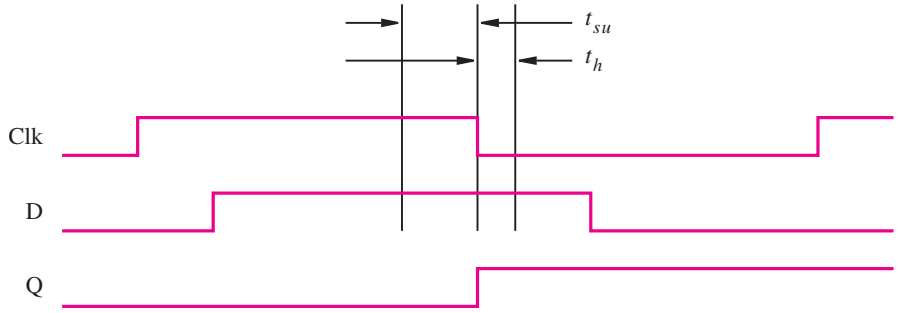


Figure 7.9 Setup and hold times.

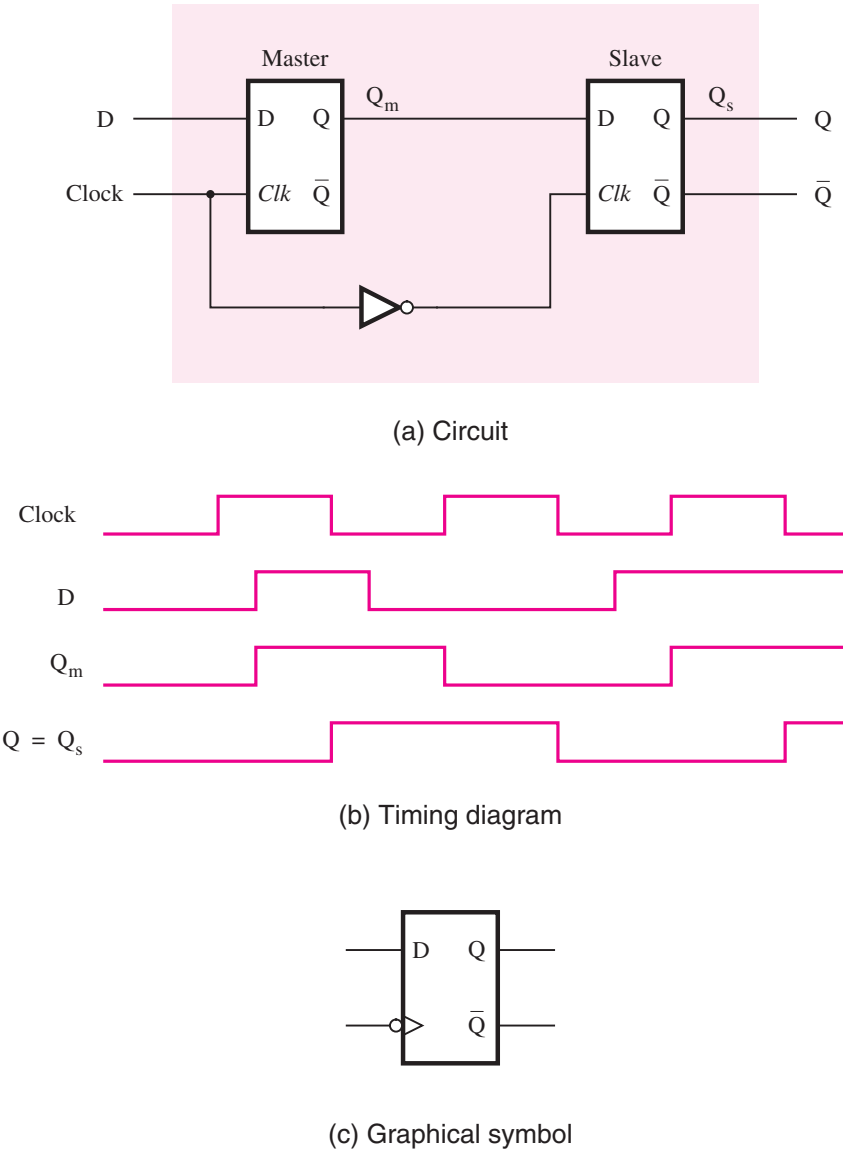
7.4 MASTER-SLAVE AND EDGE-TRIGGERED D FLIP-FLOPS

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the clock signal is active (equal to 1 in our examples). As we will see in sections 7.8 and 7.9, there is also a need for storage elements that can change their states no more than once during one clock cycle. We will discuss two types of circuits that exhibit such behavior.

7.4.1 MASTER-SLAVE D FLIP-FLOP

Consider the circuit given in Figure 7.10*a*, which consists of two gated D latches. The first, called *master*, changes its state while $Clock = 1$. The second, called *slave*, changes its state while $Clock = 0$. The operation of the circuit is such that when the clock is high, the master tracks the value of the D input signal and the slave does not change. Thus the value of Q_m follows any changes in D, and the value of Q_s remains constant. When the clock signal changes to 0, the master stage stops following the changes in the D input. At the same time, the slave stage responds to the value of the signal Q_m and changes state accordingly. Since Q_m does not change while $Clock = 0$, the slave stage can undergo at most one change of state during a clock cycle. From the external observer's point of view, namely, the circuit connected to the output of the slave stage, the master-slave circuit changes its state at the negative-going edge of the clock. The *negative edge* is the edge where the clock signal changes from 1 to 0. Regardless of the number of changes in the D input to the master stage during one clock cycle, the observer of the Q_s signal will see only the change that corresponds to the D input at the negative edge of the clock.

The circuit in Figure 7.10 is called a *master-slave D flip-flop*. The term *flip-flop* denotes a storage element that changes its output state at the edge of a controlling clock signal. The timing diagram for this flip-flop is shown in Figure 7.10*b*. A graphical symbol is given in Figure 7.10*c*. In the symbol we use the $>$ mark to denote that the flip-flop responds to the "active edge" of the clock. We place a bubble on the clock input to indicate that the active edge for this particular circuit is the negative edge.



(a) Circuit

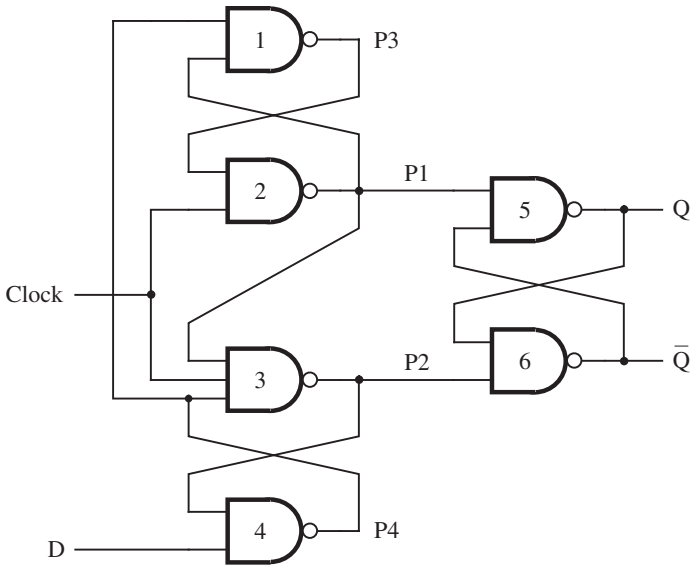
(b) Timing diagram

(c) Graphical symbol

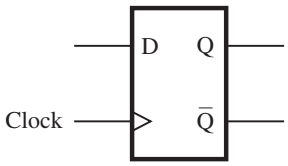
Figure 7.10 Master-slave D flip-flop.

7.4.2 EDGE-TRIGGERED D FLIP-FLOP

The output of the master-slave D flip-flop in Figure 7.10*a* responds on the negative edge of the clock signal. The circuit can be changed to respond to the positive clock edge by connecting the slave stage directly to the clock and the master stage to the complement of the clock. A different circuit that accomplishes the same task is presented in Figure 7.11*a*.



(a) Circuit



(b) Graphical symbol

Figure 7.11 A positive-edge-triggered D flip-flop.

It requires only six NAND gates and, hence, fewer transistors. The operation of the circuit is as follows. When $Clock = 0$, the outputs of gates 2 and 3 are high. Thus $P1 = P2 = 1$, which maintains the output latch, comprising gates 5 and 6, in its present state. At the same time, the signal $P3$ is equal to D, and $P4$ is equal to its complement $\overline{D}$. When *Clock* changes to 1, the following changes take place. The values of $P3$ and $P4$ are transmitted through gates 2 and 3 to cause $P1 = \overline{D}$ and $P2 = D$, which sets $Q = D$ and $\overline{Q} = \overline{D}$. To operate reliably, $P3$ and $P4$ must be stable when *Clock* changes from 0 to 1. Hence the setup time of the flip-flop is equal to the delay from the D input through gates 4 and 1 to $P3$. The hold time is given by the delay through gate 3 because once $P2$ is stable, the changes in D no longer matter.

For proper operation it is necessary to show that, after *Clock* changes to 1, any further changes in D will not affect the output latch as long as $Clock = 1$. We have to consider two cases. Suppose first that $D = 0$ at the positive edge of the clock. Then $P2 = 0$, which will

keep the output of gate 4 equal to 1 as long as $Clock = 1$, regardless of the value of the D input. The second case is if $D = 1$ at the positive edge of the clock. Then $P1 = 0$, which forces the outputs of gates 1 and 3 to be equal to 1, regardless of the D input. Therefore, the flip-flop ignores changes in the D input while $Clock = 1$.

Figure 7.11*b* gives a graphical symbol for this flip-flop. The clock input indicates that the positive edge of the clock is the active edge. A similar circuit, constructed with NOR gates, can be used as a negative-edge-triggered flip-flop.

Level-Sensitive versus Edge-Triggered Storage Elements

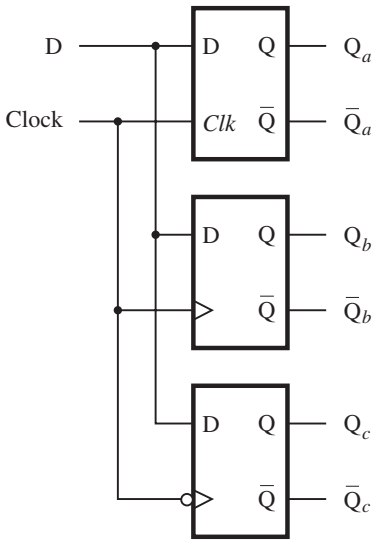
Figure 7.12 shows three different types of storage elements that are driven by the same data and clock inputs. The first element is a gated D latch, which is level sensitive. The second one is a positive-edge-triggered D flip-flop, and the third one is a negative-edge-triggered D flip-flop. To accentuate the differences between these storage elements, the D input changes its values more than once during each half of the clock cycle. Observe that the gated D latch follows the D input as long as the clock is high. The positive-edge-triggered flip-flop responds only to the value of D when the clock changes from 0 to 1. The negative-edge-triggered flip-flop responds only to the value of D when the clock changes from 1 to 0.

7.4.3 D FLIP-FLOPS WITH CLEAR AND PRESET

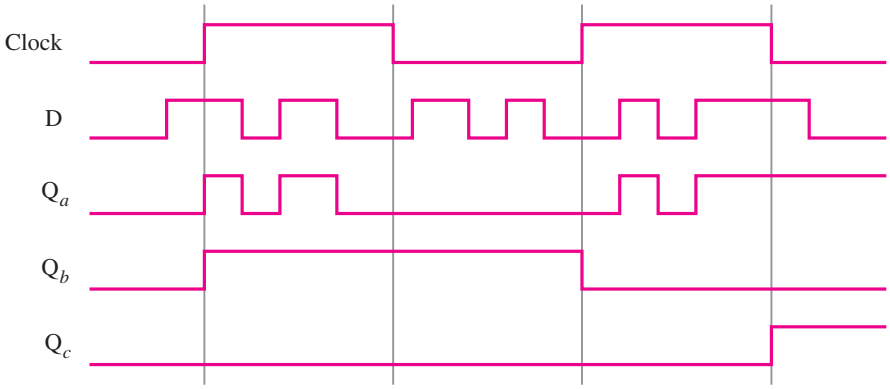
Flip-flops are often used for implementation of circuits that can have many possible states, where the response of the circuit depends not only on the present values of the circuit's inputs but also on the particular state that the circuit is in at that time. We will discuss a general form of such circuits in Chapter 8. A simple example is a counter circuit that counts the number of occurrences of some event, perhaps passage of time. We will discuss counters in detail in section 7.9. A counter comprises a number of flip-flops, whose outputs are interpreted as a number. The counter circuit has to be able to increment or decrement the number. It is also important to be able to force the counter into a known initial state (count). Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have $Q = 0$. It is equally useful to be able to preset each flip-flop to $Q = 1$, to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures 7.10 and 7.11 as follows.

Figure 7.13*a* shows an implementation of the circuit in Figure 7.10*a* using NAND gates. The master stage is just the gated D latch of Figure 7.8*a*. Instead of using another latch of the same type for the slave stage, we can use the slightly simpler gated SR latch of Figure 7.7. This eliminates one NOT gate from the circuit.

A simple way of providing the clear and preset capability is to add an extra input to each NAND gate in the cross-coupled latches, as indicated in blue. Placing a 0 on the *Clear* input will force the flip-flop into the state $Q = 0$. If $Clear = 1$, then this input will have no effect on the NAND gates. Similarly, $Preset = 0$ forces the flip-flop into the state $Q = 1$, while $Preset = 1$ has no effect. To denote that the *Clear* and *Preset* inputs are active when their value is 0, we placed an overbar on the names in the figure. We should note that the circuit that uses this flip-flop should not try to force both *Clear* and *Preset* to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 7.13*b*.



(a) Circuit

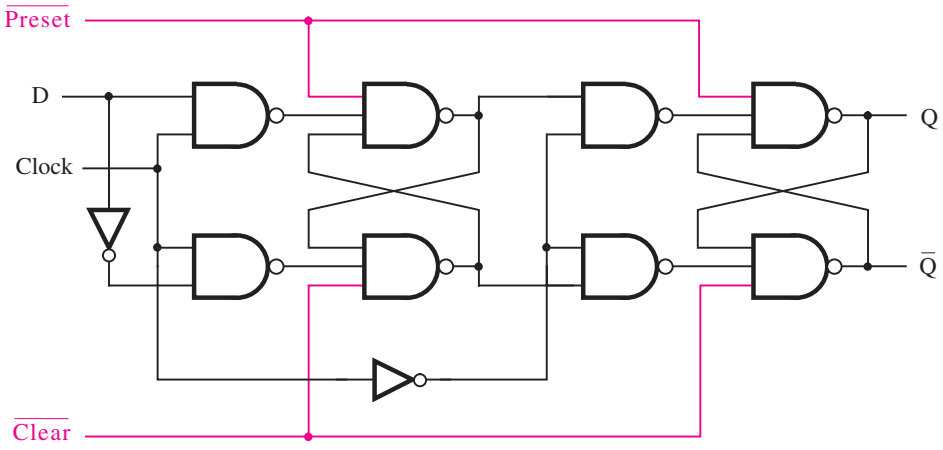


(b) Timing diagram

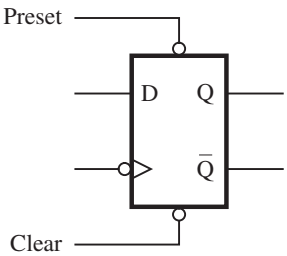
Figure 7.12 Comparison of level-sensitive and edge-triggered D storage elements.

A similar modification can be done on the edge-triggered flip-flop of Figure 7.11*a*, as indicated in Figure 7.14*a*. Again, both *Clear* and *Preset* inputs are active low. They do not disturb the flip-flop when they are equal to 1.

In the circuits in Figures 7.13*a* and 7.14*a*, the effect of a low signal on either the *Clear* or *Preset* input is immediate. For example, if *Clear* = 0 then the flip-flop goes into the state Q = 0 immediately, regardless of the value of the clock signal. In such a circuit, where the *Clear* signal is used to clear a flip-flop without regard to the clock signal, we say that the



(a) Circuit



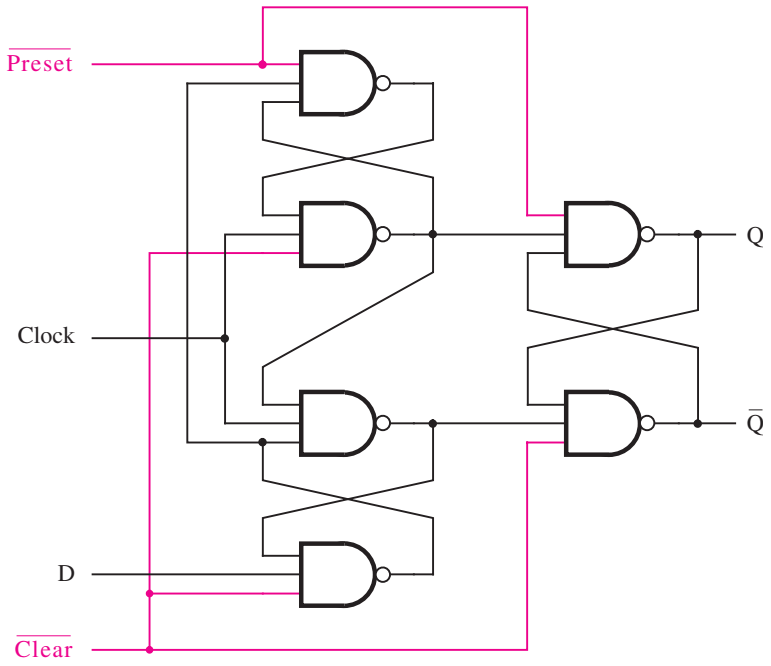
(b) Graphical symbol

Figure 7.13 Master-slave D flip-flop with *Clear* and *Preset*.

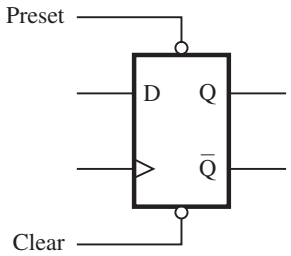
flip-flop has an *asynchronous clear*. In practice, it is often preferable to clear the flip-flops on the active edge of the clock. Such *synchronous clear* can be accomplished as shown in Figure 7.15. The flip-flop operates normally when the *Clear* input is equal to 1. But if *Clear* goes to 0, then on the next positive edge of the clock the flip-flop will be cleared to 0. We will examine the clearing of flip-flops in more detail in section 7.10.

7.5 T FLIP-FLOP

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 7.16*a*. This circuit uses a positive-edge-triggered D flip-flop. The *feedback* connections make the input signal D equal to either the value of Q or $\overline{Q}$ under the control of the signal that is labeled T. On each positive edge of the clock, the flip-flop may change its state $Q(t)$. If



(a) Circuit



(b) Graphical symbol

Figure 7.14 Positive-edge-triggered D flip-flop with *Clear* and *Preset*.

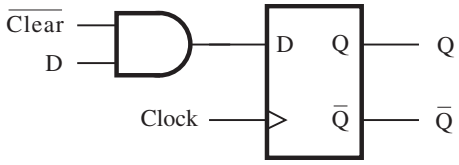
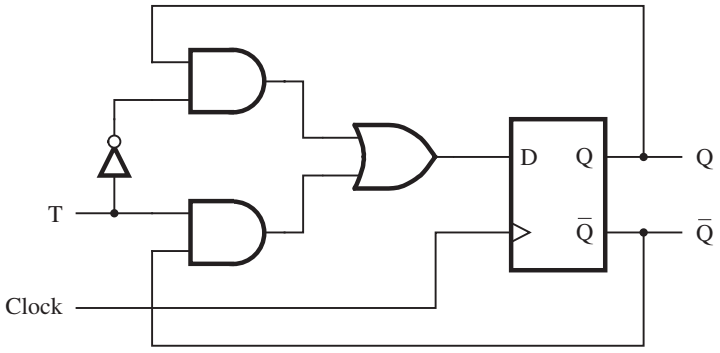


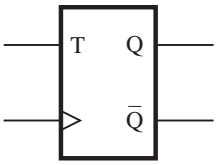
Figure 7.15 Synchronous reset for a D flip-flop.



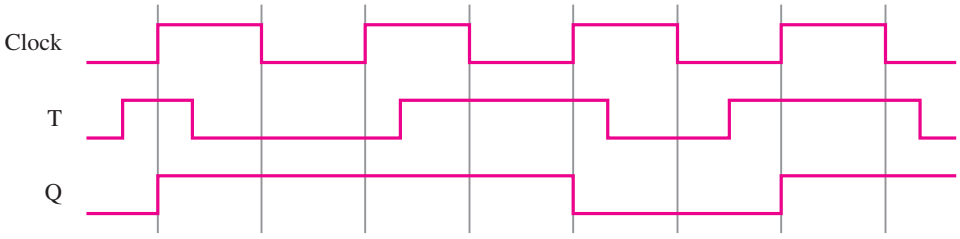
(a) Circuit

T	Q(t + 1)
0	Q(t)
1	$\overline{Q}(t)$

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure 7.16 T flip-flop.

$T = 0$, then $D = Q$ and the state will remain the same, that is, $Q(t + 1) = Q(t)$. But if $T = 1$, then $D = \overline{Q}$ and the new state will be $Q(t + 1) = \overline{Q}(t)$. Therefore, the overall operation of the circuit is that it retains its present state if $T = 0$, and it reverses its present state if $T = 1$.

The operation of the circuit is specified in the form of a truth table in Figure 7.16*b*. Any circuit that implements this truth table is called a *T flip-flop*. The name T flip-flop derives from the behavior of the circuit, which "toggles" its state when $T = 1$. The toggle feature makes the T flip-flop a useful element for building counter circuits, as we will see in section 7.9.

7.5.1 CONFIGURABLE FLIP-FLOPS

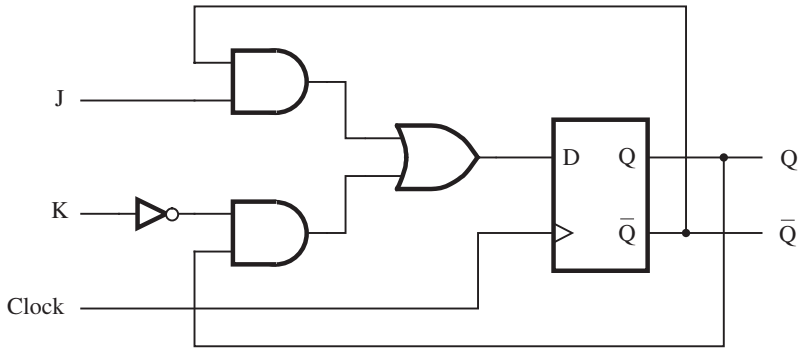
For some circuits one type of flip-flop may lead to a more efficient implementation than a different type of flip-flop. In general purpose chips like PLDs, the flip-flops that are provided are sometimes *configurable*, which means that a flip-flop circuit can be configured to be either D, T, or some other type. For example, in some PLDs the flip-flops can be configured as either D or T types (see problems 7.6 and 7.8).

