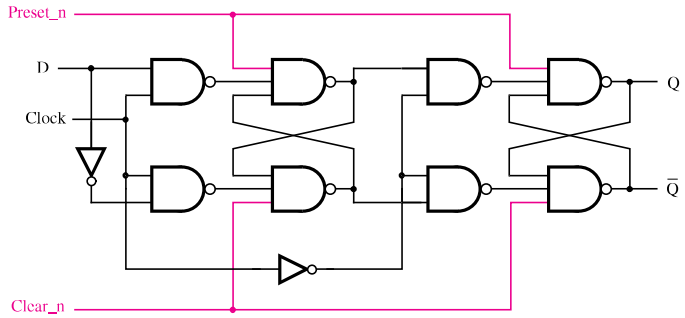
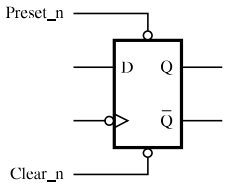


(a) Circuit



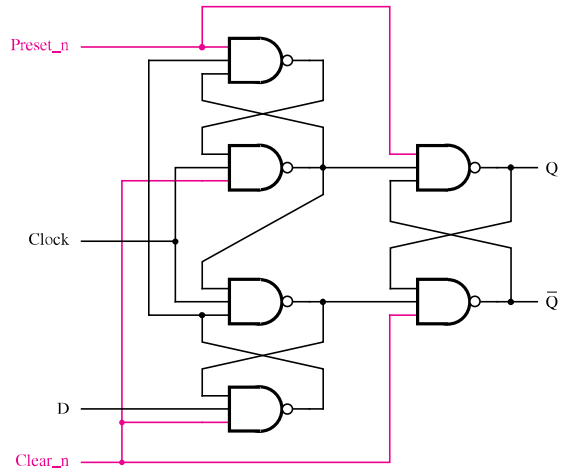
(b) Graphical symbol

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

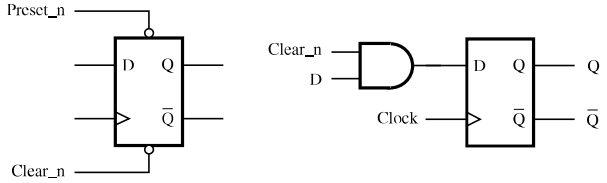
Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have Q = 0. It is equally useful to be able to preset each flip-flop to Q = 1, to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures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

Figure 5.12*a* shows an implementation of the circuit in Figure 5.9*a* using NAND gates. The master stage is just the gated D latch of Figure 5.7*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 5.6. This eliminates one NOT gate from the circuit.

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



(a) Circuit



(b) Graphical symbol (c) Adding a synchronous clear

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

Clear_n and *Preset_n* to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 5.12*b*.

A similar modification can be done on the edge-triggered flip-flop of Figure 5.11*a*, as indicated in Figure 5.13*a*. Again, both *Clear_n* and *Preset_n* inputs are active low. They do not disturb the flip-flop when they are equal to 1.

In the circuits in Figures 5.12*a* and 5.13*a*, the effect of a low signal on either the *Clear_n* or *Preset_n* input is immediate. For example, if *Clear_n* = 0 then the flip-flop goes into

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

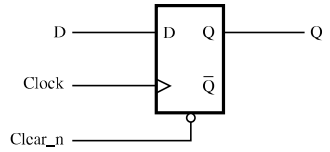
5.4.4 FLIP-FLOP TIMING PARAMETERS

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

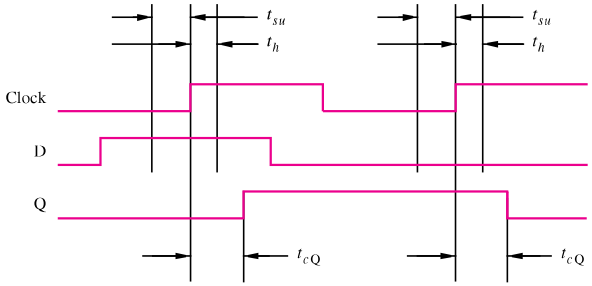
As indicated in Figure 5.14, a clock-to-Q propagation delay, t_{cQ}, is incurred before the value of Q changes after a positive clock edge. In general, the delay may not be exactly the same for the cases when Q changes from 1 to 0 or 0 to 1, but we assume for simplicity that these delays are equal. For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

5.5 T FLIP-FLOP

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 5.15*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 $T = 0$, 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 $T = 1$.



(a) D flip-flop with asynchronous clear



(b) Timing diagram

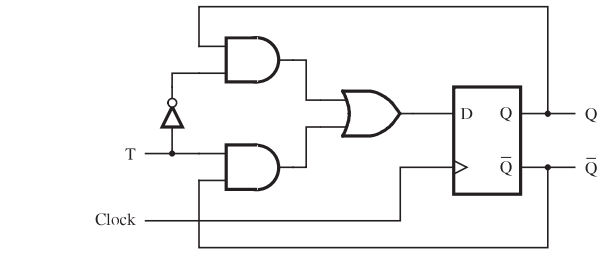
Figure 5.14 Flip-flop timing parameters.

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

5.6 JK FLIP-FLOP

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

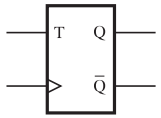
$$D = J\overline{Q} + \overline{K}Q$$



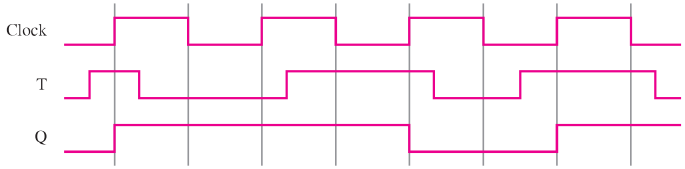
(a) Circuit

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

(b) Characteristic table



(c) Graphical symbol

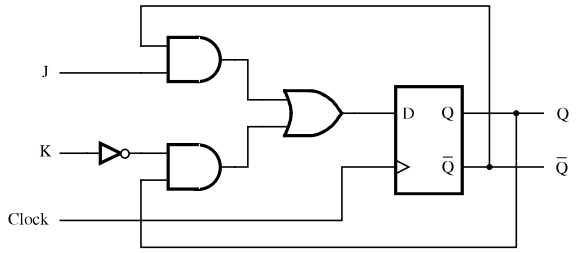


(d) Timing diagram

Figure 5.15 T flip-flop.

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

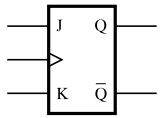
The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the J and K inputs together.



(a) Circuit

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

(b) Characteristic table



(c) Graphical symbol

Figure 5.16 JK flip-flop.

5.7 SUMMARY OF TERMINOLOGY

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

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

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

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

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

5.8 REGISTERS

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

5.8.1 SHIFT REGISTER

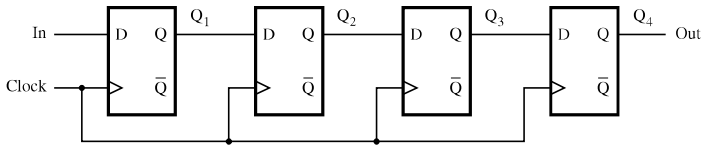
In Section 3.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to the right. A register that provides the ability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

Figure 5.17*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 5.17*b*, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.

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

5.8.2 PARALLEL-ACCESS SHIFT REGISTER

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



(a) Circuit

	In	Q_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 5.17 A simple shift register.

Figure 5.18 shows a four-bit shift register that provides the parallel access. A 2-to-1 multiplexer on its D input allows each flip-flop to be connected to two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal $\overline{Shift}/Load$ 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 5.18 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 5.18 can perform both of these functions.

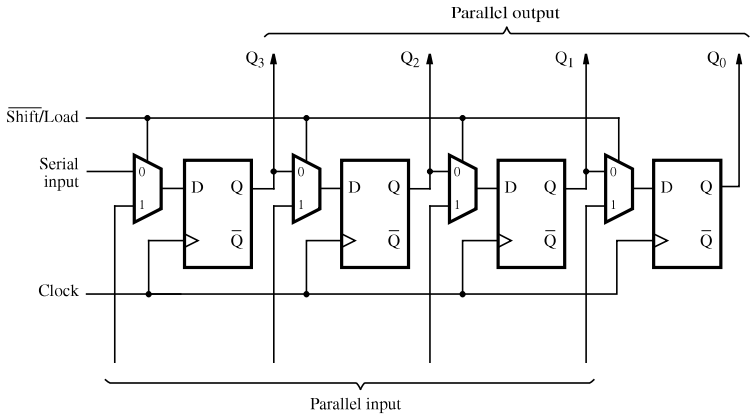


Figure 5.18 Parallel-access shift register.

5.9 COUNTERS

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

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

5.9.1 ASYNCHRONOUS COUNTERS

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

Up-Counter with T Flip-Flops

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

Figure 5.19*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

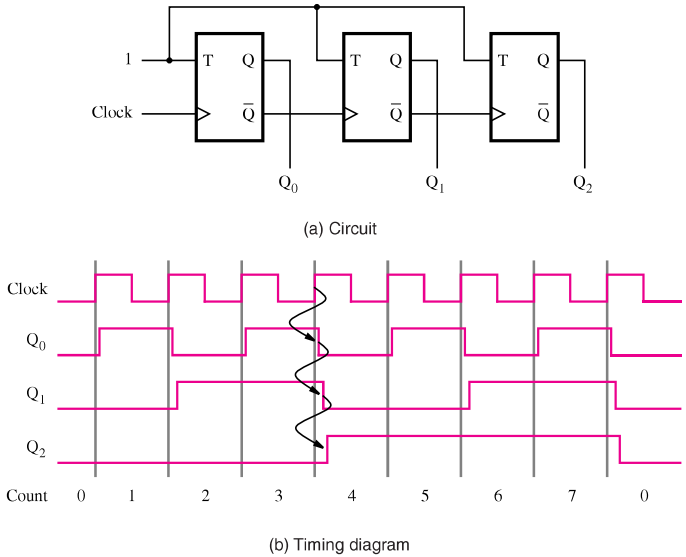


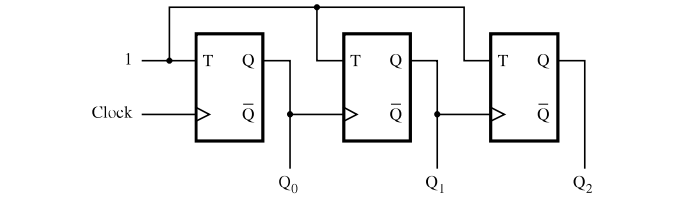
Figure 5.19 A three-bit up-counter.

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

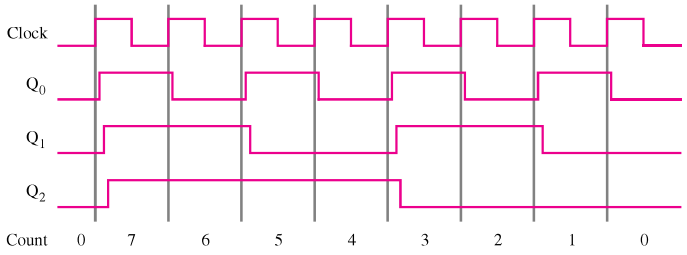
The counter in Figure 5.19*a* has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 5.19*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 3.5. The circuit in Figure 5.19*a* is an *asynchronous counter*, or a *ripple counter*.