7.6 JK FLIP-FLOP

Another interesting circuit can be derived from Figure 7.16*a*. Instead of using a single control input, T, we can use two inputs, J and K, as indicated in Figure 7.17*a*. For this circuit the input D is defined as

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

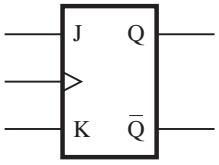
A corresponding truth table is given in Figure 7.17*b*. The circuit is called a *JK flip-flop*. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop,



(a) Circuit

J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

(b) Truth table



(c) Graphical symbol

Figure 7.17 JK flip-flop.

where $J = S$ and $K = R$, for all input values except $J = K = 1$. For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop.

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

7.7 SUMMARY OF TERMINOLOGY

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

Basic latch is a feedback connection of two NOR gates or two NAND gates, which can store one bit of information. It can be set to 1 using the S input and reset to 0 using the R input.

Gated latch is a basic latch that includes input gating and a control input signal. The latch retains its existing state when the control input is equal to 0. Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock. We considered two types of gated latches:

- **Gated SR latch** uses the S and R inputs to set the latch to 1 or reset it to 0, respectively.
- **Gated D latch** uses the D input to force the latch into a state that has the same logic value as the D input.

A **flip-flop** is a storage element based on the gated latch principle, which can have its output state changed only on the edge of the controlling clock signal. We considered two types:

- **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs.
- **Master-slave flip-flop** is built with two gated latches. The master stage is active during half of the clock cycle, and the slave stage is active during the other half. The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage. Master-slave flip-flops can be edge-triggered or level sensitive. If the master stage is a gated D latch, then it behaves as an edge-triggered flip-flop. If the master stage is a gated SR latch, then the flip-flop is level sensitive (see problem 7.19).

7.8 REGISTERS

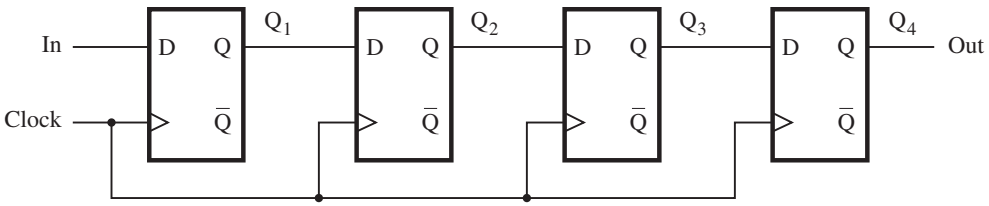
A flip-flop stores one bit of information. When a set of n flip-flops is used to store n bits of information, such as an n-bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register, and each flip-flop operates as described in the

previous sections. The term register is merely a convenience for referring to *n*-bit structures consisting of flip-flops.

7.8.1 SHIFT REGISTER

In section 5.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to the right. A register that provides the ability to shift its contents is called a *shift register*.

Figure 7.18*a* shows a four-bit shift register that is used to shift its contents one bit-position to the right. The data bits are loaded into the shift register in a serial fashion using the *In* input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. An illustration of the transfer is given in Figure 7.18*b*, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 7.18 A simple shift register.

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

7.8.2 PARALLEL-ACCESS SHIFT REGISTER

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

Figure 7.19 shows a four-bit shift register that allows the parallel access. Instead of using the normal shift register connection, the D input of each flip-flop is connected to

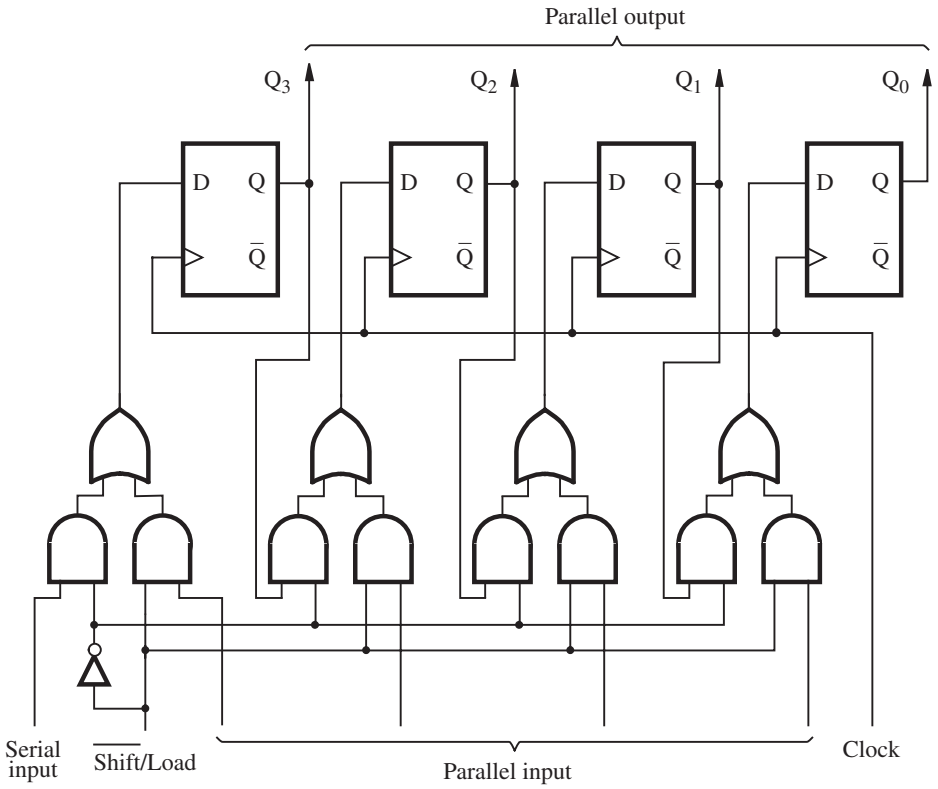


Figure 7.19 Parallel-access shift register.

two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal $\overline{Shift}/Load$ is used to select the mode of operation. If $\overline{Shift}/Load = 0$, then the circuit operates as a shift register. If $\overline{Shift}/Load = 1$, then the parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

In Figure 7.19 we have chosen to label the flip-flops outputs as $Q_3, \ldots, Q_0$ because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of Q_0 during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then accessed in parallel is called a series-to-parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 7.19 can perform both of these functions.

7.9 COUNTERS

In Chapter 5 we dealt with circuits that perform arithmetic operations. We showed how adder/subtractor circuits can be designed, either using a simple cascaded (ripple-carry) structure that is inexpensive but slow or using a more complex carry-lookahead structure that is both more expensive and faster. In this section we examine special types of addition and subtraction operations, which are used for the purpose of counting. In particular, we want to design circuits that can increment or decrement a count by 1. Counter circuits are used in digital systems for many purposes. They may count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, keep track of time elapsed between specific events, and so on.

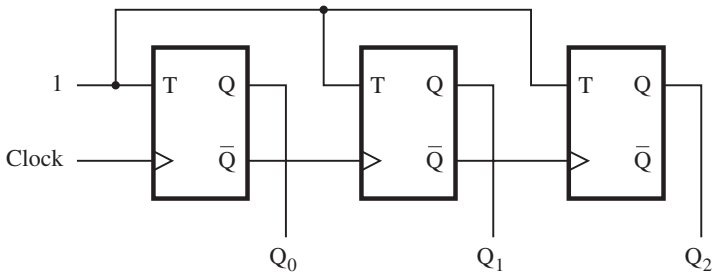
Counters can be implemented using the adder/subtractor circuits discussed in Chapter 5 and the registers discussed in section 7.8. However, since we only need to change the contents of a counter by 1, it is not necessary to use such elaborate circuits. Instead, we can use much simpler circuits that have a significantly lower cost. We will show how the counter circuits can be designed using T and D flip-flops.

7.9.1 ASYNCHRONOUS COUNTERS

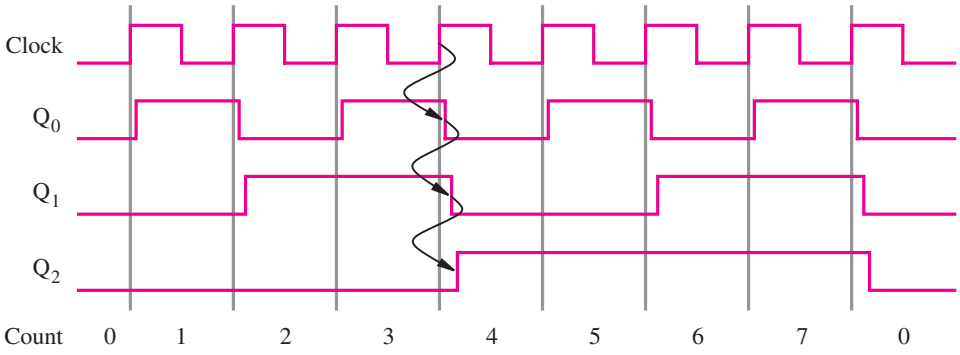
The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation.

Up-Counter with T Flip-Flops

Figure 7.20*a* gives a three-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The T input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called *Clock*. Thus the clock input of the first flip-flop is connected to the *Clock* line. The other two flip-flops have their clock inputs driven by the $\overline{Q}$ output of the preceding flip-flop. Therefore, they toggle their state



(a) Circuit



(b) Timing diagram

Figure 7.20 A three-bit up-counter.

whenever the preceding flip-flop changes its state from Q = 1 to Q = 0, which results in a positive edge of the $\overline{Q}$ signal.

Figure 7.20*b* shows a timing diagram for the counter. The value of Q_0 toggles once each clock cycle. The change takes place shortly after the positive edge of the *Clock* signal. The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by $\overline{Q}_0$, the value of Q_1 changes shortly after the negative edge of the Q_0 signal. Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. If we look at the values $Q_2Q_1Q_0$ as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an *up-counter*.

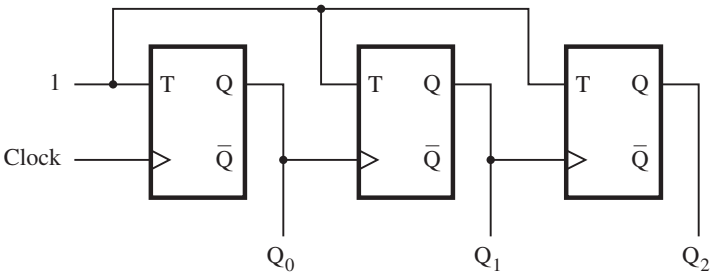
The counter in Figure 7.20*a* has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 7.20*b*, this change requires the toggling of the states of all three flip-flops. The change in Q_0 is observed only after a propagation delay from the positive edge of *Clock*. The Q_1 and Q_2 flip-flops have not yet changed; hence for a brief

time the count is $Q_2Q_1Q_0 = 010$. The change in Q_1 appears after a second propagation delay, at which point the count is 000. Finally, the change in Q_2 occurs after a third delay, at which point the stable state of the circuit is reached and the count is 100. This behavior is similar to the rippling of carries in the ripple-carry adder circuit of Figure 5.6. The circuit in Figure 7.20*a* is an *asynchronous counter*, or a *ripple counter*.

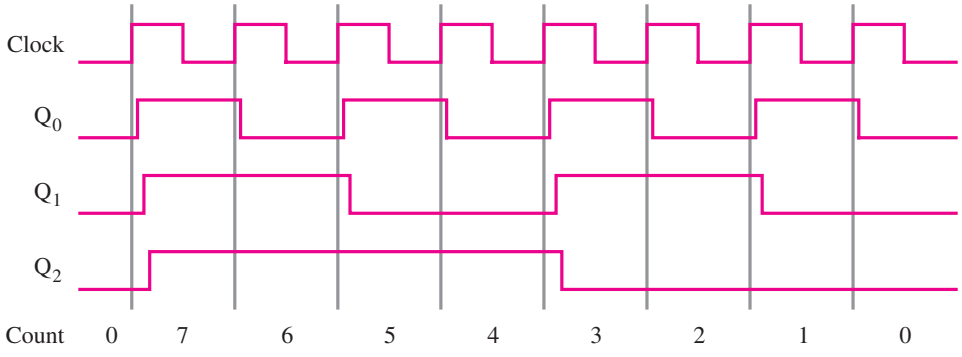
Down-Counter with T Flip-Flops

A slight modification of the circuit in Figure 7.20*a* is presented in Figure 7.21*a*. The only difference is that in Figure 7.21*a* the clock inputs of the second and third flip-flops are driven by the Q outputs of the preceding stages, rather than by the $\overline{Q}$ outputs. The timing diagram, given in Figure 7.21*b*, shows that this circuit counts in the sequence 0, 7, 6, 5, 4, 3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a *down-counter*.

It is possible to combine the functionality of the circuits in Figures 7.20*a* and 7.21*a* to form a counter that can count either up or down. Such a counter is called an *up/down-counter*. We leave the derivation of this counter as an exercise for the reader (problem 7.16).



(a) Circuit



(b) Timing diagram

Figure 7.21 A three-bit down-counter.

7.9.2 SYNCHRONOUS COUNTERS

The asynchronous counters in Figures 7.20*a* and 7.21*a* are simple, but not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the approach described below.

Synchronous Counter with T Flip-Flops

Table 7.1 shows the contents of a three-bit up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of the table, it is apparent that bit Q_0 changes on each clock cycle. Bit Q_1 changes only when $Q_0 = 1$. Bit Q_2 changes only when both Q_1 and Q_0 are equal to 1. In general, for an n-bit up-counter, a given flip-flop changes its state only when all the preceding flip-flops are in the state $Q = 1$. Therefore, if we use T flip-flops to realize the counter, then the T inputs are defined as

$$\begin{aligned} T_0 &= 1 \\ T_1 &= Q_0 \\ T_2 &= Q_0 Q_1 \\ T_3 &= Q_0 Q_1 Q_2 \\ &\vdots \\ T_n &= Q_0 Q_1 \cdots Q_{n-1} \end{aligned}$$

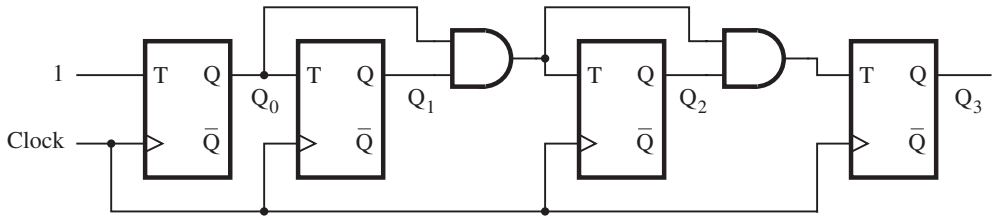
An example of a four-bit counter based on these expressions is given in Figure 7.22*a*. Instead of using AND gates of increased size for each stage, which may lead to fan-in problems, we use a factored arrangement, as shown in the figure. This arrangement does not slow down the response of the counter, because all flip-flops change their states after a

Table 7.1 Derivation of the synchronous up-counter.

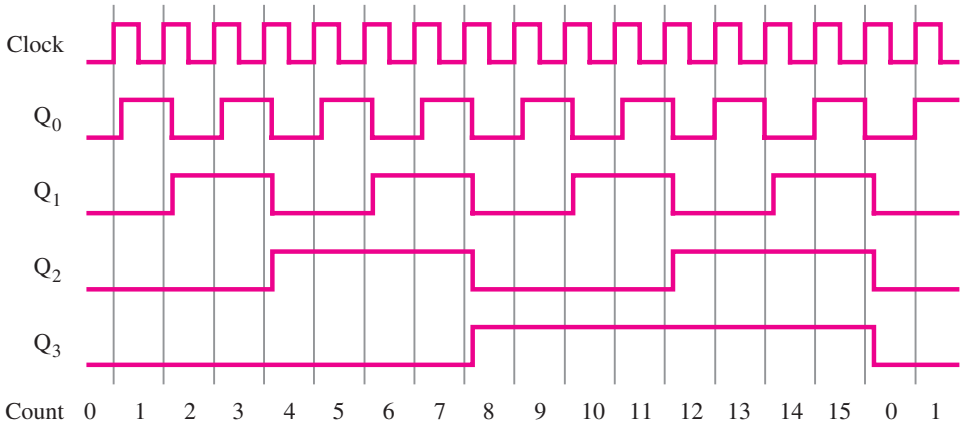
Clock cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

Q_1 changes

Q_2 changes



(a) Circuit



(b) Timing diagram

Figure 7.22 A four-bit synchronous up-counter.

propagation delay from the positive edge of the clock. Note that a change in the value of Q_0 may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops.

Figure 7.22*b* gives a timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Because all changes take place with the same delay after the active edge of the *Clock* signal, the circuit is called a *synchronous counter*.

Enable and Clear Capability

The counters in Figures 7.20 through 7.22 change their contents in response to each clock pulse. Often it is desirable to be able to inhibit counting, so that the count remains in its present state. This may be accomplished by including an *Enable* control signal, as indicated in Figure 7.23. The circuit is the counter of Figure 7.22, where the *Enable* signal controls directly the *T* input of the first flip-flop. Connecting the *Enable* also to the AND-

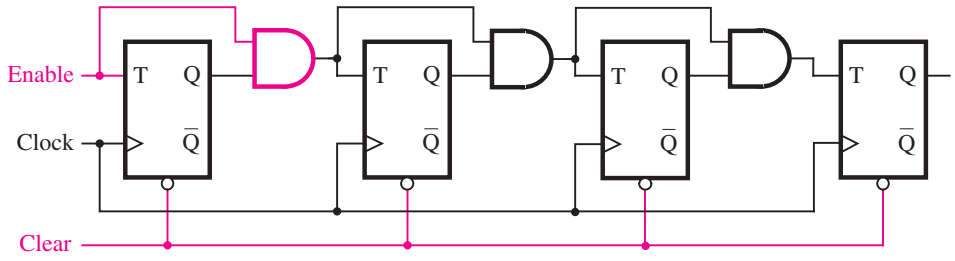


Figure 7.23 Inclusion of Enable and Clear capability.

gate chain means that if *Enable* = 0, then all *T* inputs will be equal to 0. If *Enable* = 1, then the counter operates as explained previously.

In many applications it is necessary to start with the count equal to zero. This is easily achieved if the flip-flops can be cleared, as explained in section 7.4.3. The clear inputs on all flip-flops can be tied together and driven by a *Clear* control input.

Synchronous Counter with D Flip-Flops

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

It is not obvious how D flip-flops can be used to implement a counter. We will present a formal method for deriving such circuits in Chapter 8. Here we will present a circuit structure that meets the requirements but will leave the derivation for Chapter 8. Figure 7.24 gives a four-bit up-counter that counts in the sequence 0, 1, 2, . . . , 14, 15, 0, 1, and so on. The count is indicated by the flip-flop outputs $Q_3Q_2Q_1Q_0$. If we assume that *Enable* = 1, then the *D* inputs of the flip-flops are defined by the expressions

$$D_0 = \overline{Q}_0 = 1 \oplus Q_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1Q_0$$
$$D_3 = Q_3 \oplus Q_2Q_1Q_0$$

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

$$D_i = Q_i \oplus Q_{i-1}Q_{i-2} \cdots Q_1Q_0$$

We will show how to derive these equations in Chapter 8.

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

$$D_0 = Q_0 \oplus Enable$$
$$D_1 = Q_1 \oplus Q_0 \cdot Enable$$

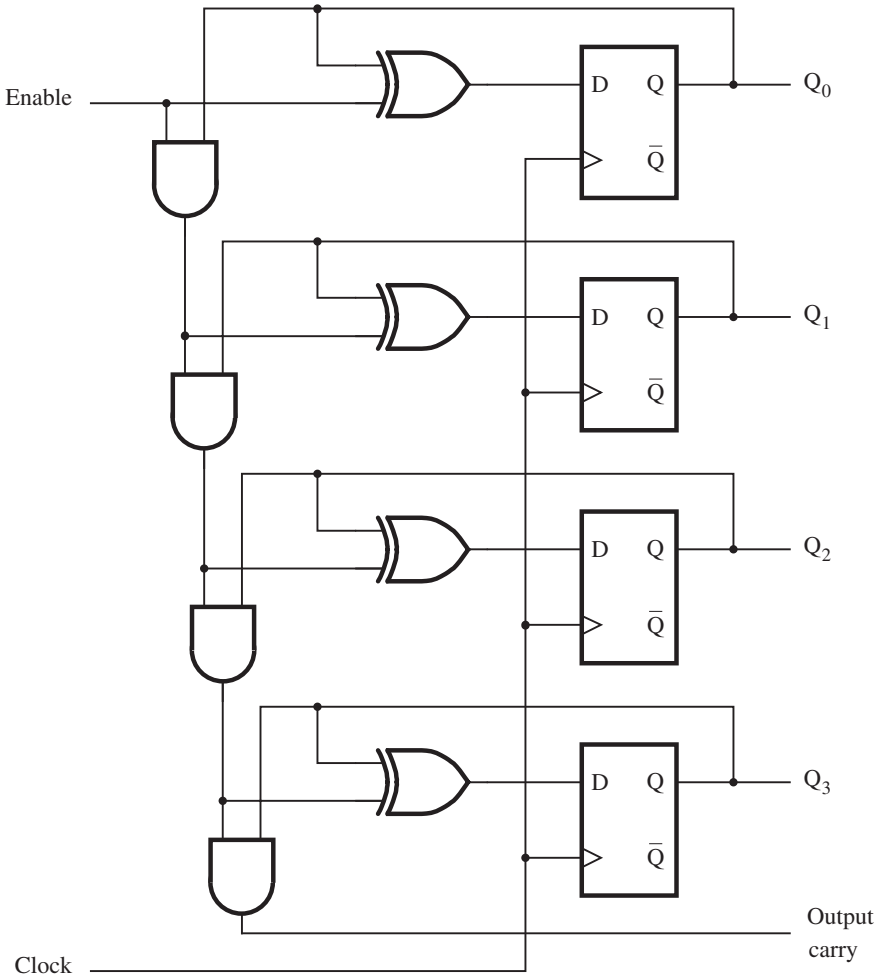


Figure 7.24 A four-bit counter with D flip-flops.

$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable$$
$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable$$

The operation of the counter is based on our observation for Table 7.1 that the state of the flip-flop in stage i changes only if all preceding flip-flops are in the state $Q = 1$. This makes the output of the AND gate that feeds stage i equal to 1, which causes the output of the XOR gate connected to D_i to be equal to $\overline{Q}_i$. Otherwise, the output of the XOR gate provides $D_i = Q_i$, and the flip-flop remains in the same state. This resembles the carry propagation in a carry-lookahead adder circuit (see section 5.4); hence the AND-gate chain can be thought of as the *carry chain*. Even though the circuit is only a four-bit counter, we have included an extra AND that produces the "output carry." This signal makes it easy to concatenate two such four-bit counters to create an eight-bit counter.