Down-Counter with T Flip-Flops

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



(a) Circuit



(b) Timing diagram

Figure 5.20 A three-bit down-counter.

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

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

5.9.2 SYNCHRONOUS COUNTERS

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

Synchronous Counter with T Flip-Flops

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

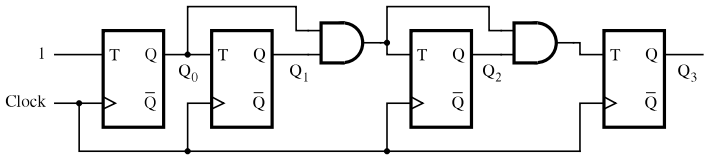
$$T_0 = 1$$
$$T_1 = Q_0$$
$$T_2 = Q_0Q_1$$

Table 5.1 Derivation of the synchronous up-counter.

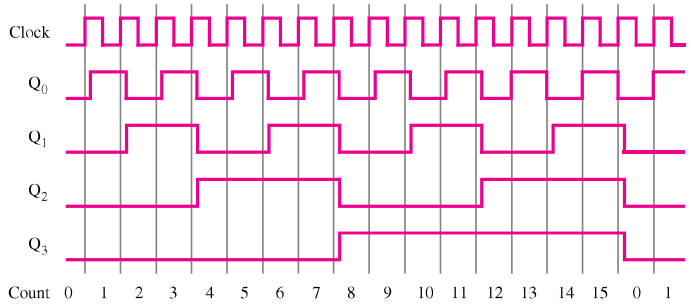
Clock cycle	Q_2 Q_1 Q_0	
0	0 0 0	
1	0 0 1	
2	0 1 0	Q_1 changes
3	0 1 1	
4	1 0 0	Q_1, Q_2 change
5	1 0 1	
6	1 1 0	Q_1 changes
7	1 1 1	
8	0 0 0	Q_1, Q_2 change

$$T_3 = Q_0Q_1Q_2$$
$$\vdots$$
$$T_n = Q_0Q_1 \cdots Q_{n-1}$$

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



(a) Circuit



(b) Timing diagram

Figure 5.21 A four-bit synchronous up-counter.

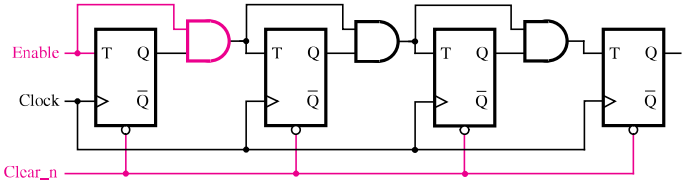


Figure 5.22 Inclusion of Enable and Clear capability.

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

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

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

Synchronous Counter with D Flip-Flops

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

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

$$D_0 = Q_0 \oplus 1 = \overline{Q}_0$$
$$D_1 = Q_1 \oplus Q_0$$

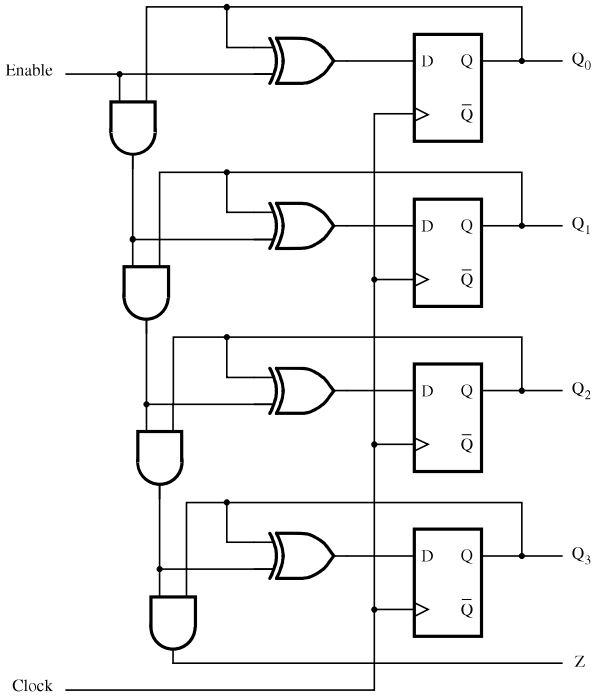


Figure 5.23 A four-bit counter with D flip-flops.

$$D_2 = Q_2 \oplus Q_1Q_0$$
$$D_3 = Q_3 \oplus Q_2Q_1Q_0$$

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 6.

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

$$D_0 = Q_0 \oplus Enable$$
$$D_1 = Q_1 \oplus Q_0 \cdot Enable$$
$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable$$
$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable$$

The operation of the counter is based on our observation for Table 5.1 that the state of the flip-flop in stage *i* changes only if all preceding flip-flops are in the state $Q = 1$. This makes the output of the AND gate that feeds stage *i* equal to 1, which causes the output of the XOR gate connected to D_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.

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

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

$$D = Q\overline{T} + \overline{Q}T$$
$$= Q \oplus T$$

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

5.9.3 COUNTERS WITH PARALLEL LOAD

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

The circuit of Figure 5.23 can be modified to provide the parallel-load capability as shown in Figure 5.24. A two-input multiplexer is inserted before each *D* input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when $Load = 0$. A new initial value, $D_3D_2D_1D_0$, is loaded into the counter when $Load = 1$.