Finally, the reader should note that the counter in Figure 7.24 is essentially the same as the circuit in Figure 7.23. We showed in Figure 7.16*a* that a T flip-flop can be formed from a D flip-flop by providing the extra gating that gives

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

Thus in each stage in Figure 7.24, the D flip-flop and the associated XOR gate implement the functionality of a T flip-flop.

7.9.3 COUNTERS WITH PARALLEL LOAD

Often it is necessary to start counting with the initial count being equal to 0. This state can be achieved by using the capability to clear the flip-flops as indicated in Figure 7.23. But sometimes it is desirable to start with a different count. To allow this mode of operation, a counter circuit must have some inputs through which the initial count can be loaded. Using the *Clear* and *Preset* inputs for this purpose is a possibility, but a better approach is discussed below.

The circuit of Figure 7.24 can be modified to provide the parallel-load capability as shown in Figure 7.25. A two-input multiplexer is inserted before each D input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when $Load = 0$. A new initial value, $D_3D_2D_1D_0$, is loaded into the counter when $Load = 1$.

7.10 RESET SYNCHRONIZATION

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

The most straightforward approach is to recognize when the count reaches 5 and then reset the counter. An AND gate can be used to detect the occurrence of the count of 5. Actually, it is sufficient to ascertain that $Q_2 = Q_0 = 1$, which is true only for 5 in our desired counting sequence. A circuit based on this approach is given in Figure 7.26*a*. It uses a three-bit synchronous counter of the type depicted in Figure 7.25. The parallel-load feature of the counter is used to reset its contents when the count reaches 5. The resetting action takes place at the positive clock edge after the count has reached 5. It involves loading $D_2D_1D_0 = 000$ into the flip-flops. As seen in the timing diagram in Figure 7.26*b*,

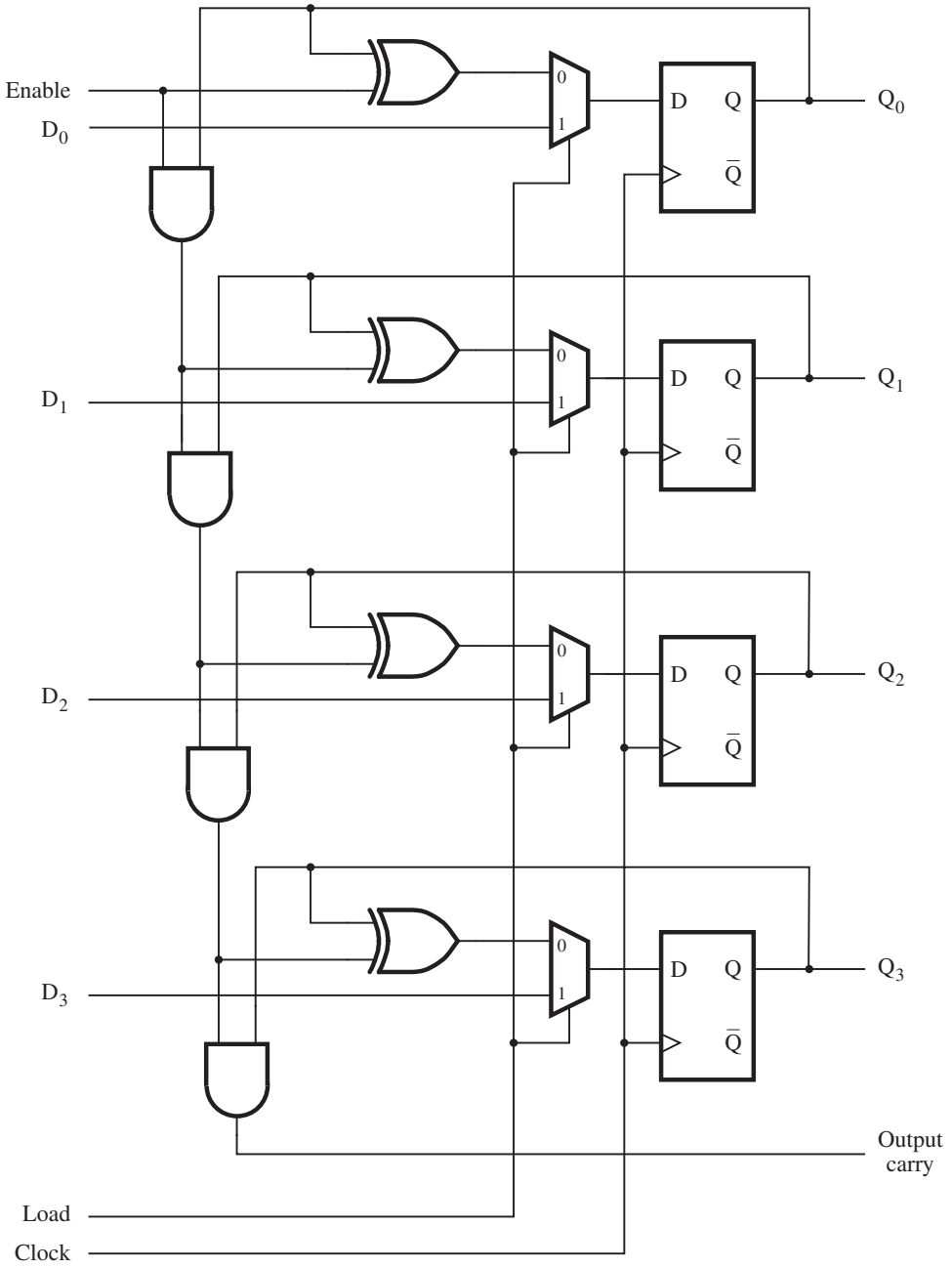
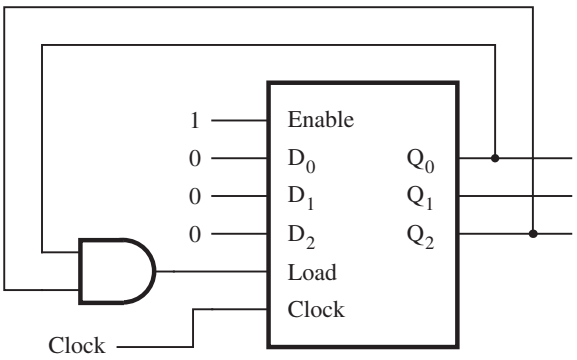
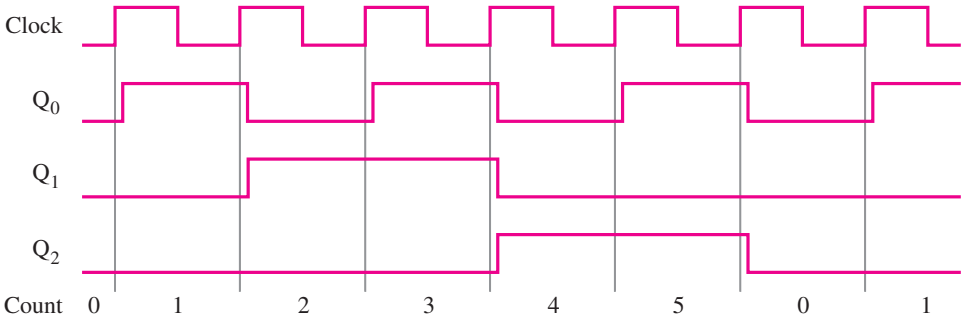


Figure 7.25 A counter with parallel-load capability.



(a) Circuit

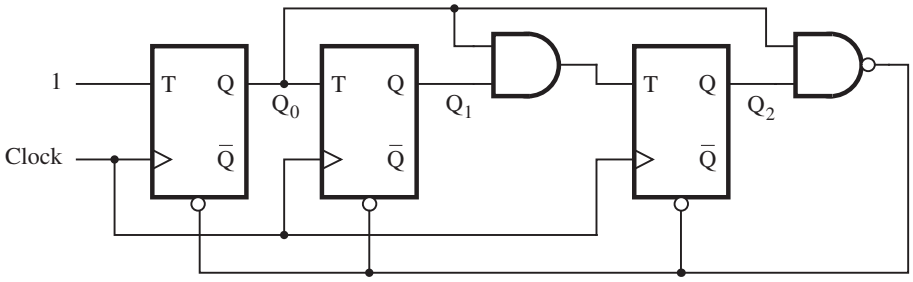


(b) Timing diagram

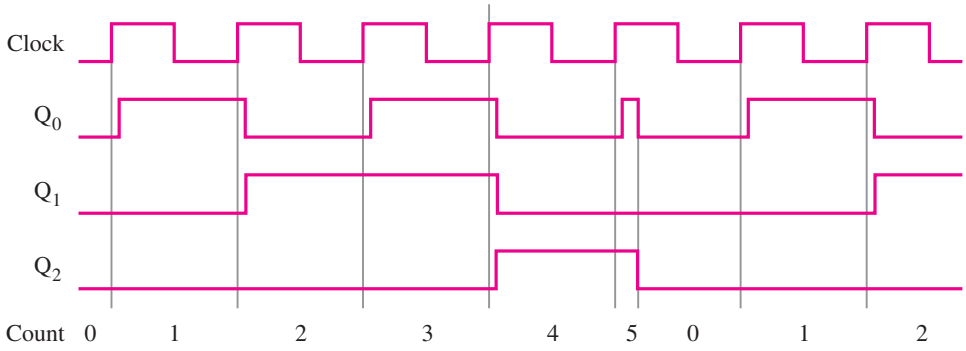
Figure 7.26 A modulo-6 counter with synchronous reset.

the desired counting sequence is achieved, with each value of the count being established for one full clock cycle. Because the counter is reset on the active edge of the clock, we say that this type of counter has a *synchronous reset*.

Consider now the possibility of using the clear feature of individual flip-flops, rather than the parallel-load approach. The circuit in Figure 7.27*a* illustrates one possibility. It uses the counter structure of Figure 7.22*a*. Since the clear inputs are active when low, a NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops. Conceptually, this seems to work fine, but closer examination reveals a potential problem. The timing diagram for this circuit is given in Figure 7.27*b*. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the



(a) Circuit



(b) Timing diagram

Figure 7.27 A modulo-6 counter with asynchronous reset.

circuit, but not on the clock. Therefore, signal values $Q_2Q_1Q_0 = 101$ are maintained for a time that is much less than a clock cycle. Depending on a particular application of such a counter, this may be adequate, but it may also be completely unacceptable. For example, if the counter is used in a digital system where all operations in the system are synchronized by the same clock, then this narrow pulse denoting *Count* = 5 would not be seen by the rest of the system. To solve this problem, we could try to use a modulo-7 counter instead, assuming that the system would ignore the short pulse that denotes the count of 6. This is not a good way of designing circuits, because undesirable pulses often cause unforeseen difficulties in practice. The approach employed in Figure 7.27*a* is said to use *asynchronous reset*.

The timing diagrams in Figures 7.26*b* and 7.27*b* suggest that synchronous reset is a better choice than asynchronous reset. The same observation is true if the natural counting sequence has to be broken by loading some value other than zero. The new value of the count can be established cleanly using the parallel-load feature. The alternative of using the clear and preset capability of individual flip-flops to set their states to reflect the desired count has the same problems as discussed in conjunction with the asynchronous reset.

7.11 OTHER TYPES OF COUNTERS

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

7.11.1 BCD COUNTER

Binary-coded-decimal (BCD) counters can be designed using the approach explained in section 7.10. A two-digit BCD counter is presented in Figure 7.28. It consists of two modulo-10 counters, one for each BCD digit, which we implemented using the parallel-load four-bit counter of Figure 7.25. Note that in a modulo-10 counter it is necessary to reset the four flip-flops after the count of 9 has been obtained. Thus the *Load* input to each stage is equal to 1 when $Q_3 = Q_0 = 1$, which causes 0s to be loaded into the flip-flops at the next positive edge of the clock signal. Whenever the count in stage 0, BCD_0, reaches 9 it is necessary to enable the second stage so that it will be incremented when the next clock

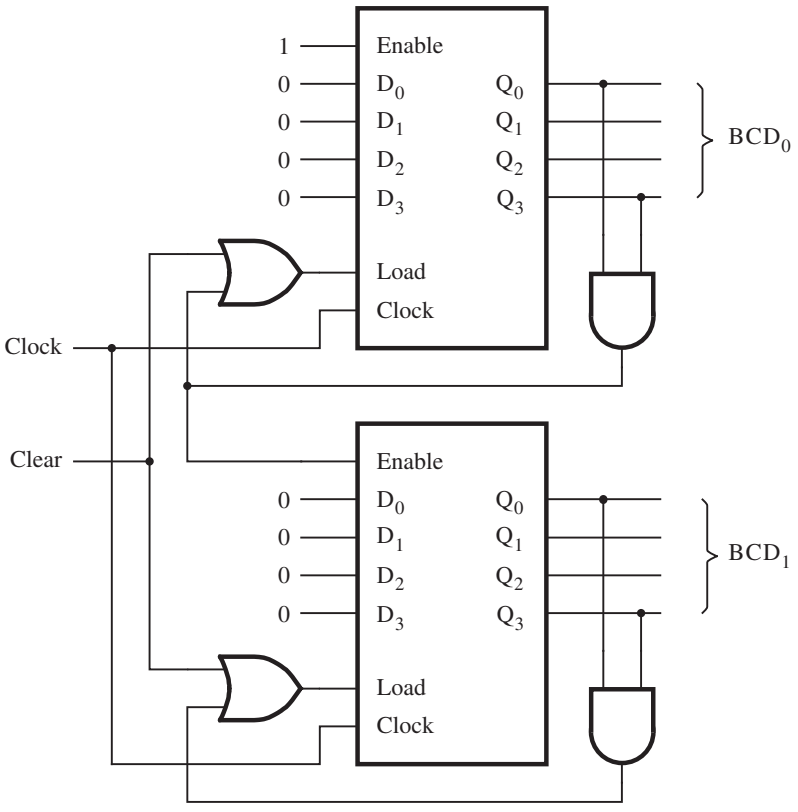


Figure 7.28 A two-digit BCD counter.

pulse arrives. This is accomplished by keeping the *Enable* signal for BCD_1 low at all times except when $BCD_0 = 9$.

In practice, it has to be possible to clear the contents of the counter by activating some control signal. Two OR gates are included in the circuit for this purpose. The control input *Clear* can be used to load 0s into the counter. Observe that in this case *Clear* is active when high. Verilog code for a two-digit BCD counter is given in Figure 7.81.

In any digital system there is usually one or more clock signals used to drive all synchronous circuitry. In the preceding counter, as well as in all counters presented in the previous figures, we have assumed that the objective is to count the number of clock pulses. Of course, these counters can be used to count the number of pulses in any signal that may be used in place of the clock signal.

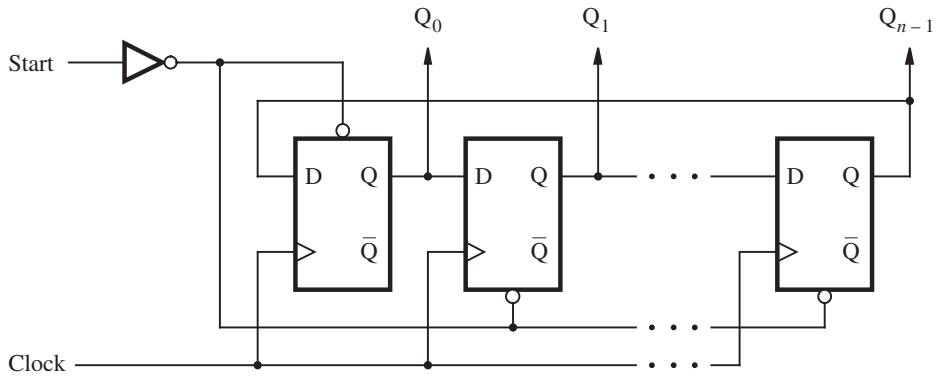
7.11.2 RING COUNTER

In the preceding counters the count is indicated by the state of the flip-flops in the counter. In all cases the count is a binary number. Using such counters, if an action is to be taken as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 7.26 through 7.28.

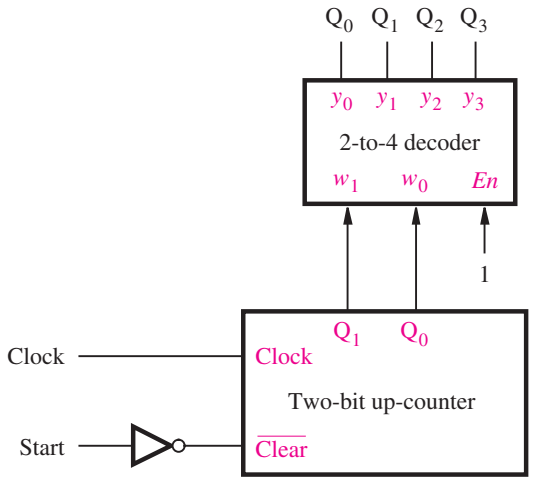
It is possible to devise a counterlike circuit in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, while for all other counts $Q_i = 0$. Then Q_i indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flips-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 7.29*a*. The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ringlike structure. If a single 1 is injected into the ring, this 1 will be shifted through the ring at successive clock cycles. For example, in a four-bit structure, the possible codes $Q_0Q_1Q_2Q_3$ will be 1000, 0100, 0010, and 0001. As we said in section 6.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.

The circuit in Figure 7.29*a* is referred to as a *ring counter*. Its operation has to be initialized by injecting a 1 into the first stage. This is achieved by using the *Start* control signal, which presets the left-most flip-flop to 1 and clears the others to 0. We assume that all changes in the value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.

The circuit in Figure 7.29*a* can be used to build a ring counter with any number of bits, n. For the specific case of $n = 4$, part (*b*) of the figure shows how a ring counter can be constructed using a two-bit up-counter and a decoder. When *Start* is set to 1, the counter is reset to 00. After *Start* changes back to 0, the counter increments its value in the normal way. The 2-to-4 decoder, described in section 6.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$, and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two. We will give an example of a larger circuit that uses the ring counter in Figure 7.29*b* as a subcircuit in section 7.14.



(a) An *n*-bit ring counter



(b) A four-bit ring counter

Figure 7.29 Ring counter.

7.11.3 JOHNSON COUNTER

An interesting variation of the ring counter is obtained if, instead of the Q output, we take the $\overline{Q}$ output of the last stage and feed it back to the first stage, as shown in Figure 7.30. This circuit is known as a *Johnson counter*. An *n*-bit counter of this type generates a counting sequence of length 2*n*. For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on. Note that in this sequence, only a single bit has a different value for two consecutive codes.

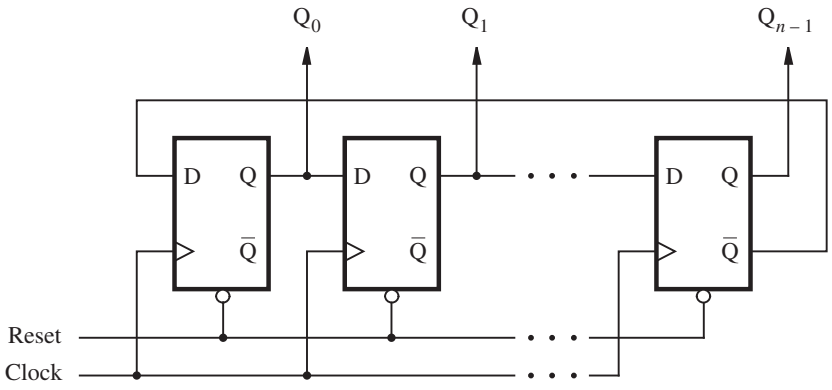


Figure 7.30 Johnson counter.

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

7.11.4 REMARKS ON COUNTER DESIGN

The sequential circuits presented in this chapter, namely, registers and counters, have a regular structure that allows the circuits to be designed using an intuitive approach. In Chapter 8 we will present a more formal approach to design of sequential circuits and show how the circuits presented in this chapter can be derived using this approach.

7.12 USING STORAGE ELEMENTS WITH CAD TOOLS

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

7.12.1 INCLUDING STORAGE ELEMENTS IN SCHEMATICS

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 7.31 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them.

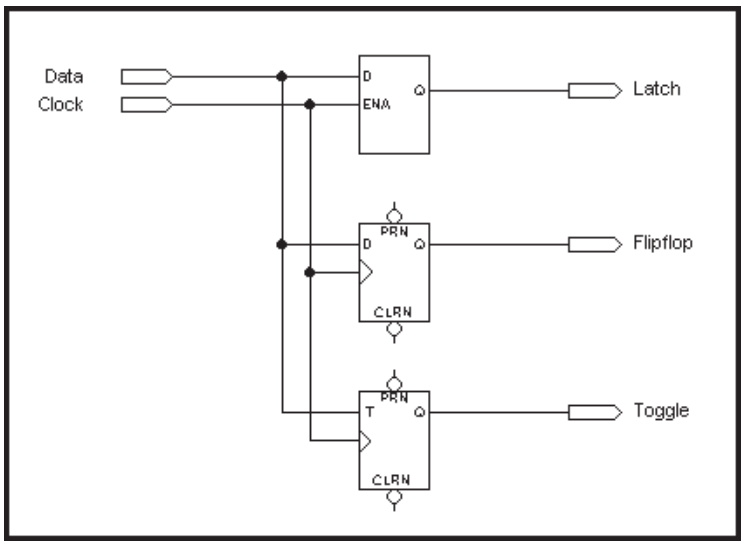


Figure 7.31 Three types of storage elements in a schematic.

When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates shown in section 7.2. In some chips, such as a CPLD, the AND-OR circuit depicted in Figure 7.32 may be preferable. This circuit is functionally equivalent to the cross-coupled version in section 7.2. The sum-of-products circuit is used because it is more suitable for implementation in a CPLD macrocell. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs *Data* and *Latch*. Without this gate, the top AND gate sets the value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a *static hazard*. A detailed explanation of hazards will be given in section 9.6.

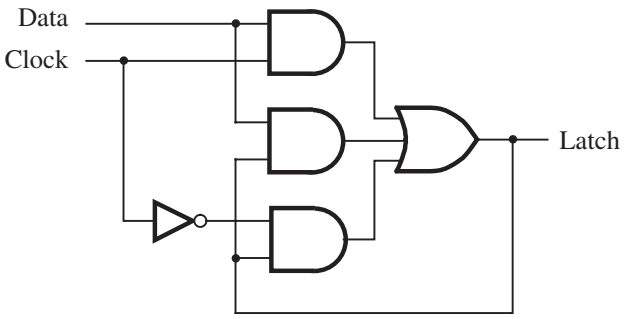


Figure 7.32 Gated D latch generated by CAD tools.

The circuit in Figure 7.31 can be implemented in a CPLD as shown in Figure 7.33. The D and T flip-flops are realized using the flip-flops on the chip that are configurable as either D or T types. The figure depicts in blue the gates and wires needed to implement the circuit in Figure 7.31.

The results of a timing simulation for the implementation in Figure 7.33 are given in Figure 7.34. The *Latch* signal, which is the output of the gated D latch, implemented as indicated in Figure 7.32, follows the *Data* input whenever the *Clock* signal is 1. Because

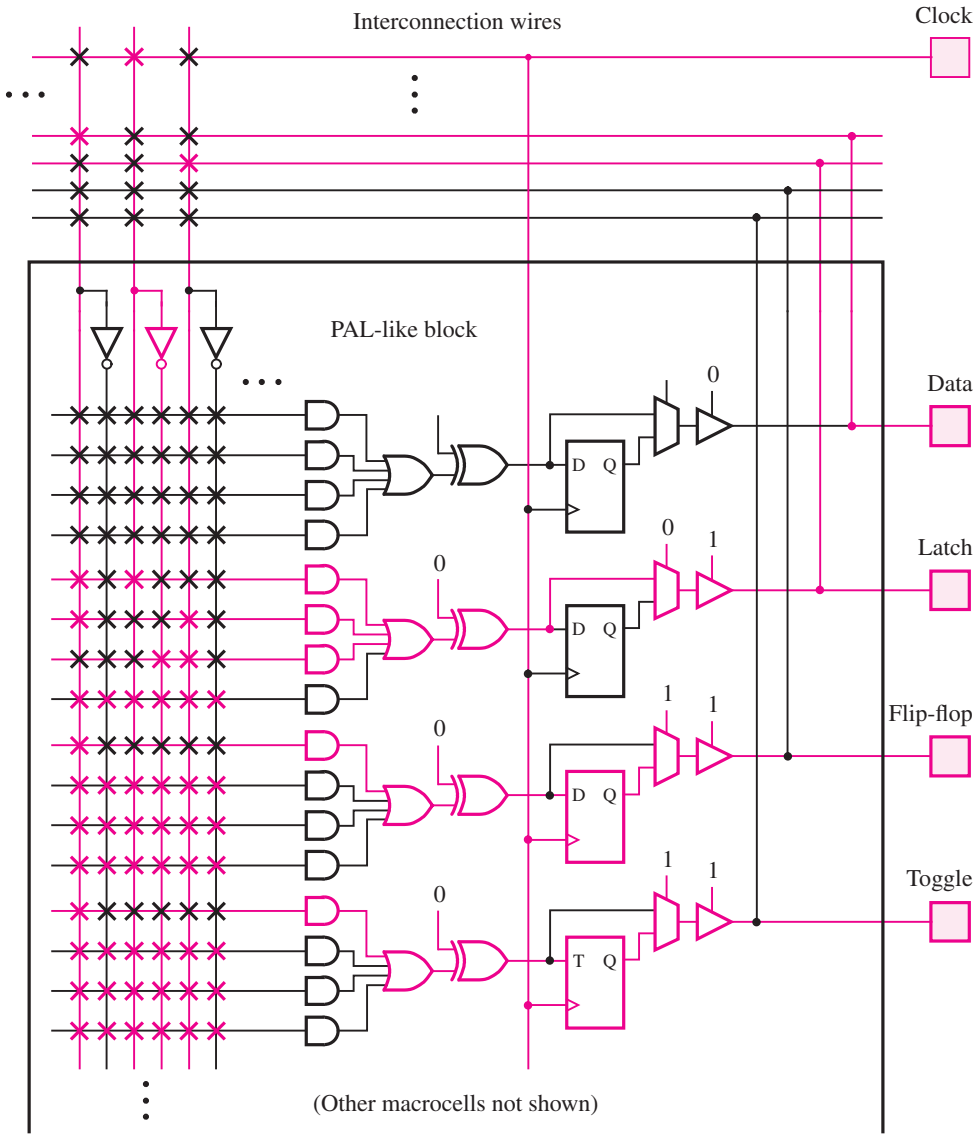


Figure 7.33 Implementation of the schematic in Figure 7.31 in a CPLD.

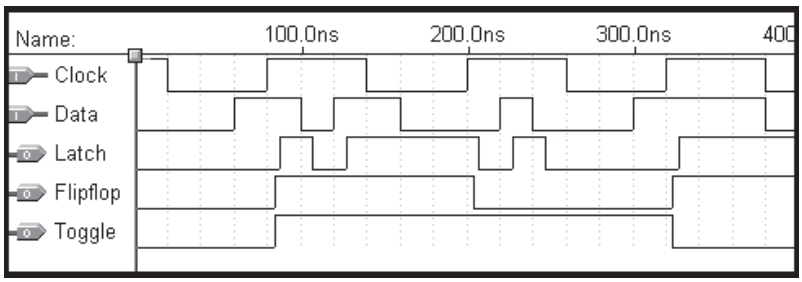


Figure 7.34 Timing simulation for the storage elements in Figure 7.31.

of propagation delays in the chip, the *Latch* signal is delayed in time with respect to the *Data* signal. Since the *Flipflop* signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called *Toggle* in the figure, toggles when *Data* = 1 and a positive clock edge occurs. The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the *clock-to-output time*, t_{co}.

7.12.2 USING VERILOG CONSTRUCTS FOR STORAGE ELEMENTS

In section 6.6 we described a number of Verilog constructs. We now show how these constructs can be used to describe storage elements.

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

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

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

Example 7.1 **CODE FOR A GATED D LATCH** The code in Figure 7.35 defines a module named *D_latch*, which has the inputs *D* and *Clk* and the output Q. The **if** clause defines that the Q output must take the value of *D* when *Clk* = 1. Since no **else** clause is given, a latch will be synthesized to maintain the value of Q when *Clk* = 0. Therefore, the code describes a gated

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

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

endmodule
```

Figure 7.35 Code for a gated D latch.

D latch. The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the Q output.

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

CODE FOR A D FLIP-FLOP **Example 7.2**

Figure 7.36 defines a module named *flipflop*, which is a positive-edge-triggered D flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. The keyword **posedge** specifies that a change may occur only on the positive edge of *Clock*. At this time the output

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

   always @(posedge Clock)
      Q = D;

endmodule
```

Figure 7.36 Code for a D flip-flop.

Q is set to the value of the input *D*. Since Q is of **reg** type it will maintain its value between the positive edges of the clock.

7.12.3 BLOCKING AND NON-BLOCKING ASSIGNMENTS

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

```
f = x1 & x2;
```

or

```
C = A + B;
```

or

```
Q = D;
```

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

Example 7.3 Consider the code in Figure 7.37. Since the **always** block is sensitive to the positive clock edge, both Q1 and Q2 will be implemented as the outputs of D flip-flops. However, because blocking assignments are involved, these two flip-flops will not be connected in cascade, as the reader might expect. The first statement

```
Q1 = D;
```

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

```
module example7_3 (D, Clock, Q1, Q2);
   input D, Clock;
   output Q1, Q2;
   reg Q1, Q2;

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

endmodule
```

Figure 7.37 Incorrect code for two cascaded flip-flops.

```
Q2 = Q1;
```

which results in Q2 = Q1 = D. The synthesized circuit has two parallel flip-flops, as illustrated in Figure 7.38. A synthesis tool will likely delete one of these redundant flip-flops as an optimization step.

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

Example 7.4

Figure 7.39 gives the same code as in Figure 7.37, but using non-blocking assignments. In the two statements

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

the variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block. This code generates a cascaded connection between flip-flops, which implements the shift register depicted in Figure 7.40.

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

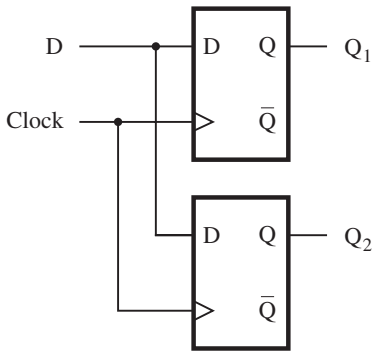


Figure 7.38 Circuit for Example 7.3.

```
module example7_4 (D, Clock, Q1, Q2);
   input D, Clock;
   output Q1, Q2;
   reg Q1, Q2;

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

endmodule
```

Figure 7.39 Code for two cascaded flip-flops.

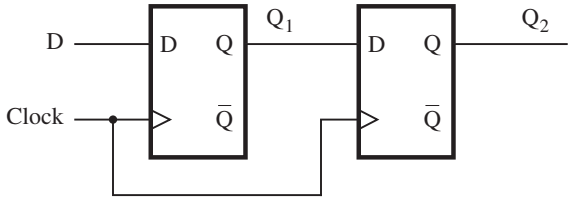


Figure 7.40 Circuit defined in Figure 7.39.

Example 7.5 Code that involves some gates in addition to flip-flops is defined in Figure 7.41 using blocking assignment statements. The resulting circuit is given in Figure 7.42. Both f and g are implemented as the outputs of D flip-flops, because the sensitivity list of the **always** block specifies the event **posedge** Clock. Since blocking assignments are used, the updated value of f generated by the statement f = x1 & x2 has to be seen immediately by the following statement g = f | x3. Thus, the AND gate that produces x1 & x2 is connected to the OR gate that feeds the g flip-flop, as shown in Figure 7.42.

Example 7.6 If non-blocking assignments are used, as given in Figure 7.43, then both f and g are updated simultaneously. Hence, the previous value of f is used in updating the value of g, which means that the output of the flip-flop that generates f is connected to the OR gate that feeds the g flip-flop. This gives rise to the circuit in Figure 7.44.

It is interesting to consider what circuit would be synthesized if the statements that specify f and g were reversed. For the code in Figure 7.41 the impact would be significant. If g is evaluated first, then the second statement does not depend on the first one, because f does not depend on g. The resulting circuit would be the same as the one in Figure 7.44.

```
module example7_5 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output f, g;
   reg f, g;

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

endmodule
```

Figure 7.41 Code for Example 7.5.

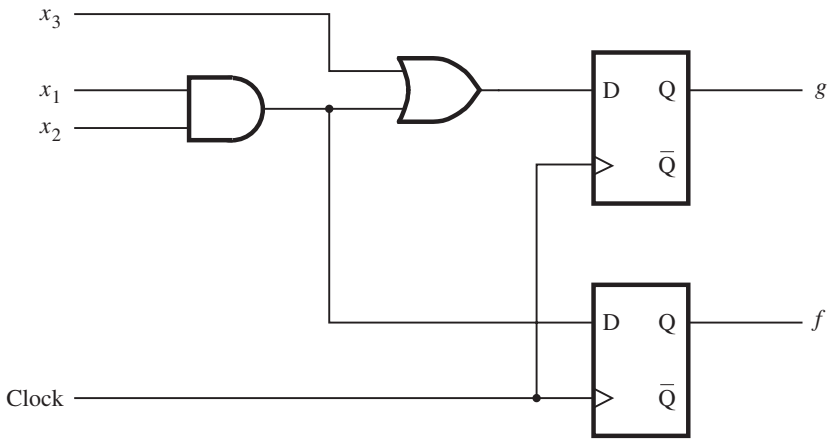


Figure 7.42 Circuit for Example 7.5.

```
module example7_6 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output f, g;
   reg f, g;

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

endmodule
```

Figure 7.43 Code for Example 7.6.

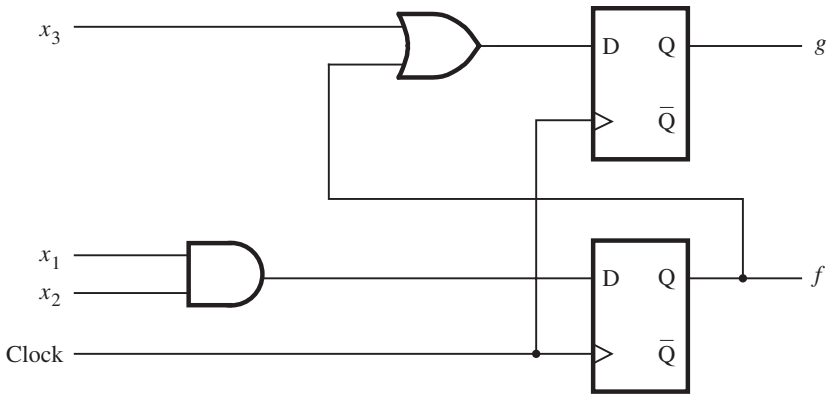


Figure 7.44 Circuit for Example 7.6.

Reversing the statement order would make no difference for the code in Figure 7.43, in which the non-blocking assignment is used.

The use of blocking assignments for sequential circuits can easily lead to wrong results, as demonstrated in Figure 7.38. The dependence on ordering of blocking assignments is dangerous, as shown in the previous example. It is better to use non-blocking assignments to describe sequential circuits.

7.12.4 NON-BLOCKING ASSIGNMENTS FOR COMBINATIONAL CIRCUITS

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

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

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

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

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

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

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

7.12.5 FLIP-FLOPS WITH CLEAR CAPABILITY

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

ASYNCHRONOUS CLEAR **Example 7.7**

Figure 7.45 gives a module that defines a D flip-flop with an asynchronous active-low reset (clear) input. When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0. Note that the sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock. We cannot omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level-sensitive signals.

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

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

endmodule
```

Figure 7.45 D flip-flop with asynchronous reset.

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

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

endmodule
```

Figure 7.46 D flip-flop with synchronous reset.

Example 7.8 **SYNCHRONOUS CLEAR** Figure 7.46 shows how a D flip-flop with a synchronous reset input can be described. In this case the reset signal is acted upon only when a positive clock edge arrives. This code generates the circuit in Figure 7.15, which has an AND gate connected to the flip-flop's D input.

7.13 USING REGISTERS AND COUNTERS WITH CAD TOOLS

In this section we show how registers and counters can be included in circuits designed with the aid of CAD tools. Examples are given using both schematic capture and Verilog code.

7.13.1 INCLUDING REGISTERS AND COUNTERS IN SCHEMATICS

In section 5.5.1 we explained that a CAD system usually includes libraries of prebuilt subcircuits. We introduced the library of parameterized modules (LPM) and used the adder/subtractor module, *lpm_add_sub*, as an example. The LPM includes modules that constitute flip-flops, registers, counters, and many other useful circuits. Figure 7.47 shows a symbol that represents the *lpm_ff* module. This module is a register with one or more positive-edge-triggered flip-flops that can be of either D or T type. The module has parameters that allow the number of flip-flops and flip-flop type to be chosen. In this case we chose to have four D flip-flops. The tutorial in Appendix D explains how the configuration of the module is done.

The D inputs to the four flip-flops, called *data* on the graphical symbol, are connected to the four-bit input signal *Data*[3..0]. The module's asynchronous active-high reset (clear) input, *aclr*, is shown in the schematic. The flip-flop outputs, *q*, are attached to the output symbol labeled Q[3..0].

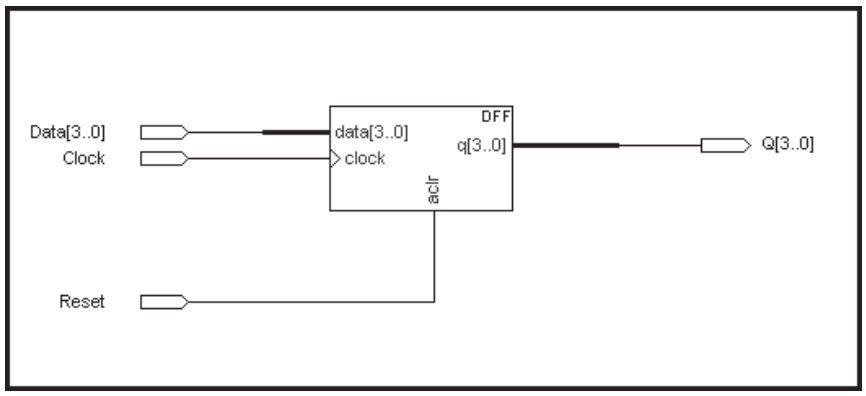


Figure 7.47 The *lpm_ff* parameterized flip-flop module.

In section 7.3 we said that a useful application of D flip-flops is to hold the results of an arithmetic computation, such as the output from an adder circuit. An example is given in Figure 7.48, which uses two LPM modules, *lpm_add_sub* and *lpm_ff*. The *lpm_add_sub* module was described in section 5.5.1. Its parameters, which are not shown in Figure 7.48, are set to configure the module as a four-bit adder circuit. The adder's four-bit data input *dataa* is driven by the *Data*[3..0] input signal. The sum bits, *result*, are connected to the *data* inputs of the *lpm_ff*, which is configured as a four-bit D register with asynchronous clear. The register generates the output of the circuit, Q[3..0], which appears on the left side of the schematic. This signal is fed back to the *datab* input of the adder. The sum bits

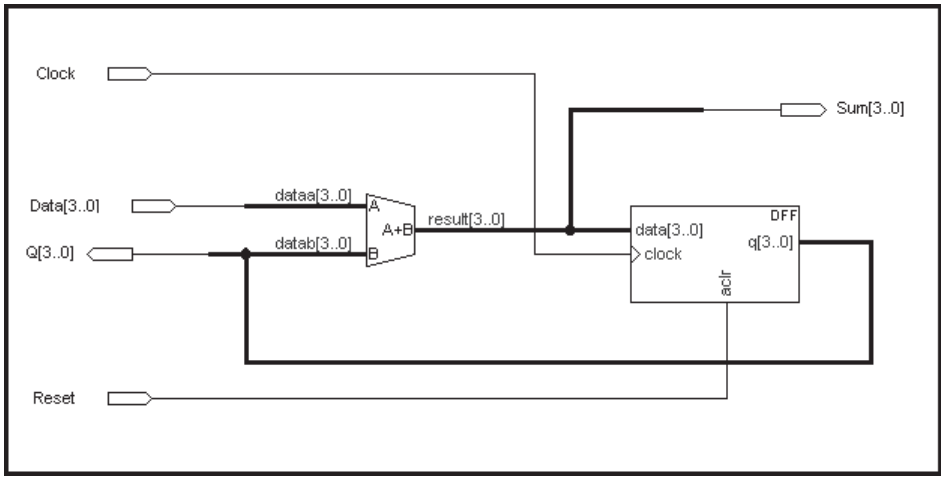


Figure 7.48 An adder with registered feedback.

from the adder are also provided as an output of the circuit, *Sum*[3..0], for ease of reference in the discussion that follows. If the register is first cleared to 0000, then the circuit can be used to add the binary numbers on the *Data*[3..0] input to a sum that is being accumulated in the register, if a new number is applied to the input during each clock cycle. A circuit that performs this function is referred to as an *accumulator* circuit.

We synthesized a circuit from the schematic and implemented the four-bit adder using the carry-lookahead structure. A timing simulation for the circuit appears in Figure 7.49. After resetting the circuit, the *Data* input is set to 0001. The adder produces the sum $0000 + 0001 = 0001$, which is then clocked into the register at the 60 ns point in time. After the t_{co} delay, Q[3..0] becomes 0001, and this causes the adder to produce the new sum $0001 + 0001 = 0010$. The time needed to generate the new sum is determined by the speed of the adder circuit, which produces the sum after 12.5 ns in this case. The new sum does not appear at the Q output until after the next positive clock edge, at 100 ns. The adder then produces 0011 as the next sum. When *Sum* changes from 0010 to 0011, some oscillations appear in the timing diagram, caused by the propagation of carry signals through the adder circuit. These oscillations are not seen at the Q output, because *Sum* is stable by the time the next positive clock edge occurs. Moving forward to the 180 ns point in time, $Sum = 0100$, and this value is clocked into the register. The adder produces the new sum 0101. Then at 200 ns *Data* is changed to 0010, which causes the sum to change to $0100 + 0010 = 0110$. At the next positive clock edge, Q is set to 0110; the value $Sum = 0101$ that was present temporarily in the circuit is not observed at the Q output. The circuit continues to add 0010 to the Q output at each successive positive clock edge.

Having simulated the behavior of the circuit, we should consider whether or not we can conclude with some certainty that the circuit works properly. Ideally, it is prudent to test all possible combinations of a circuit's inputs before declaring that it works as desired. However, in practice, such testing is often not feasible because of the number of input combinations that exist. For the circuit in Figure 7.48, we could verify that a correct sum is produced by the adder, and we could also check that each of the four flip-flops in the register properly stores either 0 or 1. We will discuss issues associated with the testing of circuits in Chapter 11.

For the circuit in Figure 7.48 to work properly, the following timing constraints must be met. When the register is clocked by a positive clock edge, a change of signal value

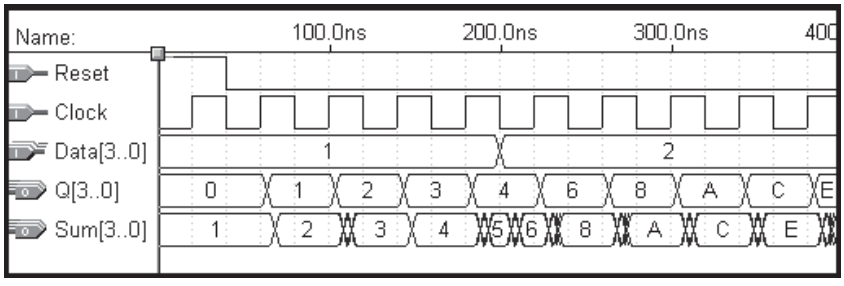


Figure 7.49 Timing simulation of the circuit from Figure 7.48.

at the register's output must propagate through the feedback path to the *datab* input of the adder. The adder then produces a new sum, which must propagate to the *data* input of the register. For the chip used to implement the circuit, the total delay incurred is 14 ns. The delay can be broken down as follows: It takes 2 ns from when the register is clocked until a change in its output reaches the *datab* input of the adder. The adder produces a new sum in 8 ns, and it takes 4 ns for the sum to propagate to the register's *data* input. In Figure 7.49 the clock period is 40 ns. Hence, after the new sum arrives at the *data* input of the register, there remain $40 - 14 = 26$ ns until the next positive clock edge occurs. The *data* input must be stable for the amount of the setup time, $t_{su} = 3$ ns, before the clock edge. Hence we have $26 - 3 = 23$ ns to spare. The clock period can be decreased by as much as 23 ns and the circuit will still work. But if the clock period is less than $40 - 23 = 17$ ns, then the circuit will not function properly. Of course, if a different chip were used to implement the circuit, then different timing results would be produced. CAD systems provide tools that can automatically determine the minimum allowable clock period for which a circuit will work correctly. The tutorial in Appendix D shows how this is done using the tools that accompany the book.