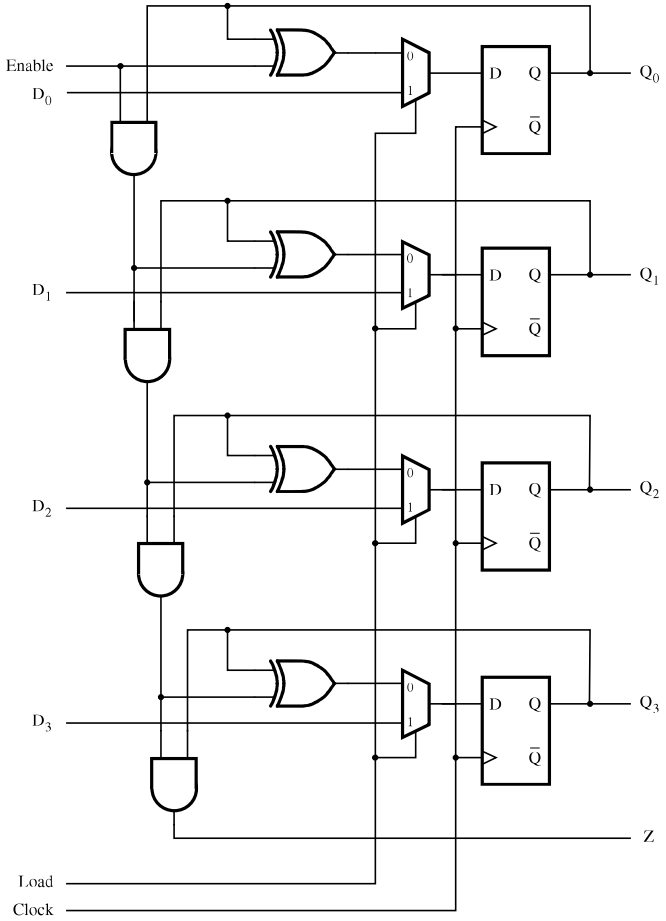
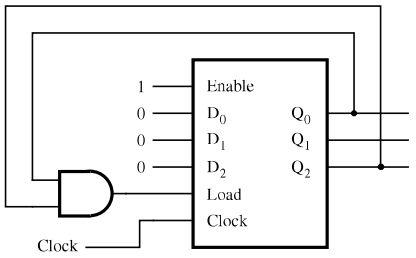


Figure 5.24 A counter with parallel-load capability.

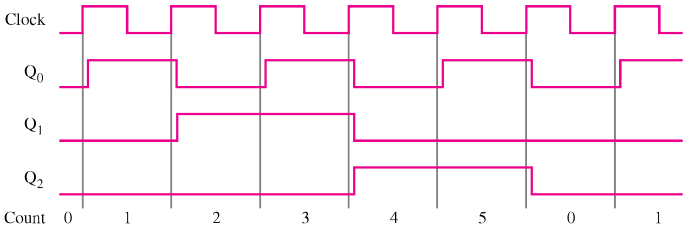
5.10 RESET SYNCHRONIZATION

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

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



(a) Circuit

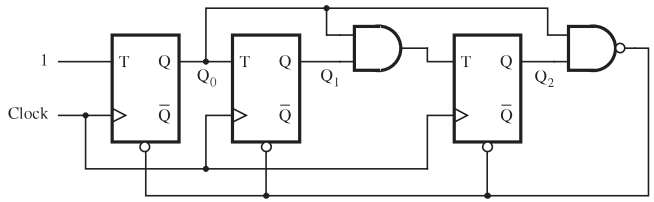


(b) Timing diagram

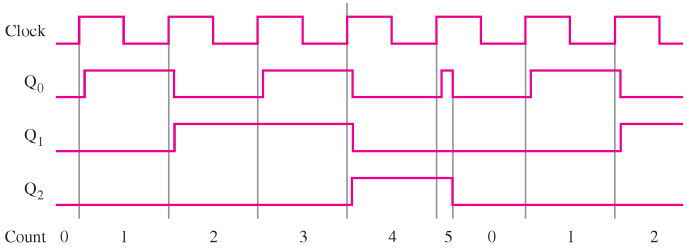
Figure 5.25 A modulo-6 counter with synchronous reset.

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

Consider now the possibility of using the clear feature of individual flip-flops, rather than the parallel-load approach. The circuit in Figure 5.26*a* illustrates one possibility. It uses the counter structure of Figure 5.21*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 5.26*b*. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the circuit, but not on the clock. Therefore, signal values $Q_2Q_1Q_0 = 101$ are maintained for a



(a) Circuit



(b) Timing diagram

Figure 5.26 A modulo-6 counter with asynchronous reset.

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

The timing diagrams in Figures 5.25*b* and 5.26*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.

5.11 OTHER TYPES OF COUNTERS

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

5.11.1 BCD COUNTER

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

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

5.11.2 RING COUNTER

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

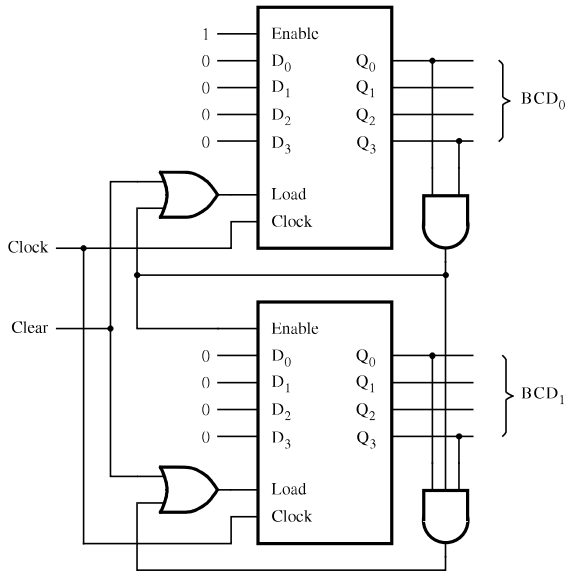
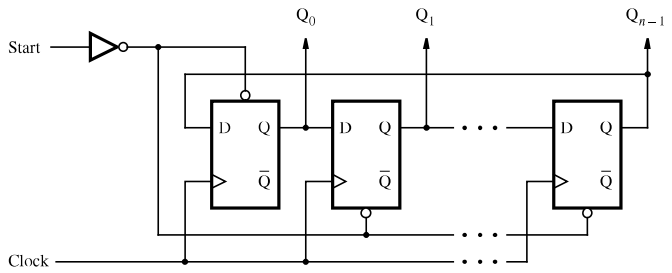


Figure 5.27 A two-digit BCD counter.