7.13.2 USING LIBRARY MODULES IN VERILOG CODE

The predefined subcircuits in a library of modules such as the LPM library can be instantiated in Verilog code. Figure 7.50 instantiates the *lpm_shiftreg* module, which is an *n*-bit shift register. The module's parameters are set using **defparam** statements. The number of flip-flops in the shift register is set to 4 using the parameter lpm_width = 4. The module can be configured to shift either left or right. The parameter lpm_direction = "RIGHT" sets the shift direction to be from the left to the right. The code uses the module's asynchronous active-high clear input, *aclr*, and the active-high parallel-load input, *load*, which allows the shift register to be loaded with the parallel data on the module's *data* input. When shifting takes place, the value on the *shiftin* input is shifted into the left-most flip-flop and the bit shifted out appears on the right-most bit of the *q* parallel output. The code uses *named*

```
module  shift (Clock, Reset, w, Load, R, Q);
   input  Clock, Reset, w, Load;
   input  [3:0] R;
   output  [3:0] Q ;

   lpm_shiftreg  shift_right (.data(R), .aclr(Reset), .clock(Clock),
      .load(Load), .shiftin(w), .q(Q)) ;
      defparam shift_right.lpm_width = 4;
      defparam shift_right.lpm_direction = "RIGHT";

endmodule
```

Figure 7.50 Instantiation of the *lpm_shiftreg* module.

ports to connect the input and output signals of the *shift* module to the ports of the module. For example, the *R* input signal is connected to the module's *data* port. This is specified by writing .data(R) in the instantiation statement. Similarly, .aclr(Reset) specifies that the *Reset* input signal is connected to the *aclr* port on the module, and so on. When translated into a circuit, the *lpm_shiftreg* has the structure shown in Figure 7.19.

Predefined modules also exist for the various types of counters, which are commonly needed in logic circuits. An example is the *lpm_counter* module, which is a variable-width counter with parallel-load inputs.

7.13.3 USING VERILOG CONSTRUCTS FOR REGISTERS AND COUNTERS

Rather than instantiating predefined subcircuits for registers, shift registers, counters, and the like, the circuits can be described in Verilog code. Figure 7.45 gives code for a D flip-flop. One way to describe an *n*-bit register is to write hierarchical code that includes *n* instances of the D flip-flop subcircuit. A simpler approach is to use the same code as in Figure 7.45 and define the *D* input and Q output as multibit signals.

Example 7.9 **AN N-BIT REGISTER** Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed. The code for an *n*-bit register is given in Figure 7.51. The parameter *n* specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

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

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

endmodule
```

Figure 7.51 Code for an *n*-bit register with asynchronous clear.

A FOUR-BIT SHIFT REGISTER Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Figure 7.19. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the *D* input. Figure 7.52 defines the module named *muxdff*, which represents this subcircuit. The two data inputs are named D_0 and D_1, and they are selected using the *Sel* input. The **if-else** statement specifies that on the positive clock edge if *Sel* = 0, then Q is assigned the value of D_0; otherwise, Q is assigned the value of D_1.

Example 7.10

Figure 7.53 defines the four-bit shift register. The module *Stage3* instantiates the left-most flip-flop, which has the output Q_3, and the module *Stage0* instantiates the right-most flip-flop, Q_0. When $L = 1$, the register is loaded in parallel from the *R* input; and when $L = 0$, shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit, Q_3, from the *w* input.

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

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

endmodule
```

Figure 7.52 Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

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

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

endmodule
```

Figure 7.53 Hierarchical code for a four-bit shift register.

Example 7.11 **ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER** A different style of code for the four-bit shift register is given in Figure 7.54. Instead of using subcircuits, the shift register is defined using the approach presented in Example 7.4. All actions take place at the positive edge of the clock. If $L = 1$, the register is loaded in parallel with the four bits of input R. If $L = 0$, the contents of the register are shifted to the right and the value of the input w is loaded into the most-significant bit Q_3.

Example 7.12 **AN N-BIT SHIFT REGISTER** Figure 7.55 shows the code that can be used to represent shift registers of any size. The parameter n, which has the default value 16 in the figure, sets the number of flip-flops. The code is identical to that in Figure 7.54 with two exceptions. First, R and Q are defined in terms of n. Second, the **else** clause that describes the shifting operation is generalized to work for any number of flip-flops by using a **for** loop.

Example 7.13 **UP-COUNTER** Figure 7.56 represents a four-bit up-counter with a reset input, *Resetn*, and an enable input, E. The outputs of the flip-flops in the counter are represented by the vector named Q. The **if** statement specifies an asynchronous reset of the counter if $Resetn = 0$. The **else if** clause specifies that if $E = 1$ the count is incremented on the positive clock edge.

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

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

endmodule
```

Figure 7.54 Alternative code for a four-bit shift register.

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

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

endmodule
```

Figure 7.55 An n-bit shift register.

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

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

endmodule
```

Figure 7.56 Code for a four-bit up-counter.

UP-COUNTER WITH PARALLEL LOAD The code in Figure 7.57 defines an up-counter that has a parallel-load input in addition to a reset input. The parallel data is provided as the input vector R. The first **if** statement provides the same asynchronous reset as in Figure 7.56. The **else if** clause specifies that if $L = 1$ the flip-flops in the counter are loaded in **Example 7.14**

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

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

endmodule
```

Figure 7.57 A four-bit up-counter with a parallel load.

parallel from the R inputs on the positive clock edge. If $L = 0$, the count is incremented, under control of the enable input E.

Example 7.15 DOWN-COUNTER WITH PARALLEL LOAD Figure 7.58 shows the code for a down-counter named *downcount*. A down-counter is normally used by loading it with some starting count and then decrementing its contents. The starting count is represented in the code by the vector R. On the positive clock edge, if $L = 1$ the counter is loaded with the input R, and if $L = 0$ the count is decremented. The counter also includes an enable input, E. Setting

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

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

endmodule
```

Figure 7.58 A down-counter with a parallel load.

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

   always @(posedge Clock)
   begin
      if (up_down)
         direction = 1;
      else
         direction = −1;
      if (L)
         Q <= R;
      else if (E)
         Q <= Q + direction;
   end

endmodule
```

Figure 7.59 Code for an up/down counter.

$E = 0$ prevents the contents of the flip-flops from changing when an active clock edge occurs.

UP/DOWN COUNTER **Example 7.16**

Verilog code for an up/down counter is given in Figure 7.59. This module combines the capabilities of the counters defined in Figures 7.57 and 7.58. It includes a control signal *up_down* that governs the direction of counting. It also includes an **integer** variable named *direction*, which is equal to 1 for up-count and equal to −1 for down-count.

7.14 DESIGN EXAMPLES

This section presents examples of digital systems that make use of some of the building blocks described in this chapter and in Chapter 6.

7.14.1 BUS STRUCTURE

Digital systems often contain a set of registers used to store data. Figure 7.60 gives an example of a system that has k n-bit registers, $R1$ to Rk. Each register is connected to a

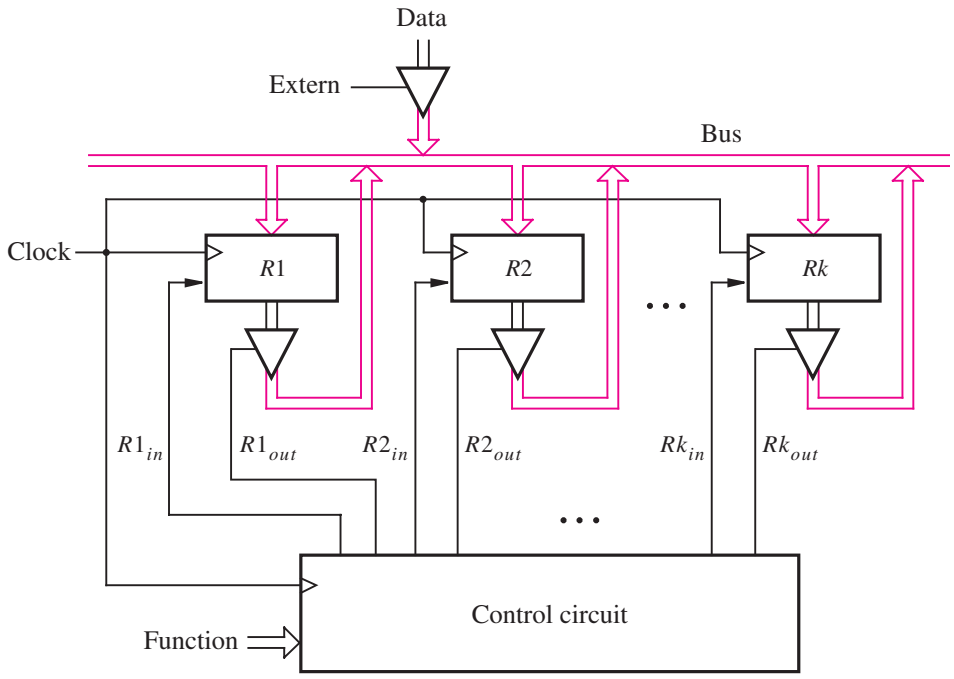


Figure 7.60 A digital system with k registers.

common set of n wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a *bus*. In addition to registers, in a real system other types of circuit blocks would be connected to the bus. The figure shows how n bits of data can be placed on the bus from another circuit block, using the control input *Extern*. The data stored in any of the registers can be transferred via the bus to a different register or to another circuit block that is connected to the bus.

It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. In Figure 7.60 each register is connected to the bus through an n-bit tri-state buffer. A control circuit is used to ensure that only one of the tri-state buffer enable inputs, $R1_{out}, \ldots, Rk_{out}$, is asserted at a given time. The control circuit also produces the signals $R1_{in}, \ldots, Rk_{in}$, which control when data is loaded into each register. In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and the like. Figure 7.60 shows an input signal named *Function* that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the k registers.

Figure 7.61 provides a more detailed view of how the registers from Figure 7.60 can be connected to a bus. To keep the picture simple, 2 two-bit registers are shown, but the same scheme can be used for larger registers. For register $R1$, two tri-state buffers enabled by $R1_{out}$ are used to connect each flip-flop output to a wire in the bus. The D input on

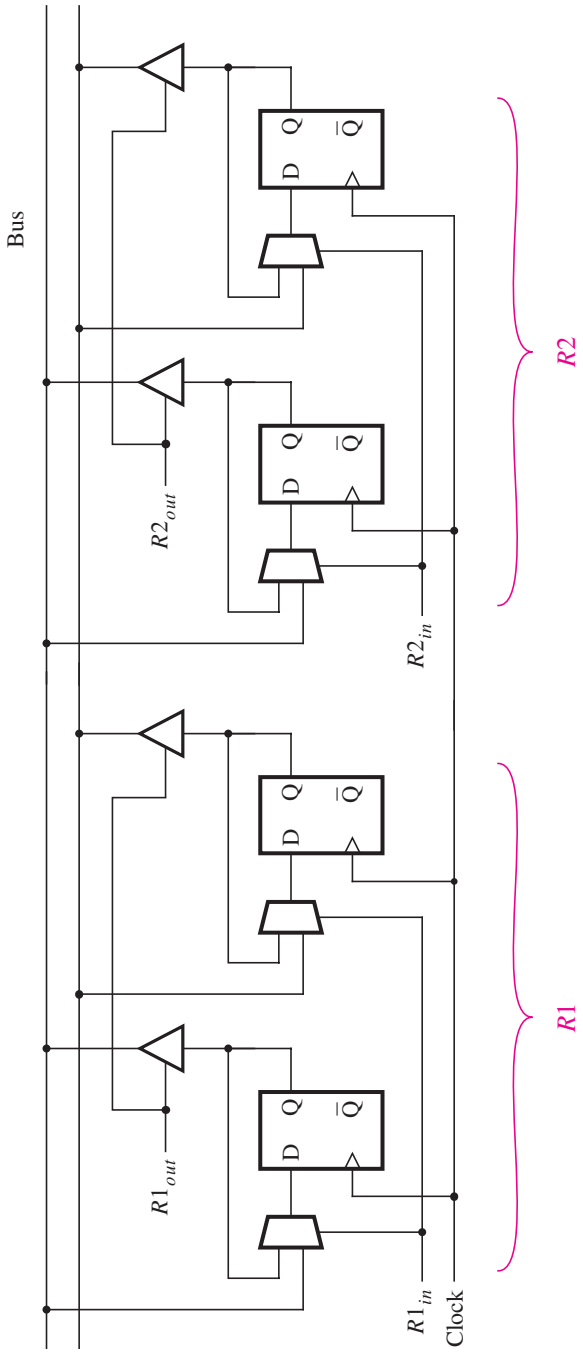


Figure 7.61 Details for connecting registers to a bus.

each flip-flop is connected to a 2-to-1 multiplexer, whose select input is controlled by $R1_{in}$. If $R1_{in} = 0$, the flip-flops are loaded from their Q outputs; hence the stored data does not change. But if $R1_{in} = 1$, data is loaded into the flip-flops from the bus. Instead of using multiplexers on the flip-flop inputs, one could attempt to connect the D inputs on the flip-flops directly to the bus. Then it is necessary to control the clock inputs on all flip-flops to ensure that they are clocked only when new data should be loaded into the register. This approach is not good because it may happen that different flip-flops will be clocked at slightly different times, leading to a problem known as *clock skew*. A detailed discussion of the issues related to the clocking of flip-flops is provided in section 10.3.

The system in Figure 7.60 can be used in many different ways, depending on the design of the control circuit and on how many registers and other circuit blocks are connected to the bus. As a simple example, consider a system that has three registers, $R1$, $R2$, and $R3$. Each register is connected to the bus as indicated in Figure 7.61. We will design a control circuit that performs a single function—it swaps the contents of registers $R1$ and $R2$, using $R3$ for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of $R2$ are transferred into $R3$. Then the contents of $R1$ are transferred into $R2$. Finally, the contents of $R3$, which are the original contents of $R2$, are transferred into $R1$. Note that we say that the contents of one register, R_i, are "transferred" into another register, R_j. This jargon is commonly used to indicate that the new contents of R_j will be a copy of the contents of R_i. The contents of R_i are not changed as a result of the transfer. Therefore, it would be more precise to say that the contents of R_i are "copied" into R_j.

Using a Shift Register for Control

There are many ways to design a suitable control circuit for the swap operation. One possibility is to use the left-to-right shift register shown in Figure 7.62. Assume that the reset input is used to clear the flip-flops to 0. Hence the control signals $R1_{in}$, $R1_{out}$, and so on are not asserted, because the shift register outputs have the value 0. The serial input w normally has the value 0. We assume that changes in the value of w are synchronized to occur shortly after the active clock edge. This assumption is reasonable because w would normally be generated as the output of some circuit that is controlled by the same clock signal. When the desired swap should be performed, w is set to 1 for one clock cycle, and then w returns to 0. After the next active clock edge, the output of the left-most flip-flop

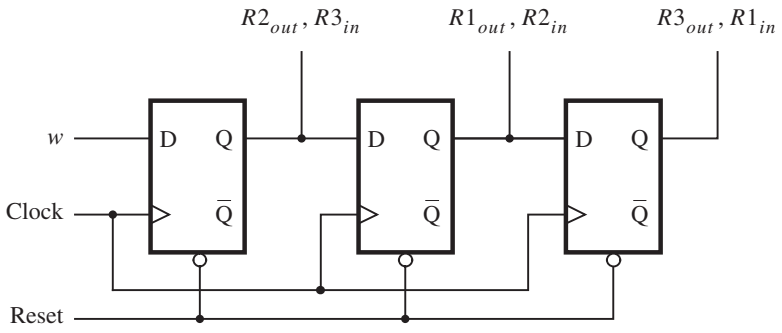


Figure 7.62 A shift-register control circuit.

becomes equal to 1, which asserts both $R2_{out}$ and $R3_{in}$. The contents of register $R2$ are placed onto the bus wires and are loaded into register $R3$ on the next active clock edge. This clock edge also shifts the contents of the shift register, resulting in $R1_{out} = R2_{in} = 1$. Note that since w is now 0, the first flip-flop is cleared, causing $R2_{out} = R3_{in} = 0$. The contents of $R1$ are now on the bus and are loaded into $R2$ on the next clock edge. After this clock edge the shift register contains 001 and thus asserts $R3_{out}$ and $R1_{in}$. The contents of $R3$ are now on the bus and are loaded into $R1$ on the next clock edge.

Using the control circuit in Figure 7.62, when w changes to 1 the swap operation does not begin until after the next active clock edge. We can modify the control circuit so that it starts the swap operation in the same clock cycle in which w changes to 1. One possible approach is illustrated in Figure 7.63. The reset signal is used to set the shift-register contents to 100, by presetting the left-most flip-flop to 1 and clearing the other two flip-flops. As long as $w = 0$, the output control signals are not asserted. When w changes to 1, the signals $R2_{out}$ and $R3_{in}$ are immediately asserted and the contents of $R2$ are placed onto the bus. The next active clock edge loads this data into $R3$ and also shifts the shift register contents to 010. Since the signal $R1_{out}$ is now asserted, the contents of $R1$ appear on the bus. The next clock edge loads this data into $R2$ and changes the shift register contents to 001. The contents of $R3$ are now on the bus; this data is loaded into $R1$ at the next clock edge, which also changes the shift register contents to 100. We assume that w had the value 1 for only one clock cycle; hence the output control signals are not asserted at this point. It may not be obvious to the reader how to design a circuit such as the one in Figure 7.63, because we have presented the design in an ad hoc fashion. In section 8.3 we will show how this circuit can be designed using a more formal approach.

The circuit in Figure 7.63 assumes that a preset input is available on the left-most flip-flop. If the flip-flop has only a clear input, then we can use the equivalent circuit shown in Figure 7.64. In this circuit we use the $\overline{Q}$ output of the left-most flip-flop and also complement the input to this flip-flop by using a NOR gate instead of an OR gate.

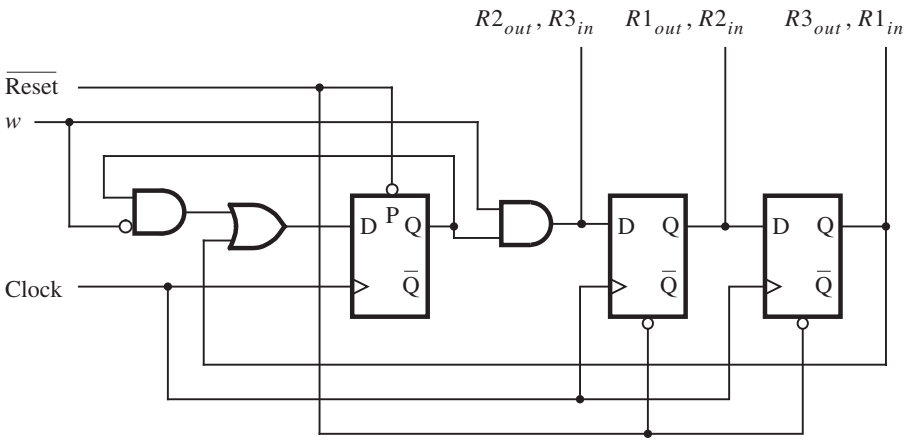


Figure 7.63 A modified control circuit.

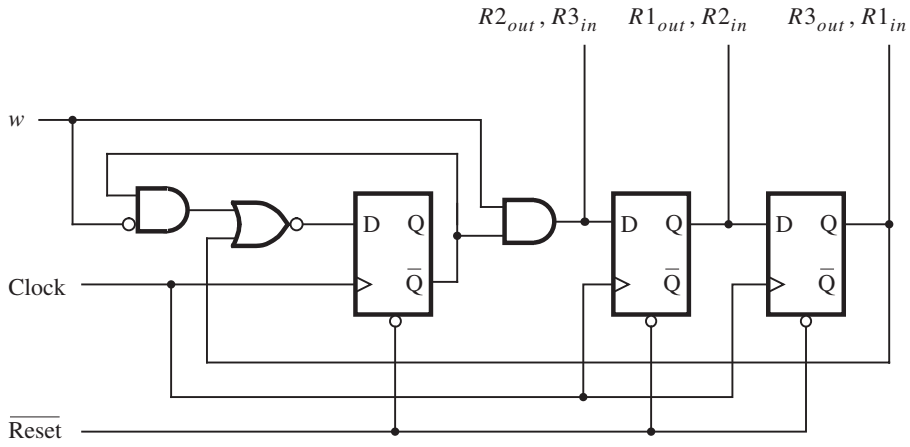


Figure 7.64 A modified version of the circuit in Figure 7.63.

Using Multiplexers to Implement a Bus

In Figure 7.60 we used tri-state buffers to control access to the bus. An alternative approach is to use multiplexers, as depicted in Figure 7.65. The outputs of each register are connected to a multiplexer. This multiplexer's output is connected to the inputs of the registers, thus realizing the bus. The multiplexer select inputs determine which register's contents appear on the bus. Although the figure shows just one multiplexer symbol, we actually need one multiplexer for each bit in the registers. For example, assume that there are 4 eight-bit registers, $R1$ to $R4$, plus the externally-supplied eight-bit *Data*. To

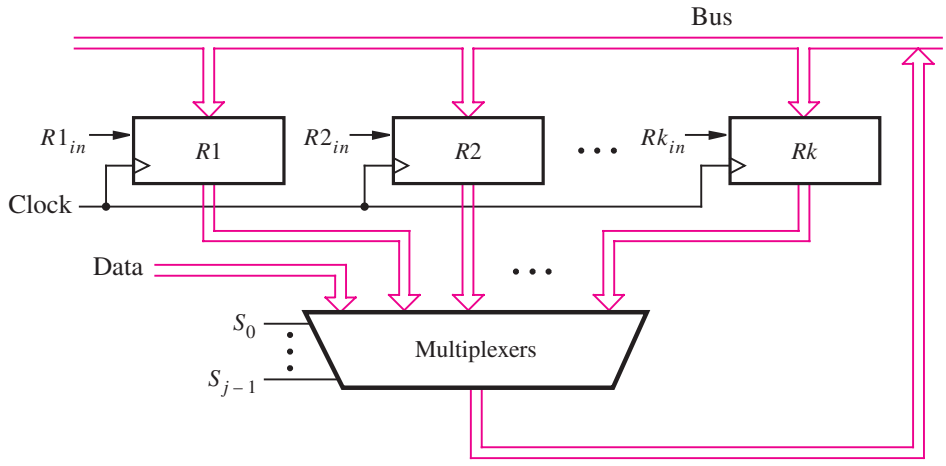


Figure 7.65 Using multiplexers to implement a bus.

interconnect them, we need eight 5-to-1 multiplexers. In Figure 7.62 we used a shift register to implement the control circuit. A similar approach can be used with multiplexers. The signals that control when data is loaded into a register, like $R1_{in}$, can still be connected directly to the shift-register outputs. However, instead of using control signals like $R1_{out}$ to place the contents of a register onto the bus, we have to generate the select inputs for the multiplexers. One way to do so is to connect the shift-register outputs to an encoder circuit that produces the select inputs for the multiplexer. We discussed encoder circuits in section 6.3.

The tri-state buffer and multiplexer approaches for implementing a bus are both equally valid. However, some types of chips, such as most PLDs, do not contain a sufficient number of tri-state buffers to realize even moderately large buses. In such chips the multiplexer-based approach is the only practical alternative. In practice, circuits are designed with CAD tools. If the designer describes the circuit using tri-state buffers, but there are not enough such buffers in the target device, then the CAD tools automatically produce an equivalent circuit that uses multiplexers.

Verilog Code

This section presents Verilog code for our circuit example that swaps the contents of two registers. We first give the code for the style of circuit in Figure 7.60 that uses tri-state buffers to implement the bus and then give the code for the style of circuit in Figure 7.65 that uses multiplexers. The code is written in a hierarchical fashion, using subcircuits for the registers, tri-state buffers, and the shift register. Figure 7.66 gives the code for an n-bit register of the type in Figure 7.61. The number of bits in the register is set by the parameter n, which has the default value of 8. The register is specified such that if the input $Rin = 1$, then the flip-flops are loaded from the n-bit input R. Otherwise, the flip-flops retain their presently stored values.

Figure 7.67 gives the code for a subcircuit that represents n tri-state buffers, each enabled by the input E. The inputs to the buffers are the n-bit signal Y, and the outputs are the n-bit signal F. The conditional assignment statement specifies that the output of

```
module  regn (R, Rin, Clock, Q);
    parameter n = 8;
    input  [n−1:0] R;
    input  Rin, Clock;
    output  [n−1:0] Q;
    reg  [n−1:0] Q;

    always @(posedge Clock)
        if (Rin)
            Q <= R;

endmodule
```

Figure 7.66 Code for an n-bit register of the type in Figure 7.61.