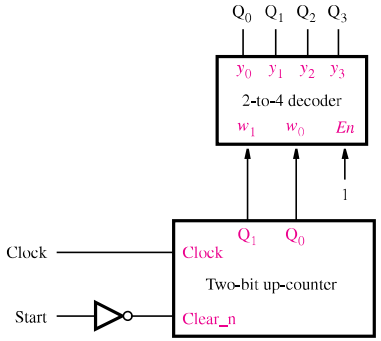
as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 5.25 through 5.27.

It is possible to devise a counterlike circuit in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, while for all other counts $Q_i = 0$. Then Q_i indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flips-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 5.28*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 4.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.

The circuit in Figure 5.28*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



(a) An n-bit ring counter



(b) A four-bit ring counter

Figure 5.28 Ring counter.

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

The circuit in Figure 5.28*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 4.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$, and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two.

5.11.3 JOHNSON COUNTER

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

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

5.11.4 REMARKS ON COUNTER DESIGN

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

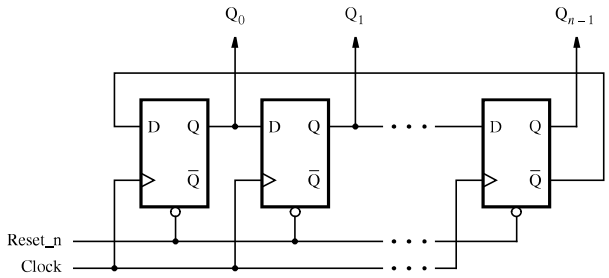


Figure 5.29 Johnson counter.

5.12 USING STORAGE ELEMENTS WITH CAD TOOLS

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

5.12.1 INCLUDING STORAGE ELEMENTS IN SCHEMATICS

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

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

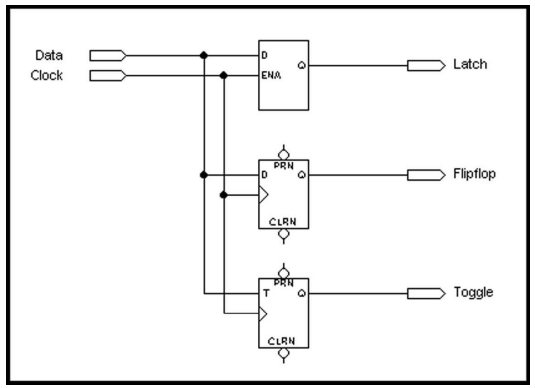


Figure 5.30 Three types of storage elements in a schematic.

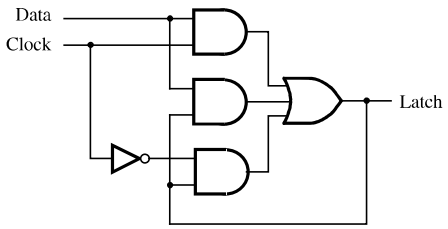


Figure 5.31 Gated D latch generated by CAD tools.

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

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

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

5.12.2 USING VERILOG CONSTRUCTS FOR STORAGE ELEMENTS

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

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

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

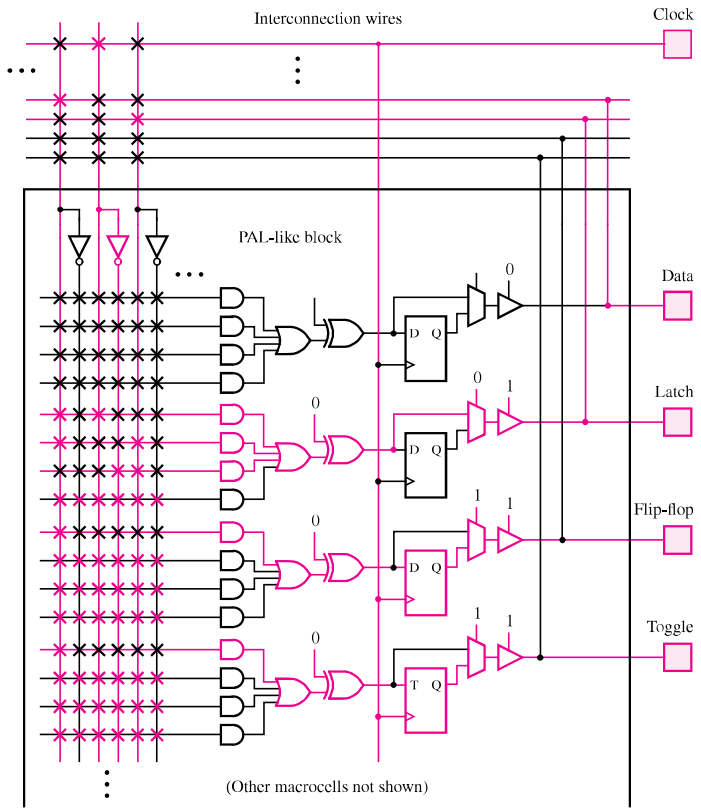



Figure 5.32 Implementation of the schematic in Figure 5.30 in a CPLD.