```
module trin (Y, E, F);
   parameter n = 8;
   input [n−1:0] Y;
   input E;
   output [n−1:0] F;
   wire [n−1:0] F;

   assign F = E ? Y : 'bz;

endmodule
```

Figure 7.67 Code for an n-bit tri-state module.

each buffer is set to $F = Y$ if $E = 1$; otherwise, the output is set to the high impedance value z. The conditional assignment statement uses an unsized number to define the high impedance case. The Verilog compiler will make the size of this number the same as the size of vector Y, namely n. We cannot define the number as n'bz because the size of a sized number cannot be given as a parameter.

Figure 7.68 defines a shift register that can be used to implement the control circuit in Figure 7.62. The number of flip-flops is set by the generic parameter m, which has the default value of 4. The shift register has an active-low asynchronous reset input. The shift operation is defined with a **for** loop in the style used in Example 7.12.

```
module shiftr (Resetn, w, Clock, Q);
   parameter m = 4;
   input Resetn, w, Clock;
   output [1:m] Q;
   reg [1:m] Q;
   integer k;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
      begin
         for (k = m; k > 1 ; k = k−1)
            Q[k] <= Q[k−1];
         Q[1] <= w;
      end

endmodule
```

Figure 7.68 Code for the shift register in Figure 7.62.

The code in Figure 7.69 represents a digital system like the one in Figure 7.60, with 3 eight-bit registers, *R*1, *R*2, and *R*3. The circuit in Figure 7.60 includes tri-state buffers that are used to place *n* bits of externally supplied data on the bus. In Figure 7.69, these buffers are instantiated in the module *tri_ext*. Each of the eight buffers is enabled by the input signal *Extern*, and the data inputs on the buffers are attached to the eight-bit signal *Data*. When *Extern* = 1, the value of *Data* is placed on the bus, which is represented by the signal *BusWires*. The *BusWires* vector represents the circuit's output as well as the internal bus wiring. We declared this vector to be of **tri** type rather than of **wire** type. The keyword **tri** is treated in the same way as the keyword **wire** by the Verilog compiler. The designation **tri** makes it obvious to a reader that the synthesized connections will have tri-state capability.

We assume that a three-bit control signal named *RinExt* exists, which allows the externally supplied data to be loaded from the bus into register *R*1, *R*2, or *R*3. The *RinExt*

```
module  swap (Data, Resetn, w, Clock, Extern, RinExt, BusWires);
   input  [7:0] Data;
   input  Resetn, w, Clock, Extern;
   input  [1:3] RinExt;
   output  [7:0] BusWires;
   tri  [7:0] BusWires;
   wire  [1:3] Rin, Rout, Q;
   wire  [7:0] R1, R2, R3;

   shiftr  control (Resetn, w, Clock, Q);
      defparam control.m = 3;

   assign  Rin[1] = RinExt[1] | Q[3];
   assign  Rin[2] = RinExt[2] | Q[2];
   assign  Rin[3] = RinExt[3] | Q[1];
   assign  Rout[1] = Q[2];
   assign  Rout[2] = Q[1];
   assign  Rout[3] = Q[3];

   regn  reg_1 (BusWires, Rin[1], Clock, R1);
   regn  reg_2 (BusWires, Rin[2], Clock, R2);
   regn  reg_3 (BusWires, Rin[3], Clock, R3);

   trin  tri_ext (Data, Extern, BusWires);
   trin  tri_1 (R1, Rout[1], BusWires);
   trin  tri_2 (R2, Rout[2], BusWires);
   trin  tri_3 (R3, Rout[3], BusWires);

endmodule
```

Figure 7.69 A digital system like the one in Figure 7.60.

input is not shown in Figure 7.60, to keep the figure simple, but it would be generated by the same external circuit block that produces *Extern* and *Data*. When *RinExt*[1] = 1, the data on the bus is loaded into register *R*1; when *RinExt*[2] = 1, the data is loaded into *R*2; and when *RinExt*[3] = 1, the data is loaded into *R*3.

In Figure 7.69 the three-bit shift register is instantiated using the *shiftr* module under the instance name *control*. The outputs of the shift register are the three-bit signal Q. The parameter that defines the number of flip-flops in the *shiftr* module, *m*, has the default value of 4. Since we need to instantiate only a three-bit shift register, we have to change the value of parameter *m*. The parameter is set with the statement

defparam control.m = 3;

The **defparam** statement defines the values of the parameters indicated. The intended module instance is identified using the syntax *instance_name.parameter_name*. In our example, the instance name is *control* and the parameter name is *m*.

The next three statements in Figure 7.69 connect Q to the control signals that determine when data is loaded into each register, which are represented by the three-bit signal *Rin*. The signals *Rin*[1], *Rin*[2], and *Rin*[3] in the code correspond to the signals $R1_{in}$, $R2_{in}$, and $R3_{in}$ in Figure 7.60. As specified in Figure 7.62, the left-most shift-register output, Q[1], controls when data is loaded into register *R*3. Similarly, Q[2] controls register *R*2, and Q[3] controls *R*1. Each bit in *Rin* is ORed with the corresponding bit in *RinExt* so that externally supplied data can be stored in the registers as discussed above. The code also connects the shift-register outputs to the enable inputs, *Rout*, on the tri-state buffers. Figure 7.62 shows that Q[1] is used to put the contents of *R*2 onto the bus; hence *Rout*[2] is assigned the value of Q[1]. Similarly, *Rout*[1] is assigned the value of Q[2], and *Rout*[3] is assigned the value of Q[3]. The remaining statements in the code instantiate the registers and tri-state buffers in the system.

Verilog Code Using Multiplexers

Figure 7.70 shows how the code in Figure 7.69 can be modified to use multiplexers instead of tri-state buffers. Using the circuit structure shown in Figure 7.65, the bus is implemented with eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers *R*1, *R*2, and *R*3. The fourth data input is connected to one bit of the *Data* input signal to allow externally supplied data to be written into the registers. When the shift register's contents are 000, the multiplexers select *Data* to be placed on the bus. This data is loaded into the register selected by *RinExt*. It is loaded into *R*1 if *RinExt*[1] = 1, *R*2 if *RinExt*[2] = 1, and *R*3 if *RinExt*[3] = 1.

The *Rout* signal in Figure 7.69, which enables the tri-state buffers connected to the bus, is not needed for the multiplexer implementation. Instead, we have to provide the select inputs on the multiplexers. In Figure 7.70, the shift-register outputs are called Q. These signals generate the *Rin* control signals for the registers in the same way as shown in Figure 7.69. We said in the discussion concerning Figure 7.65 that an encoder is needed between the shift-register outputs and the multiplexer select inputs. A suitable encoder is described in the first **if-else** statement in Figure 7.70. It produces the multiplexer select inputs, which are named *S*. It sets $S = 00$ when the shift register contains 000, $S = 10$ when the shift register contains 100, and so on, as given in the code. The multiplexers are described by

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock;
   input [1:3] RinExt;
   output [7:0] BusWires;
   reg [7:0] BusWires;
   wire [1:3] Rin, Q;
   wire [7:0] R1, R2, R3;
   reg [1:0] S;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;
   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];
   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   always @(Q or Data or R1 or R2 or R3 or S)
   begin
      // Encoder
      if (Q == 3'b000)   S = 2'b00;
      else if (Q == 3'b100)   S = 2'b10;
      else if (Q == 3'b010)   S = 2'b01;
      else   S = 2'b11;

      // Multiplexers
      if (S == 2'b00)   BusWires = Data;
      else if (S == 2'b01)   BusWires = R1;
      else if (S == 2'b10)   BusWires = R2;
      else   BusWires = R3;
   end

endmodule
```

Figure 7.70 Using multiplexers to implement a bus.

the second **if-else** statement, which places the value of *Data* onto the bus (*BusWires*) if $S = 00$, the contents of register $R1$ if $S = 01$, and so on. Using this scheme, when the swap operation is not active, the multiplexers place the bits from the *Data* input on the bus.

As described above, Figure 7.70 uses two **if-else** statements, one to describe an encoder and the other to describe the bus multiplexers. A simpler approach is to write a single **if-else** statement as shown in Figure 7.71. Here, each clause specifies directly which signal should

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
    input [7:0] Data;
    input Resetn, w, Clock;
    input [1:3] RinExt;
    output [7:0] BusWires;
    reg [7:0] BusWires;
    wire [1:3] Rin, Q;
    wire [7:0] R1, R2, R3;

    shiftr control (Resetn, w, Clock, Q);
        defparam control.m = 3;

    assign Rin[1] = RinExt[1] | Q[3];
    assign Rin[2] = RinExt[2] | Q[2];
    assign Rin[3] = RinExt[3] | Q[1];

    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    always @(Q or Data or R1 or R2 or R3)
    begin
        if (Q == 3'b000)   BusWires = Data;
        else if (Q == 3'b100)   BusWires = R2;
        else if (Q == 3'b010)   BusWires = R1;
        else   BusWires = R3;
    end

endmodule
```

Figure 7.71 A simplified version of the specification in Figure 7.70.

appear on *BusWires* for each pattern of the shift-register outputs. The circuit generated from the code in Figure 7.71 is equivalent to the one generated from the code in Figure 7.70.

Figure 7.72 gives an example of a timing simulation for a circuit synthesized from the code in Figure 7.71. In the first half of the simulation, the circuit is reset, and the contents of registers $R1$ and $R2$ are initialized. The hex value 55 is loaded into $R1$, and the value AA is loaded into $R2$. The clock edge at 275 ns, marked by the vertical reference line in Figure 7.72, loads the value $w = 1$ into the shift register. The contents of $R2$ (AA) then appear on the bus and are loaded into $R3$ by the clock edge at 325 ns. Following this clock edge, the contents of the shift register are 010, and the data stored in $R1$ (55) is on the bus. The clock edge at 375 ns loads this data into $R2$ and changes the shift register to 001. The contents of $R3$ (AA) now appear on the bus and are loaded into $R1$ by the clock edge at 425 ns. The shift register is now in state 000, and the swap is completed.

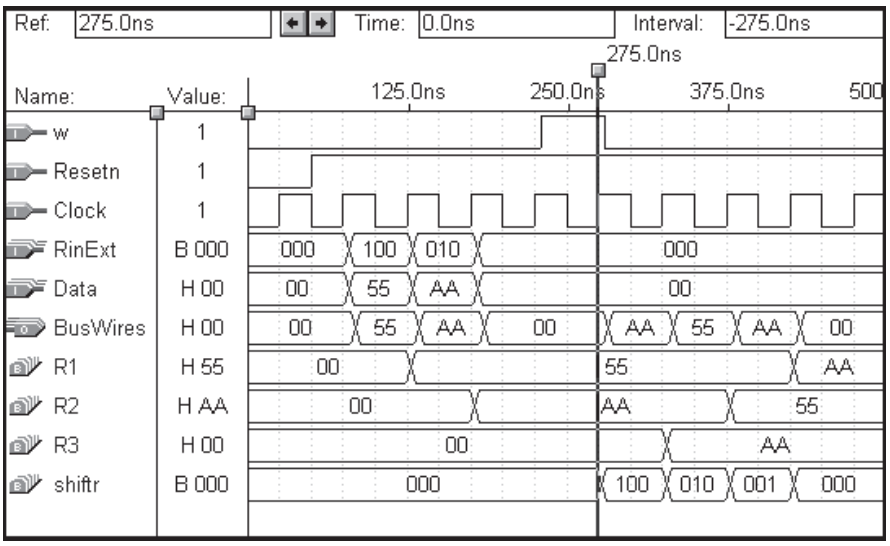


Figure 7.72 Timing simulation for the Verilog code in Figure 7.71.

7.14.2 SIMPLE PROCESSOR

A second example of a digital system like the one in Figure 7.60 is shown in Figure 7.73. It has four n-bit registers, $R0, \ldots, R3$, that are connected to the bus with tri-state buffers. External data can be loaded into the registers from the n-bit *Data* input, which is connected to the bus using tri-state buffers enabled by the *Extern* control signal. The system also includes an adder/subtractor module. One of its data inputs is provided by an n-bit register, A, that is attached to the bus, while the other data input, B, is directly connected to the bus. If the *AddSub* signal has the value 0, the module generates the sum $A + B$; if $AddSub = 1$, the module generates the difference $A - B$. To perform the subtraction, we assume that the adder/subtractor includes the required XOR gates to form the 2's complement of B, as discussed in section 5.3. The register G stores the output produced by the adder/subtractor. The A and G registers are controlled by the signals A_{in}, G_{in}, and G_{out}.

The system in Figure 7.73 can perform various functions, depending on the design of the control circuit. As an example, we will design a control circuit that can perform the four operations listed in Table 7.2. The left column in the table shows the name of an operation and its operands; the right column indicates the function performed in the operation. For the *Load* operation the meaning of $Rx \leftarrow Data$ is that the data on the external *Data* input is transferred across the bus into any register, Rx, where Rx can be $R0$ to $R3$. The *Move* operation copies the data stored in register Ry into register Rx. In the table the square brackets, as in $[Rx]$, refer to the *contents* of a register. Since only a single transfer across the bus is needed, both the *Load* and *Move* operations require only one step (clock cycle) to be completed. The *Add* and *Sub* operations require three steps, as follows: In the first step

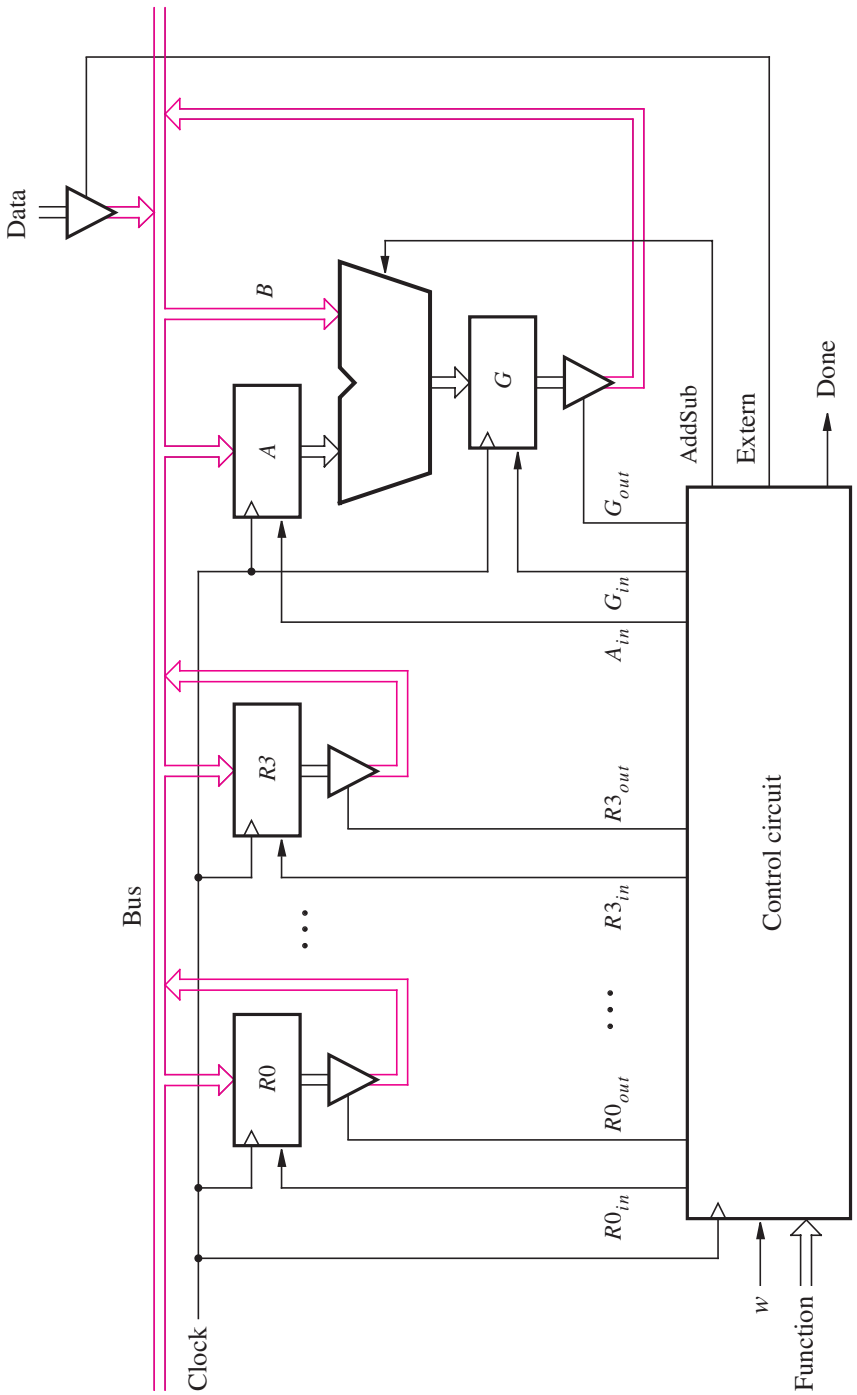


Figure 7.73 A digital system that implements a simple processor.

Table 7.2 Operations performed in the processor.

Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

the contents of *Rx* are transferred across the bus into register *A*. Then in the next step, the contents of *Ry* are placed onto the bus. The adder/subtractor module performs the required function, and the results are stored in register *G*. Finally, in the third step the contents of *G* are transferred into *Rx*.

A digital system that performs the types of operations listed in Table 7.2 is usually called a *processor*. The specific operation to be performed at any given time is indicated using the control circuit input named *Function*. The operation is initiated by setting the *w* input to 1, and the control circuit asserts the *Done* output when the operation is completed.

In Figure 7.60 we used a shift register to implement the control circuit. It is possible to use a similar design for the system in Figure 7.73. To illustrate a different approach, we will base the design of the control circuit on a counter. This circuit has to generate the required control signals in each step of each operation. Since the longest operations (*Add* and *Sub*) need three steps (clock cycles), a two-bit counter can be used. Figure 7.74 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in section 6.2. The decoder is enabled at all times by setting its enable (*En*) input permanently to the value 1. Each of the decoder outputs represents a step in an operation. When no operation is currently being performed, the count value is 00; hence the T_0 output of the decoder is

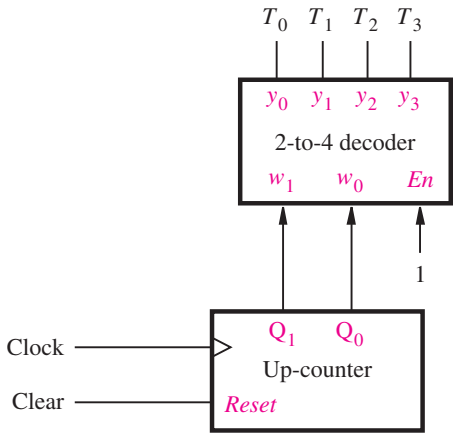


Figure 7.74 A part of the control circuit for the processor.

asserted. In the first step of an operation, the count value is 01, and T_1 is asserted. During the second and third steps of the *Add* and *Sub* operations, T_2 and T_3 are asserted, respectively.

In each of steps T_0 to T_3, various control signal values have to be generated by the control circuit, depending on the operation being performed. Figure 7.75 shows that the operation is specified with six bits, which form the *Function* input. The two left-most bits, $F = f_1 f_0$, are used as a two-bit number that identifies the operation. To represent *Load*, *Move*, *Add*, and *Sub*, we use the codes $f_1 f_0 = 00$, 01, 10, and 11, respectively. The inputs $Rx_1 Rx_0$ are a binary number that identifies the *Rx* operand, while $Ry_1 Ry_0$ identifies the *Ry* operand. The *Function* inputs are stored in a six-bit Function Register when the FR_{in} signal is asserted.

Figure 7.75 also shows three 2-to-4 decoders that are used to decode the information encoded in the *F*, *Rx*, and *Ry* inputs. We will see shortly that these decoders are included as a convenience because their outputs provide simple-looking logic expressions for the various control signals.

The circuits in Figures 7.74 and 7.75 form a part of the control circuit. Using the input *w* and the signals $T_0, \ldots, T_3$, $I_0, \ldots, I_3$, $X_0, \ldots, X_3$, and $Y_0, \ldots, Y_3$, we will show how to derive the rest of the control circuit. It has to generate the outputs *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}, \ldots, R3_{in}$, and $R0_{out}, \ldots, R3_{out}$. The control circuit also has to generate the *Clear* and FR_{in} signals used in Figures 7.74 and 7.75.

Clear and FR_{in} are defined in the same way for all operations. *Clear* is used to ensure that the count value remains at 00 as long as $w = 0$ and no operation is being executed. Also, it is used to clear the count value to 00 at the end of each operation. Hence an appropriate

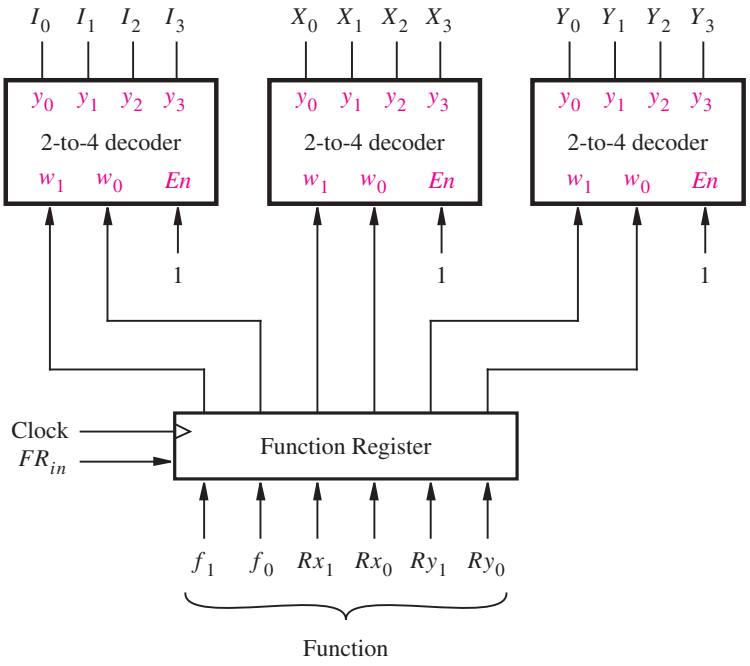


Figure 7.75 The function register and decoders.

logic expression is

$$Clear = \overline{w}\, T_0 + Done$$

The FR_{in} signal is used to load the values on the *Function* inputs into the Function Register when w changes to 1. Hence

$$FR_{in} = wT_0$$

The rest of the outputs from the control circuit depend on the specific step being performed in each operation. The values that have to be generated for each signal are shown in Table 7.3. Each row in the table corresponds to a specific operation, and each column represents one time step. The *Extern* signal is asserted only in the first step of the *Load* operation. Therefore, the logic expression that implements this signal is

$$Extern = I_0 T_1$$

Done is asserted in the first step of *Load* and *Move*, as well as in the third step of *Add* and *Sub*. Hence

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

The A_{in}, G_{in}, and G_{out} signals are asserted in the *Add* and *Sub* operations. A_{in} is asserted in step T_1, G_{in} is asserted in T_2, and G_{out} is asserted in T_3. The *AddSub* signal has to be set to 0 in the *Add* operation and to 1 in the *Sub* operation. This is achieved with the following logic expressions

$$\begin{aligned} A_{in} &= (I_2 + I_3)T_1 \\ G_{in} &= (I_2 + I_3)T_2 \\ G_{out} &= (I_2 + I_3)T_3 \\ AddSub &= I_3 \end{aligned}$$

The values of $R0_{in}, \ldots, R3_{in}$ are determined using either the $X_0, \ldots, X_3$ signals or the $Y_0, \ldots, Y_3$ signals. In Table 7.3 these actions are indicated by writing either $R_{in} = X$ or $R_{in} = Y$. The meaning of $R_{in} = X$ is that $R0_{in} = X_0$, $R1_{in} = X_1$, and so on. Similarly, the values of $R0_{out}, \ldots, R3_{out}$ are specified using either $R_{out} = X$ or $R_{out} = Y$.

Table 7.3 Control signals asserted in each operation/time step.

	T_1	T_2	T_3
(Load): I_0	$Extern, R_{in} = X, Done$		
(Move): I_1	$R_{in} = X, R_{out} = Y, Done$		
(Add): I_2	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in}, AddSub = 0$	$G_{out}, R_{in} = X, Done$
(Sub): I_3	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in}, AddSub = 1$	$G_{out}, R_{in} = X, Done$

We will develop the expressions for $R0_{in}$ and $R0_{out}$ by examining Table 7.3 and then show how to derive the expressions for the other register control signals. The table shows that $R0_{in}$ is set to the value of X_0 in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Similarly, $R0_{out}$ is set to the value of Y_0 in the first step of *Move*. It is set to X_0 in the first step of *Add* and *Sub* and to Y_0 in the second step of these operations, which gives

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

The expressions for $R1_{in}$ and $R1_{out}$ are the same as those for $R0_{in}$ and $R0_{out}$ except that X_1 and Y_1 are used in place of X_0 and Y_0. The expressions for $R2_{in}$, $R2_{out}$, $R3_{in}$, and $R3_{out}$ are derived in the same way.

The circuits shown in Figures 7.74 and 7.75, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.73.

Processors are extremely useful circuits that are widely used. We have presented only the most basic aspects of processor design. However, the techniques presented can be extended to design realistic processors, such as modern microprocessors. The interested reader can refer to books on computer organization for more details on processor design [1–2].

Verilog Code

In this section we give two different styles of Verilog code for describing the system in Figure 7.73. The first style uses tri-state buffers to represent the bus, and it gives the logic expressions shown above for the outputs of the control circuit. The second style of code uses multiplexers to represent the bus, and it uses **case** statements that correspond to Table 7.3 to describe the outputs of the control circuit.

Verilog code for an up-counter is shown in Figure 7.56. A modified version of this counter, named *upcount*, is shown in the code in Figure 7.76. It has a synchronous reset input, which is active high. Other subcircuits that we use in the Verilog code for the processor are the *dec2to4*, *regn*, and *trin* modules in Figures 6.35, 7.66, and 7.67.

```
module upcount (Clear, Clock, Q);
   input Clear, Clock;
   output [1:0] Q;
   reg [1:0] Q;

   always @(posedge Clock)
      if (Clear)
         Q <= 0;
      else
         Q <= Q + 1;

endmodule
```

Figure 7.76 A two-bit up-counter with synchronous reset.

Complete code for the processor is given in Figure 7.77. The instantiated modules *counter* and *decT* represent the subcircuits in Figure 7.74. Note that we have assumed that the circuit has an active-high reset input, *Reset*, which is used to initialize the counter to 00. The statement **assign** Func = {F, Rx, Ry} uses the concatenate operator to create the six-bit signal *Func*, which represents the inputs to the Function Register in Figure 7.75. The *functionreg* module represents the Function Register with the data inputs *Func* and the

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
   input [7:0] Data;
   input Reset, w, Clock;
   input [1:0] F, Rx, Ry;
   output [7:0] BusWires;
   output Done;
   wire [7:0] BusWires;
   reg [0:3] Rin, Rout;
   reg [7:0] Sum;
   wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
   wire [1:0] Count;
   wire [0:3] T, I, Xreg, Y;
   wire [7:0] R0, R1, R2, R3, A, G;
   wire [1:6] Func, FuncReg;
   integer k;

   upcount counter (Clear, Clock, Count);
   dec2to4 decT (Count, 1, T);

   assign Clear = Reset | Done | (~w & T[0]);
   assign Func = {F, Rx, Ry};
   assign FRin = w & T[0];

   regn functionreg (Func, FRin, Clock, FuncReg);
      defparam functionreg.n = 6;
   dec2to4 decI (FuncReg[1:2], 1, I);
   dec2to4 decX (FuncReg[3:4], 1, Xreg);
   dec2to4 decY (FuncReg[5:6], 1, Y);

   assign Extern = I[0] & T[1];
   assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
   assign Ain = (I[2] | I[3]) & T[1];
   assign Gin = (I[2] | I[3]) & T[2];
   assign Gout = (I[2] | I[3]) & T[3];
   assign AddSub = I[3];
```

. . . continued in Part *b*.

Figure 7.77 Code for the prcoessor (Part *a*).

```
// RegCntl
always @(I or T or Xreg or Y)
    for (k = 0; k < 4; k = k+1)
    begin
        Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
            ((I[2] | I[3]) & T[1] & Y[k]);
        Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
            ((T[1] & Xreg[k]) | (T[2] & Y[k])));
    end

trin tri_ext (Data, Extern, BusWires);
regn  reg_0 (BusWires, Rin[0], Clock, R0);
regn  reg_1 (BusWires, Rin[1], Clock, R1);
regn  reg_2 (BusWires, Rin[2], Clock, R2);
regn  reg_3 (BusWires, Rin[3], Clock, R3);

trin  tri_0 (R0, Rout[0], BusWires);
trin  tri_1 (R1, Rout[1], BusWires);
trin  tri_2 (R2, Rout[2], BusWires);
trin  tri_3 (R3, Rout[3], BusWires);
regn  reg_A (BusWires, Ain, Clock, A);

// alu
always @(AddSub or A or BusWires)
    if (!AddSub)
        Sum = A + BusWires;
    else
        Sum = A − BusWires;

regn  reg_G (Sum, Gin, Clock, G);
trin  tri_G (G, Gout, BusWires);

endmodule
```

Figure 7.77 Code for the processor (Part *b*).

outputs *FuncReg*. The instantiated modules *decI*, *decX*, and *decY* represent the decoders in Figure 7.75. Following these statements the previously derived logic expressions for the outputs of the control circuit are given. For $R0_{in}, \ldots, R3_{in}$ and $R0_{out}, \ldots, R3_{out}$, a **for** loop is used to produce the expressions.

At the end of the code, the adder/subtractor module is defined and the tri-state buffers and registers in the processor are instantiated.

Using Multiplexers and Case Statements

We showed in Figure 7.65 that a bus can be implemented with multiplexers, rather than tri-state buffers. Verilog code that describes the processor using this approach is shown in Figure 7.78. The code illustrates a different way of describing the control circuit in the

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output [7:0] BusWires;
    output Done;
    reg [7:0] BusWires, Sum;
    reg [0:3] Rin, Rout;
    reg Extern, Done, Ain, Gin, Gout, AddSub;
    wire [1:0] Count, I;
    wire [0:3] Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg, Sel;

    wire Clear = Reset | Done | (~w & ~Count[1] & ~Count[0]);
    upcount counter (Clear, Clock, Count);
    assign Func = {F, Rx, Ry};
    wire FRin = w & ~Count[1] & ~Count[0];
    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    assign I = FuncReg[1:2];
    dec2to4 decX (FuncReg[3:4], 1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1, Y);

    always @(Count or I or Xreg or Y)
    begin
        Extern = 1'b0; Done = 1'b0; Ain = 1'b0; Gin = 1'b0;
        Gout = 1'b0; AddSub = 1'b0; Rin = 4'b0; Rout = 4'b0;
        case (Count)
            2'b00: ; //no signals asserted in time step T0
            2'b01:    //define signals in time step T1
                case (I)
                    2'b00: begin //Load
                                Extern = 1'b1; Rin = Xreg; Done = 1'b1;
                           end
                    2'b01: begin //Move
                                Rout = Y; Rin = Xreg; Done = 1'b1;
                           end
                    default: begin //Add, Sub
                                Rout = Xreg; Ain = 1'b1;
                             end
                endcase
```

. . . continued in Part *b*.

Figure 7.78 Alternative code for the processor (Part *a*).

```
            2'b10: //define signals in time step T2
               case(I)
                  2'b10: begin //Add
                              Rout = Y; Gin = 1'b1;
                         end
                  2'b11: begin //Sub
                              Rout = Y; AddSub = 1'b1; Gin = 1'b1;
                         end
                  default: ; //Add, Sub
               endcase
            2'b11:
               case (I)
                  2'b10, 2'b11: begin
                                   Gout = 1'b1; Rin = Xreg; Done = 1'b1;
                                end
                  default: ; //Add, Sub
               endcase
         endcase
      end

      regn  reg_0 (BusWires, Rin[0], Clock, R0);
      regn  reg_1 (BusWires, Rin[1], Clock, R1);
      regn  reg_2 (BusWires, Rin[2], Clock, R2);
      regn  reg_3 (BusWires, Rin[3], Clock, R3);
      regn  reg_A (BusWires, Ain, Clock, A);
```

. . . continued in Part *c*.

Figure 7.78 Alternative code for the processor (Part *b*).

processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as in Figure 7.77. Instead, **case** statements are used to represent the information shown in Table 7.3. Each control signal is first assigned the value 0 as a default. This is required because the **case** statements specify the values of the control signals only when they should be asserted, as we did in Table 7.3. As explained in section 7.12.2, when the value of a signal is not specified, the signal retains its current value. This implied memory results in a feedback connection in the synthesized circuit. We avoid this problem by providing the default value of 0 for each of the control signals involved in the **case** statements.

In Figure 7.77 the decoders *decT* and *decI* are used to decode the *Count* signal and the stored values of the F input, respectively. The *decT* decoder has the outputs $T_0, \ldots, T_3$, and *decI* produces $I_0, \ldots, I_3$. In Figure 7.78 these two decoders are not used, because they do not serve a useful purpose in this code. Instead, the signal I is defined as a two-bit signal, and the two-bit signal *Count* is used instead of T. These signals are used in the **case** statements. The code sets I to the value of the two left-most bits in the Function Register, which correspond to the stored values of the input F.

```
// alu
always @(AddSub or A or BusWires)
begin
    if (!AddSub)
        Sum = A + BusWires;
    else
        Sum = A − BusWires;
end

regn reg_G (Sum, Gin, Clock, G);
assign Sel = {Rout, Gout, Extern};

always @(Sel or R0 or R1 or R2 or R3 or G or Data)
begin
    if (Sel == 6'b100000)
        BusWires = R0;
    else if (Sel == 6'b010000)
        BusWires = R1;
    else if (Sel == 6'b001000)
        BusWires = R2;
    else if (Sel == 6'b000100)
        BusWires = R3;
    else if (Sel == 6'b000010)
        BusWires = G;
    else BusWires = Data;
end

endmodule
```

Figure 7.78 Alternative code for the processor (Part *c*).

There are two nested levels of **case** statements. The first one enumerates the possible values of *Count*. For each alternative in this **case** statement, which represents a column in Table 7.3, there is a nested **case** statement that enumerates the four values of *I*. As indicated by the comments in the code, the nested **case** statements correspond exactly to the information given in Table 7.3.

At the end of Figure 7.78, the bus is described with an **if-else** statement which represents multiplexers that place the appropriate data onto *BusWires*, depending on the values of R_{out}, G_{out}, and *Extern*.

The circuits synthesized from the code in Figures 7.77 and 7.78 are functionally equivalent. The style of code in Figure 7.78 has the advantage that it does not require the manual effort of analyzing Table 7.3 to generate the logic expressions for the control signals in Figure 7.77. By using the style of code in Figure 7.78, these expressions are produced automatically by the Verilog compiler as a result of analyzing the **case** statements. The style of code in Figure 7.78 is less prone to careless errors. Also, using this style of code it

would be straightforward to provide additional capabilities in the processor, such as adding other operations.

We synthesized a circuit to implement the code in Figure 7.78 in a chip. Figure 7.79 gives an example of the results of a timing simulation. Each clock cycle in which $w = 1$ in this timing diagram indicates the start of an operation. In the first such operation, at 250 ns in the simulation time, the values of both inputs F and Rx are 00. Hence the operation corresponds to "*Load* $R0$, *Data*." The value of *Data* is 2A, which is loaded into $R0$ on the next positive clock edge. The next operation loads 55 into register $R1$, and the subsequent operation loads 22 into $R2$. At 850 ns the value of the input F is 10, while $Rx = 01$ and $Ry = 00$. This operation is "*Add* $R1$, $R0$." In the following clock cycle, the contents of $R1$ (55) appear on the bus. This data is loaded into register A by the clock edge at 950 ns, which also results in the contents of $R0$ (2A) being placed on the bus. The adder/subtractor module generates the correct sum (7F), which is loaded into register G at 1050 ns. After this clock edge the new contents of G (7F) are placed on the bus and loaded into register $R1$ at 1150 ns. Two more operations are shown in the timing diagram. The one at 1250 ns ("*Move* $R3$, $R1$") copies the contents of $R1$ (7F) into $R3$. Finally, the operation starting at 1450 ns ("*Sub* $R3$, $R2$") subtracts the contents of $R2$ (22) from the contents of $R3$ (7F), producing the correct result, $7F - 22 = 5D$.

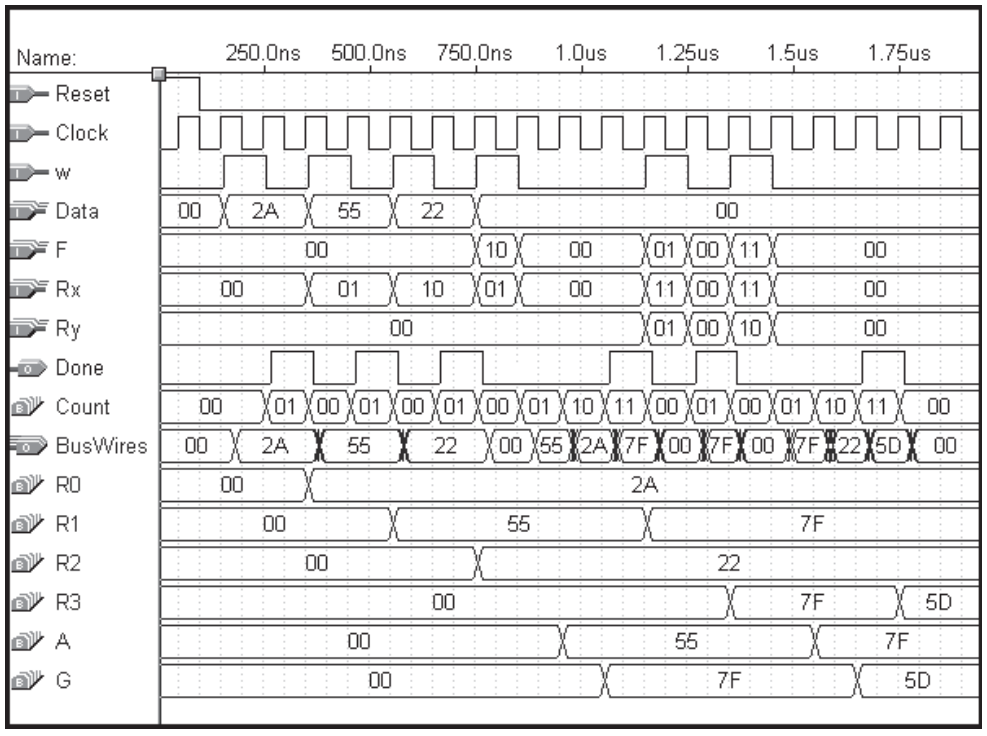


Figure 7.79 Timing simulation for the Verilog code in Figure 7.78.

7.14.3 REACTION TIMER

We showed in Chapter 3 that electronic devices operate at remarkably fast speeds, with the typical delay through a logic gate being less than 1 ns. In this example we use a logic circuit to measure the speed of a much slower type of device—a person.

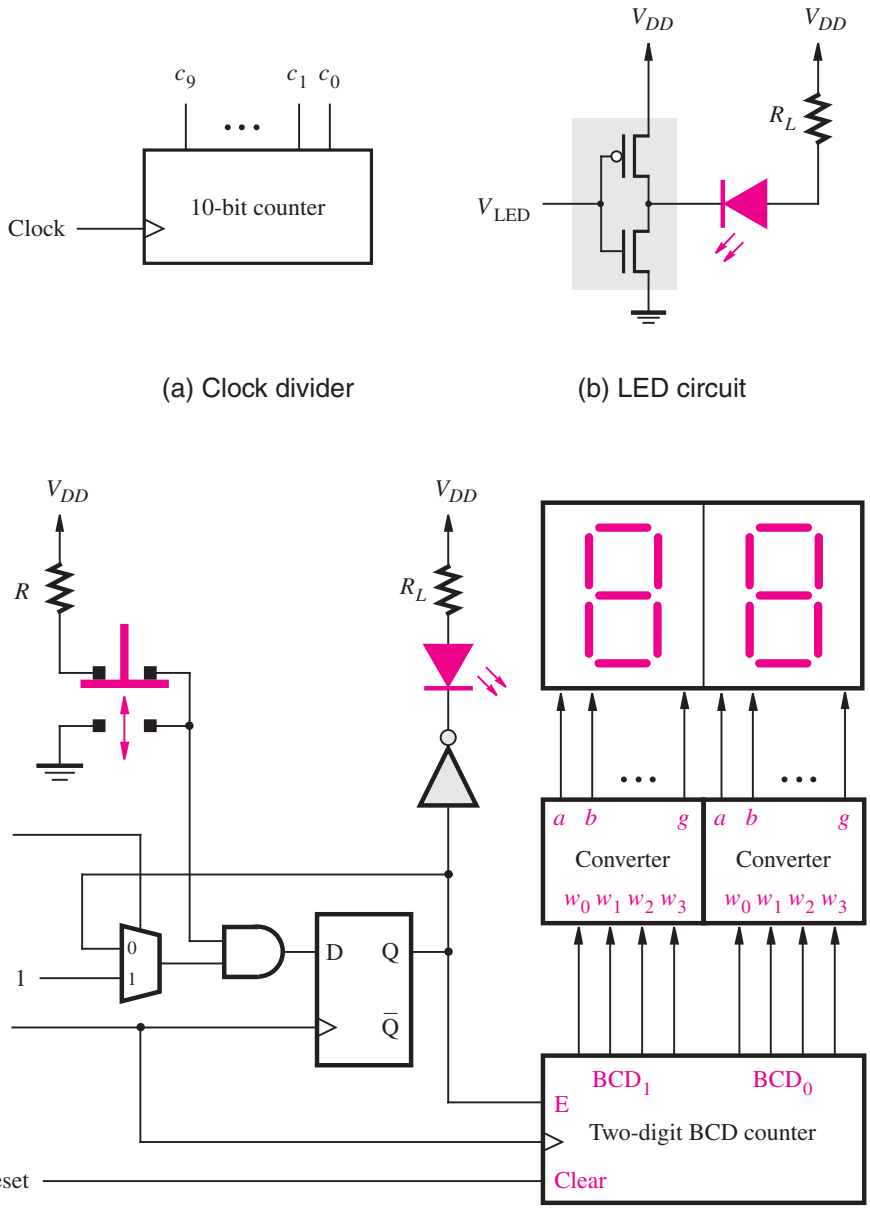
We will design a circuit that can be used to measure the reaction time of a person to a specific event. The circuit turns on a small light, called a *light-emitting diode (LED)*. In response to the LED being turned on, the person attempts to press a switch as quickly as possible. The circuit measures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

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

Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a four-bit counter is given in Figure 7.22. It shows that the least-significant bit output, Q_0, of the counter is a periodic signal with half the frequency of the clock input. Hence we can view Q_0 as dividing the clock frequency by two. Similarly, the Q_1 output divides the clock frequency by four. In general, output Q_i in an n-bit counter divides the clock frequency by 2^{i+1}. In the case of our 102.4 kHz clock signal, we can use a 10-bit counter, as shown in Figure 7.80*a*. The counter output c_9 has the required 100 Hz frequency because 102400 Hz/1024 = 100 Hz.

The reaction timer circuit has to be able to turn an LED on and off. The graphical symbol for an LED is shown in blue in Figure 7.80*b*. Small blue arrows in the symbol represent the light that is emitted when the LED is turned on. The LED has two terminals: the one on the left in the figure is the *cathode*, and the terminal on the right is the *anode*. To turn the LED on, the cathode has to be set to a lower voltage than the anode, which causes a current to flow through the LED. If the voltages on its two terminals are equal, the LED is off.

Figure 7.80*b* shows one way to control the LED, using an inverter. If the input voltage $V_{LED} = 0$, then the voltage at the cathode is equal to V_{DD}; hence the LED is off. But if $V_{LED} = V_{DD}$, the cathode voltage is 0 V and the LED is on. The amount of current that flows is limited by the value of the resistor R_L. This current flows through the LED and the NMOS transistor in the inverter. Since the current flows *into* the inverter, we say that the inverter *sinks* the current. The maximum current that a logic gate can sink without sustaining permanent damage is usually called I_{OL}, which stands for the "maximum current when the output is low." The value of R_L is chosen such that the current is less than I_{OL}. As an example assume that the inverter is implemented inside a PLD device. The typical value of I_{OL}, which would be specified in the data sheet for the PLD, is about 12 mA. For $V_{DD} = 5$ V, this leads to $R_L \approx 450\ \Omega$ because $5\ V/450\ \Omega = 11$ mA (there is actually a small voltage drop across the LED when it is turned on, but we ignore this for simplicity). The amount of light emitted by the LED is proportional to the current flow. If 11 mA is insufficient, then the inverter should be implemented in



(a) Clock divider

(b) LED circuit

(c) Push-button switch, LED, and 7-segment displays

Figure 7.80 A reaction-timer circuit.

a buffer chip, like those described in section 3.5, because buffers provide a higher value of I_{OL}.

The complete reaction-timer circuit is illustrated in Figure 7.80*c*, with the inverter from part (*b*) shaded in grey. The graphical symbol for a push-button switch is shown in the top left of the diagram. The switch normally makes contact with the top terminals, as depicted in the figure. When depressed, the switch makes contact with the bottom terminals; when released, it automatically springs back to the top position. In the figure the switch is connected such that it normally produces a logic value of 1, and it produces a 0 pulse when pressed.

The push-button switch is connected to the clear input on a D flip-flop. The output of this flip-flop determines whether the LED is on or off, and it also provides the count enable input to a two-digit BCD counter. As discussed in section 7.11, each digit in a BCD counter has four bits that take the values 0000 to 1001. Thus the counting sequence can be viewed as decimal numbers from 00 to 99. A circuit for the BCD counter is given in Figure 7.28. In Figure 7.80*c* both the flip-flop and the counter are clocked by the c_9 output of the clock divider in part (*a*) of the figure. The intended use of the reaction-timer circuit is to first depress the switch to turn off the LED and disable the counter. Then the *Reset* input is asserted to clear the contents of the counter to 00. The input w normally has the value 0, which keeps the flip-flop cleared and prevents the count value from changing. The reaction test is initiated by setting $w = 1$ for one c_9 clock cycle. After the next positive edge of c_9, the flip-flop output becomes a 1, which turns on the LED. We assume that w returns to 0 after one clock cycle, but the flip-flop output remains at 1 because of the 2-to-1 multiplexer connected to the D input. The counter is then incremented every 1/100 of a second. Each digit in the counter is connected through a code converter to a 7-segment display, which we described in the discussion for Figure 6.25. When the user depresses the switch, the flip-flop is cleared, which turns off the LED and stops the counter. The two-digit display shows the elapsed time to the nearest 1/100 of a second from when the LED was turned on until the user was able to respond by depressing the switch.

Verilog Code

To describe the circuit in Figure 7.80*c* using Verilog code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. The code for the latter subcircuit is given in Figure 6.38 and is not repeated here. Code for the BCD counter, which represents the circuit in Figure 7.28, is shown in Figure 7.81. The two-digit BCD output is represented by the 2 four-bit signals *BCD*1 and *BCD*0. The *Clear* input provides a synchronous reset for both digits in the counter. If $E = 1$, the count value is incremented on the positive clock edge; and if $E = 0$, the count value is unchanged. Each digit can take the values from 0000 to 1001.

Figure 7.82 gives the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit*1 and *Digit* 0.

In Figure 7.61 we showed how a register, R, can be designed with a control signal R_{in}. If $R_{in} = 1$ data is loaded into the register on the active clock edge and if $R_{in} = 0$, the stored contents of the register are not changed. The flip-flop in Figure 7.80 is used in the same

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

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

endmodule
```