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

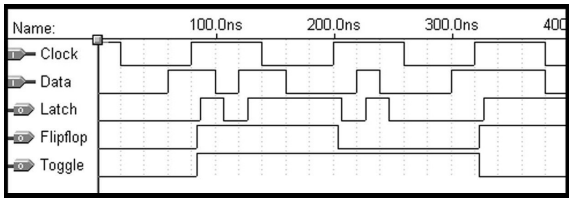


Figure 5.33 Timing simulation for the storage elements in Figure 5.30.

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

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

endmodule
```

Figure 5.34 Code for a gated D latch.

CODE FOR A GATED D LATCH The code in Figure 5.34 defines a module named *D_latch*, which has the inputs *D* and *Clk* and the output Q. The **if** clause defines that the Q output must take the value of *D* when $Clk = 1$. Since no **else** clause is given, a latch will be synthesized to maintain the value of Q when $Clk = 0$. Therefore, the code describes a gated D latch. The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the Q output. **Example 5.1**

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

CODE FOR A D FLIP-FLOP Figure 5.35 defines a module named *flipflop*, which is a positive-edge-triggered D flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. The keyword **posedge** specifies **Example 5.2**


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

   always @(posedge Clock)
      Q = D;

endmodule
```

Figure 5.35 Code for a D flip-flop.

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

5.12.3 BLOCKING AND NON-BLOCKING ASSIGNMENTS

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

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

or

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

or

```
Q = D;
```

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

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

```
Q1 = D;
```

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

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

endmodule
```

Figure 5.36 Incorrect code for two cascaded flip-flops.

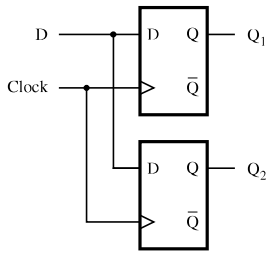


Figure 5.37 Circuit for Example 5.3.

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

$$Q2 = Q1;$$

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

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

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

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

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

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

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

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

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

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

endmodule
```

Figure 5.38 Code for two cascaded flip-flops.

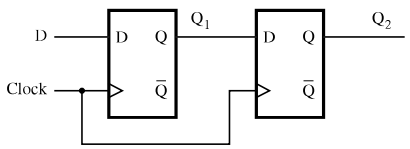


Figure 5.39 Circuit defined in Figure 5.38.

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

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

endmodule
```

Figure 5.40 Code for Example 5.5.

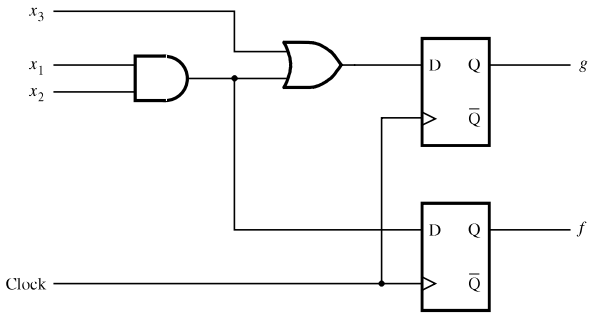


Figure 5.41 Circuit for Example 5.5.


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

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

endmodule
```

Figure 5.42 Code for Example 5.6.

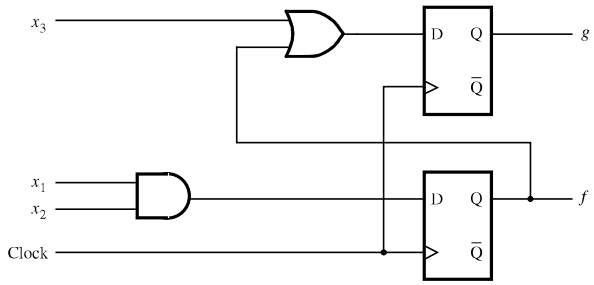


Figure 5.43 Circuit for Example 5.6.

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

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

5.12.4 NON-BLOCKING ASSIGNMENTS FOR COMBINATIONAL CIRCUITS

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

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

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

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

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

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

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

5.12.5 FLIP-FLOPS WITH CLEAR CAPABILITY

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

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

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

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

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

endmodule
```

Figure 5.44 D flip-flop with asynchronous reset.

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

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

endmodule
```

Figure 5.45 D flip-flop with synchronous reset.

5.13 USING VERILOG CONSTRUCTS FOR REGISTERS AND COUNTERS

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

AN N-BIT REGISTER Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed. The code for an n-bit register is given in Figure 5.46. The parameter n specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size. **Example 5.9**

A FOUR-BIT SHIFT REGISTER Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Figure 5.18. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the D input. Figure 5.47 defines the module named *muxdff*, which represents this subcircuit. The two data inputs are named D_0 and D_1, and they are selected using the *Sel* input. The **if-else** statement specifies that on the positive clock edge if *Sel* = 0, then Q is assigned the value of D_0; otherwise, Q is assigned the value of D_1. An alternative way of defining the same circuit is given in Figure 5.48. In this code, the conditional assignment statement specifies a 2-to-1 multiplexer with the output D, which is then connected to the flip-flop in the **always** block. **Example 5.10**

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

ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER A different style of code for the four-bit shift register is given in Figure 5.50. Instead of using subcircuits, the shift register is defined using the approach presented in Example 5.4. All actions take place at the positive edge of the clock. If $L = 1$, the register is loaded in parallel with the four bits of input R. If $L = 0$, the contents of the register are shifted to the right and the value of the input w is loaded into the most-significant bit Q_3. **Example 5.11**


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

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

endmodule
```

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

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

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

endmodule
```

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

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

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

   always @(posedge Clock)
      Q <= D;

endmodule
```

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

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

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

endmodule
```

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

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

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

endmodule
```

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

AN N-BIT SHIFT REGISTER Figure 5.51 shows the code that can be used to represent shift registers of any size. The parameter n, which has the default value 16 in the figure, sets the number of flip-flops. The code is identical to that in Figure 5.50 with two exceptions. First, R and Q are defined in terms of n. Second, the **else** clause that describes the shifting operation is generalized to work for any number of flip-flops by using a **for** loop. **Example 5.12**


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

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

endmodule
```

Figure 5.51 An n-bit shift register.