Figure 7.81 Code for the two-digit BCD counter in Figure 7.28.

way. If $w = 1$, the flip-flop is loaded with the value 1, but if $w = 0$ the stored value in the flip-flop is not changed. This circuit is described by the **always** block in Figure 7.82, which also includes a synchronous reset input. We have chosen to use a synchronous reset because the flip-flop output is connected to the enable input E on the BCD counter. As we know from the discussion in section 7.3, it is important that all signals connected to flip-flops meet the required setup and hold times. The push-button switch can be pressed at any time and is not synchronized to the c_9 clock signal. By using a synchronous reset for the flip-flop in Figure 7.80, we avoid possible timing problems in the counter.

The flip-flop output is named *LED*, which is inverted to produce the *LEDn* signal that controls the LED. In the device used to implement the circuit, *LEDn* would be generated by a buffer that is connected to an output pin on the chip package. If a PLD is used, this buffer has the associated value of $I_{OL} = 12$ mA that we mentioned earlier. At the end of Figure 7.82, the BCD counter and 7-segment code converters are instantiated as subcircuits.

A simulation of the reaction-timer circuit implemented in a chip is shown in Figure 7.83. Initially, *Pushn* is set to 0 to simulate depressing the switch to turn off the LED, and

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

   always @(posedge c9)
   begin
      if (Pushn == 0)
         LED <= 0;
      else if (w)
         LED <= 1;
   end

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

endmodule
```

Figure 7.82 Code for the reaction timer.

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

7.14.4 REGISTER TRANSFER LEVEL (RTL) CODE

At this point, we have introduced most of the Verilog constructs that are needed for synthesis. Most of our examples give behavioral code, utilizing **if-else** statements, **case** statements, **for** loops, and other procedural statements. It is possible to write behavioral code in a style that resembles a computer program, in which there is a complex flow of control with many loops and branches. With such code, sometimes called *high-level* behavioral code, it is difficult to relate the code to the final hardware implementation; it may even be difficult to predict what circuit a high-level synthesis tool will produce. In this book we do not use the high-level

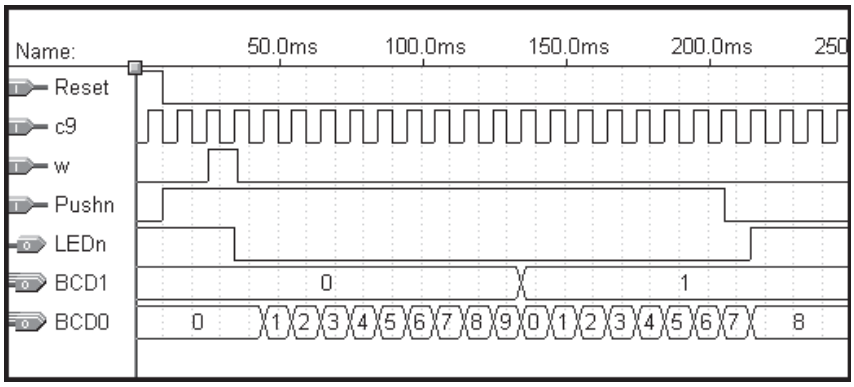


Figure 7.83 Simulation of the reaction timer circuit.

style of code. Instead, we present Verilog code in such a way that the code can be easily related to the circuit that is being described. Most design modules presented are fairly small, to facilitate simple descriptions. Larger designs are built by interconnecting the smaller modules. This approach is usually referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. RTL code is characterized by a straightforward flow of control through the code; it comprises well-understood subcircuits that are connected together in a simple way.

7.15 CONCLUDING REMARKS

In this chapter we have presented circuits that serve as basic storage elements in digital systems. These elements are used to build larger units such as registers, shift registers, and counters. Many other texts that deal with this material are available [3–11]. We have illustrated how circuits with flip-flops can be described using Verilog code. More information on Verilog can be found in [12–19]. In the next chapter a more formal method for designing circuits with flip-flops will be presented.

PROBLEMS

7.1 Consider the timing diagram in Figure P7.1. Assuming that the *D* and *Clock* inputs shown are applied to the circuit in Figure 7.12, draw waveforms for the Q_a, Q_b, and Q_c signals.

7.2 Can the circuit in Figure 7.3 be modified to implement an SR latch? Explain your answer.

7.3 Figure 7.5 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its truth table and show its timing diagram.

7.4 Show a circuit that implements the gated SR latch using NAND gates only.

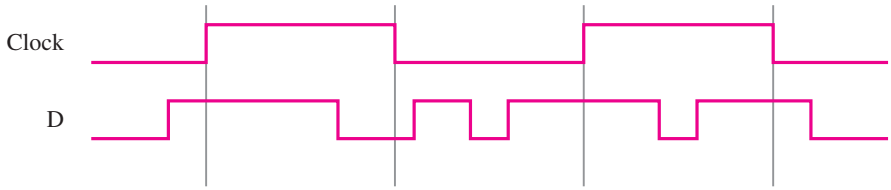


Figure P7.1 Timing diagram for problem 7.1.

7.5 Given a 100-MHz clock signal, derive a circuit using D flip-flops to generate 50-MHz and 25-MHz clock signals. Draw a timing diagram for all three clock signals, assuming reasonable delays.

7.6 An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.

7.7 The gated SR latch in Figure 7.6*a* has unpredictable behavior if the S and R inputs are both equal to 1 when the *Clk* changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition $S = R = 1$ cause the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.

7.8 Show how a JK flip-flop can be constructed using a T flip-flop and other logic gates.

7.9 Consider the circuit in Figure P7.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?

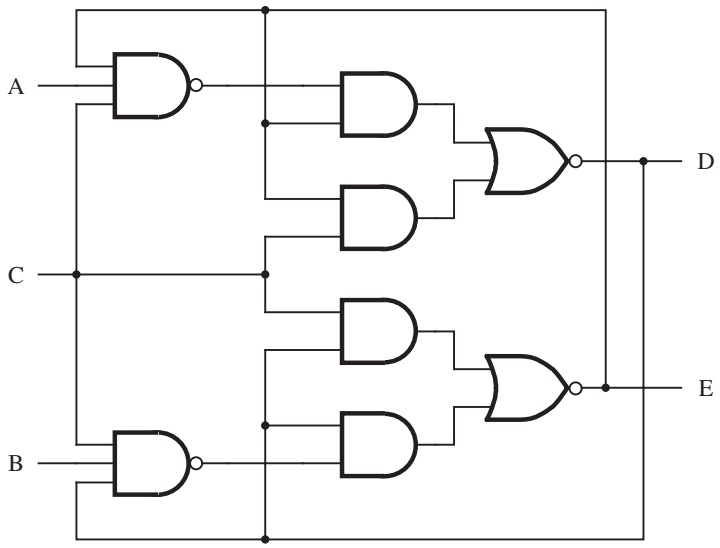


Figure P7.2 Circuit for problem 7.9.

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

7.11 Write Verilog code that represents a JK flip-flop. Use behavioral code, rather than structural code.

7.12 Synthesize a circuit for the code written for problem 7.11 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.

7.13 A four-bit barrel shifter is a combinational circuit with four data inputs, two control inputs, and two data outputs. It allows the data inputs to be shifted onto the outputs by 0, 1, 2, or 3 bit positions, with the rightmost bits wrapping around (rotating) to the leftmost bits. For example, if the data inputs are 1100 and the control input specifies a two-bit shift, then the output would be 0011. If the data input is 1110, a two-bit rotation produces 1011. Design a four-bit shift register using a barrel shifter that can shift to the right by 0, 1, 2, or 3 positions.

7.14 Write Verilog code for the shift register described in problem 7.13.

7.15 Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in section 7.9.3.

7.16 Design a three-bit up/down counter using T flip-flops. It should include a control input called $\overline{\text{Up}}$/Down. If $\overline{\text{Up}}$/Down = 0, then the circuit should behave as an up-counter. If $\overline{\text{Up}}$/Down = 1, then the circuit should behave as a down-counter.

7.17 Repeat problem 7.16 using D flip-flops.

7.18 The circuit in Figure P7.3 looks like a counter. What is the sequence that this circuit counts in?

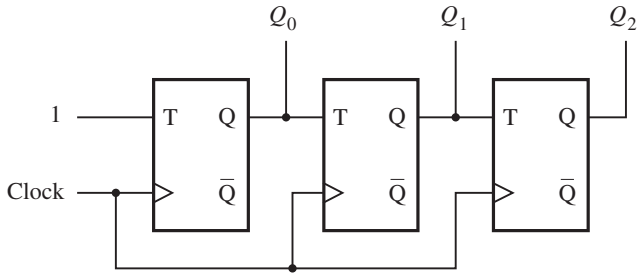


Figure P7.3 The circuit for problem 7.18.

7.19 Consider the circuit in Figure P7.4. How does this circuit compare with the circuit in Figure 7.17? Can the circuits be used for the same purposes? If not, what is the key difference between them?

7.20 Construct a NOR-gate circuit, similar to the one in Figure 7.11*a*, which implements a negative-edge-triggered D flip-flop.

7.21 Write Verilog code that represents a modulo-12 up-counter with synchronous reset.

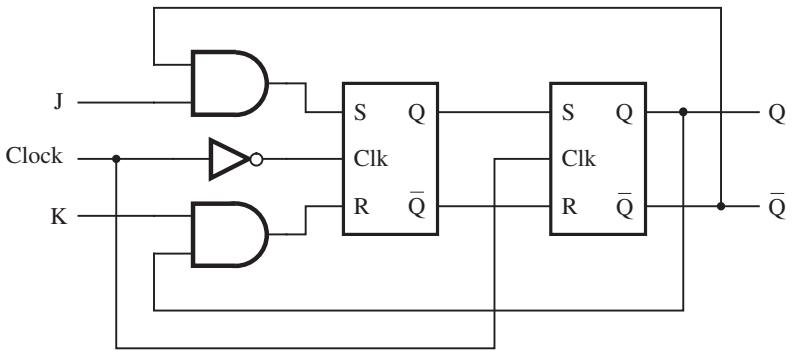


Figure P7.4 Circuit for problem 7.19.

7.22 For the flip-flops in the counter in Figure 7.25, assume that $t_{su} = 3$ ns, $t_h = 1$ ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?

7.23 Write Verilog code that represents an eight-bit Johnson counter. Synthesize the code with your CAD tools and give a timing simulation that shows the counting sequence.

7.24 Write Verilog code in the style shown in Figure 7.55 that represents a ring counter. Your code should have a parameter *n* that sets the number of flip-flops in the counter.

7.25 Write Verilog code that describes the functionality of the circuit shown in Figure 7.48.

7.26 Write Verilog code that instantiates the *lpm_counter* module from the LPM library. Configure the module as a 32-bit up-counter. For the counter circuit in Figure 7.24, we said that the AND-gate chain can be thought of as the carry-chain. The FLEX 10K FPGA contains special-purpose logic to implement this carry-chain such that it has minimal propagation delay. Use the MAX+plusII synthesis options to implement the *lpm_counter* in two ways: with the dedicated carry-chain used and with the dedicated carry-chain not used. Use the Timing Analyzer in MAX+plusII to determine the maximum speed of operation of the counter in both cases. See the tutorials in Appendices B, C, and D for instructions on using the appropriate features of the CAD tools.

7.27 Figure 7.69 gives Verilog code for a digital system that swaps the contents of two registers, *R*1 and *R*2, using register *R*3 for temporary storage. Create an equivalent schematic using your CAD tools for this system. Synthesize a circuit for this schematic and perform a timing simulation.

7.28 Repeat problem 7.27 using the control circuit in Figure 7.63.

7.29 Modify the code in Figure 7.71 to use the control circuit in Figure 7.63. Synthesize the code for implementation in a chip and perform a timing simulation.

7.30 In section 7.14.2 we designed a processor that performs the operations listed in Table 7.3. Design a modified circuit that performs an additional operation Swap *Rx*, *Ry*. This operation

swaps the contents of registers *Rx* and *Ry*. Use three bits $f_2 f_1 f_0$ to represent the input *F* shown in Figure 7.75 because there are now five operations, rather than four. Add a new register, named *Tmp*, into the system, to be used for temporary storage during the swap operation. Show logic expressions for the outputs of the control circuit, as was done in section 7.14.2.

7.31 A ring oscillator is a circuit that has an odd number, *n*, of inverters connected in a ringlike structure, as shown in Figure P7.5. The output of each inverter is a periodic signal with a certain period.
(a) Assume that all the inverters are identical; hence they all have the same delay, t_p. Let the output of one of the inverters be named *f*. Give an equation that expresses the period of the signal *f* in terms of *n* and t_p.
(b) For this part you are to design a circuit that can be used to experimentally measure the delay t_p through one of the inverters in the ring oscillator. Assume the existence of an input called *Reset* and another called *Interval*. The timing of these two signals is shown in Figure P7.6. The length of time for which *Interval* has the value 1 is known. Assume that this length of time is 100 ns. Design a circuit that uses the *Reset* and *Interval* signals and the signal *f* from part (*a*) to experimentally measure t_p. In your design you may use logic gates and subcircuits such as adders, flip-flops, counters, registers, and so on.

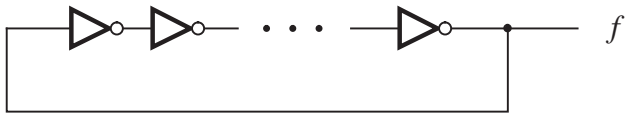


Figure P7.5 A ring oscillator.

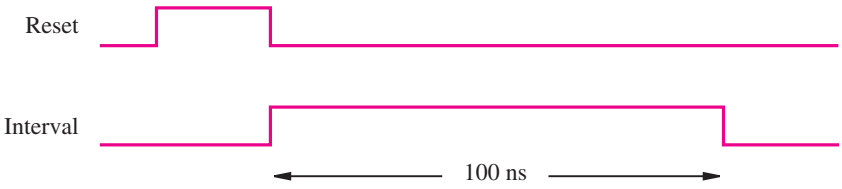


Figure P7.6 Timing of signals for problem 7.31.

7.32 A circuit for a gated D latch is shown in Figure P7.7. Assume that the propagation delay through either a NAND gate or an inverter is 1 ns. Complete the timing diagram given in the figure, which shows the signal values with 1 ns resolution.

7.33 A logic circuit has two inputs, *Clock* and *Start*, and two outputs, *f* and *g*. The behavior of the circuit is described by the timing diagram in Figure P7.8. When a pulse is received on the *Start* input, the circuit produces pulses on the *f* and *g* outputs as shown in the

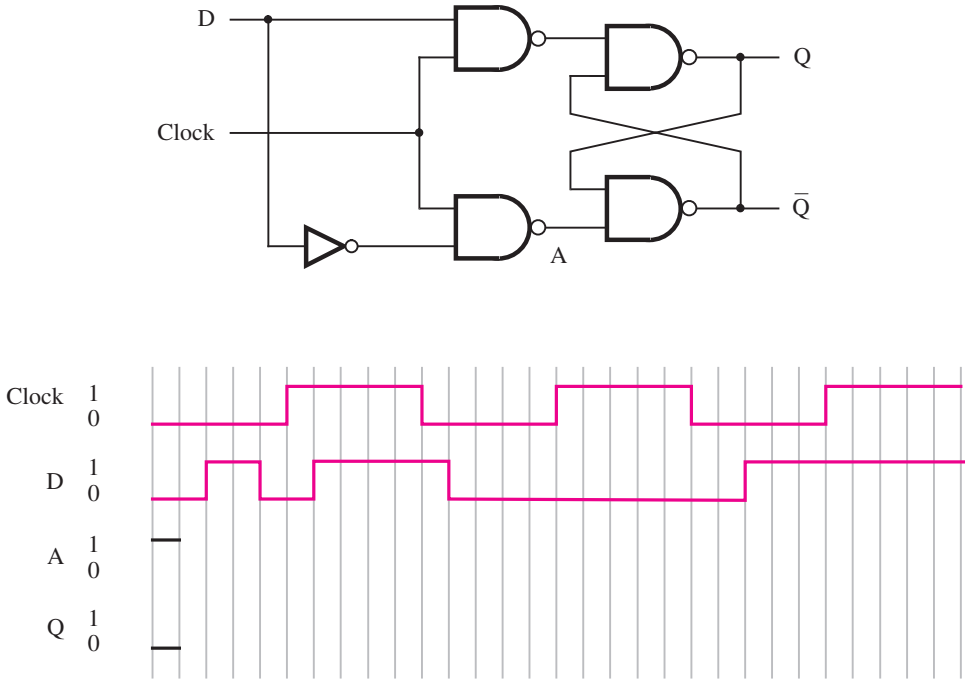


Figure P7.7 Circuit and timing diagram for problem 7.32.

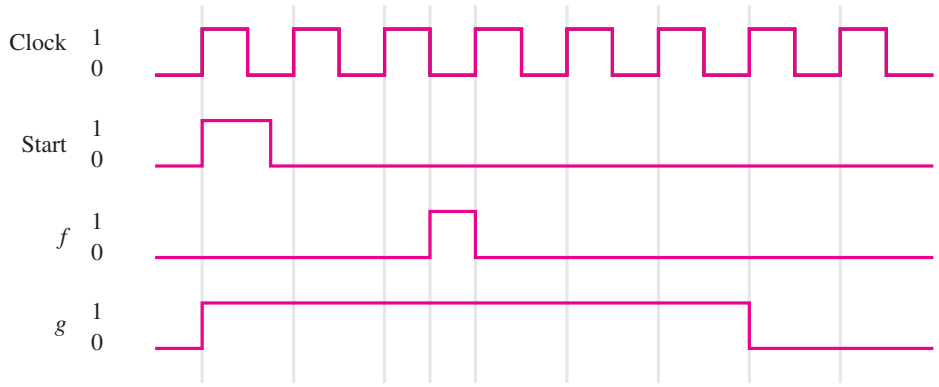


Figure P7.8 Timing diagram for problem 7.33.

timing diagram. Design a suitable circuit using only the following components: a three-bit resettable positive-edge-triggered synchronous counter and basic logic gates. For your answer assume that the delays through all logic gates and the counter are negligible.

7.34 The following code checks for adjacent ones in an n-bit vector.

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

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

7.35 The Verilog code in Figure P7.9 represents a 3-bit *linear-feedback shift register* (LFSR). This type of circuit generates a counting sequence of pseudo-random numbers that repeats after $2^n - 1$ clock cycles, where n is the number of flip-flops in the LFSR. Synthesize a circuit to implement the LFSR in a chip. Draw a diagram of the circuit. Simulate the circuit's behavior by loading the pattern 001 into the LFSR and then enabling the register to count. What is the counting sequence?

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

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

endmodule
```

Figure P7.9 Code for a linear-feedback shift register.

7.36 Repeat problem 7.35 for the Verilog code in Figure P7.10.

7.37 The Verilog code in Figure P7.11 is equivalent to the code in Figure P7.9, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

7.38 The Verilog code in Figure P7.12 is equivalent to the code in Figure P7.10, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

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

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

endmodule
```

Figure P7.10 Code for a linear-feedback shift register.

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

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

endmodule
```

Figure P7.11 Code for problem 7.37.

7.39 A universal shift register can shift in both the left-to-right and right-to-left directions, and it has parallel-load capability. Draw a circuit for such a shift register.

7.40 Write Verilog code for a universal shift register with n bits.

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

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

endmodule
```

Figure P7.12 Code for problem 7.38.

REFERENCES

1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2nd ed., (Morgan Kaufmann: San Francisco, CA, 1998).
3. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. R. H. Katz, *Contemporary Logic Design*, (Benjamin/Cummings: Redwood City, CA, 1994).
8. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed., (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, (Prentice-Hall: Englewood Cliffs, NJ, 1990).

11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. Institute of Electrical and Electronics Engineers, *IEEE Standard Verilog Hardware Description Language Reference Manual*, (IEEE: Piscataway, NJ, 1995).
13. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed., (Kluwer: Norwell, MA, 1998).
14. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, (Prentice-Hall: Upper Saddle River, NJ, 1996).
15. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
16. Z. Navabi, *Verilog Digital System Design*, (McGraw-Hill: New York, 1999).
17. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
18. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
19. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).