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

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

endmodule
```

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

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

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

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

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

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

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

endmodule
```

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


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

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

endmodule
```

Figure 5.54 A down-counter with a parallel load.

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

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

endmodule
```

Figure 5.55 Code for an up/down counter.

5.13.1 FLIP-FLOPS AND REGISTERS WITH ENABLE INPUTS

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

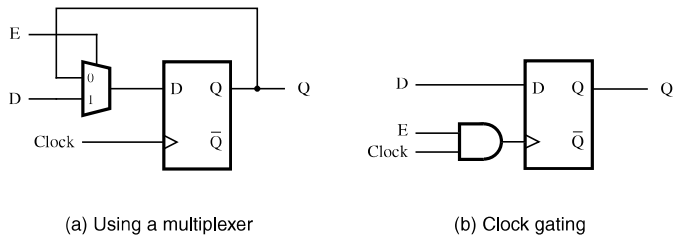


Figure 5.56 Providing an enable input for a D flip-flop.

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

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

endmodule
```

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

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

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


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

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

endmodule
```

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

5.13.2 SHIFT REGISTERS WITH ENABLE INPUTS

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

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

5.14 DESIGN EXAMPLE

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

5.14.1 REACTION TIMER

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

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

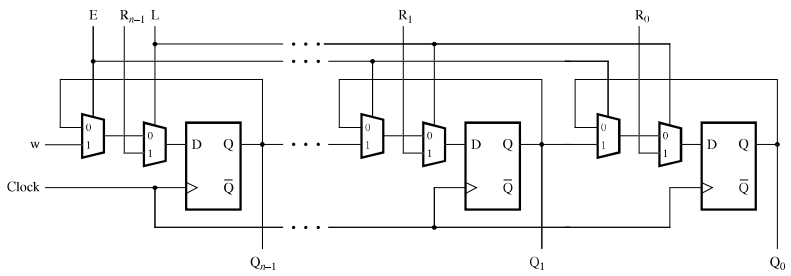


Figure 5.59 A shift register with parallel-load and enable control inputs.


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

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

endmodule
```

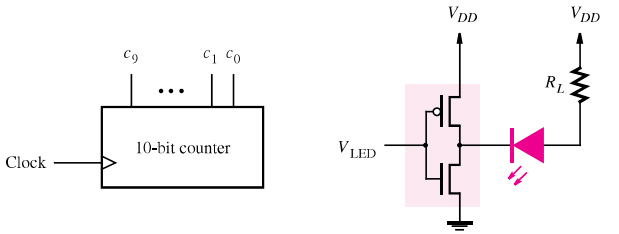
Figure 5.60 A left-to-right shift register with an enable input.

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

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

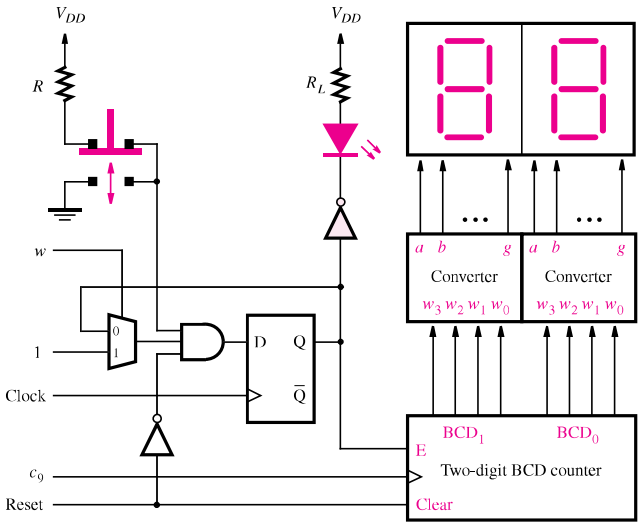
Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a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



(a) Clock divider

(b) LED circuit



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

Figure 5.61 A reaction-timer circuit.

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

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

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

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

Verilog Code

To describe the circuit in Figure 5.61*c* using Verilog code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. Code for the BCD counter, which represents the circuit in Figure 5.27, is shown in Figure 5.62. The two-digit BCD output is represented by the 2 four-bit signals *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 5.63 gives the code for the BCD-to-7-segment decoder.

Figure 5.64 shows the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit*1 and *Digit*0.

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

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

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

endmodule
```

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


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

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

endmodule
```

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

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

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

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

endmodule
```

Figure 5.64 Code for the reaction timer.

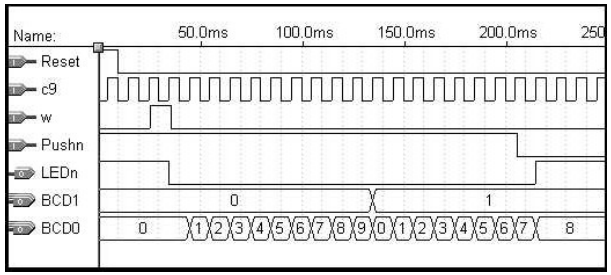


Figure 5.65 Simulation of the reaction timer circuit.

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

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

5.14.2 REGISTER TRANSFER LEVEL (RTL) CODE

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

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

5.15 TIMING ANALYSIS OF FLIP-FLOP CIRCUITS

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

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

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

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

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

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

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

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

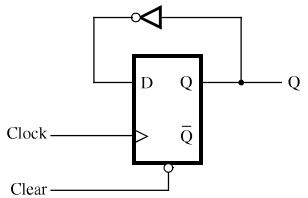


Figure 5.66 A simple flip-flop circuit.

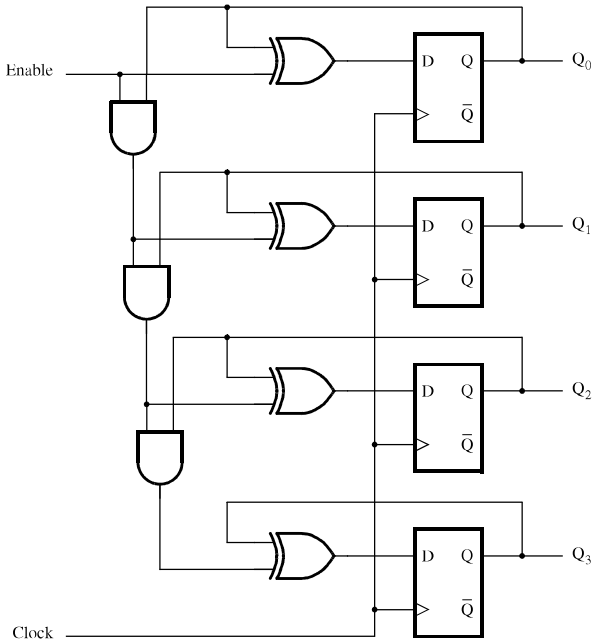


Figure 5.67 A 4-bit counter.

account for the setup time of flip-flop Q_3. This gives

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

Using the maximum value of t_{cQ} gives

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

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

5.15.1 TIMING ANALYSIS WITH CLOCK SKEW

In the above analysis we assumed that the clock signal arrived at exactly the same time at all four flip-flops. We will now repeat this analysis assuming that the clock signal still arrives at flip-flops Q_0, Q_1, and Q_2 simultaneously, but that there is a delay in the arrival of the clock signal at flip-flop Q_3. Such a variation in the arrival time of a clock signal at different flip-flops is called *clock skew*, t_{skew}, and can be caused by a number of factors.

In Figure 5.67 the critical path through the circuit is from flip-flop Q_0 to Q_3. However, the clock skew at Q_3 has the effect of reducing this delay, because it provides additional time before data is loaded into this flip-flop. Taking a clock skew of 1.5 ns into account, the delay of the path from flip-flop Q_0 to Q_3 is given by $t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su} - t_{skew} = 6.4 - 1.5$ ns $= 4.9$ ns. There is now a different critical path through the circuit, which starts at flip-flop Q_0 and ends at Q_2. The delay of this path gives

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

In this case the clock skew results in an increase in the circuit's maximum clock frequency. But if the clock skew had been negative, which would be the case if the clock signal arrived earlier at flip-flop Q_3 than at other flip-flops, then the result would have been a reduced F_{max}.

Since the loading of data into flip-flop Q_3 is delayed by the clock skew, it has the effect of increasing the hold time requirement of this flip-flop to $t_h + t_{skew}$, for all paths that end at Q_3 but start at Q_0, Q_1, or Q_2. The shortest such path in the circuit is from flip-flop Q_2 to Q_3 and has the delay $t_{cQ} + t_{AND} + t_{XOR} = 0.8 + 1.2 + 1.2 = 3.2$ ns. Since 3.2 ns $> t_h + t_{skew} = 1.9$ ns there is no hold time violation.

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

Example 5.17

Consider the circuit in Figure 5.68. In this circuit, there is a path that starts at flip-flop Q_1, passes through some network of logic gates, and ends at the D input of flip-flop Q_2. As indicated in the figure, different delays may be incurred before the clock signal reaches the flip-flops. Let Δ_1 and Δ_2 be the clock-signal delays for flip-flops Q_1 and Q_2, respectively. The clock skew between these two flip-flops is then defined as

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

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

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

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

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

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

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

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

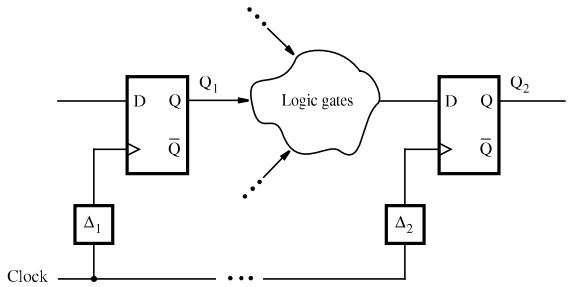


Figure 5.68 A general example of clock skew.

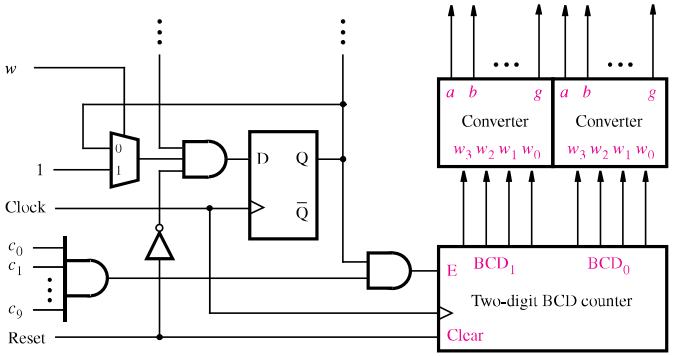


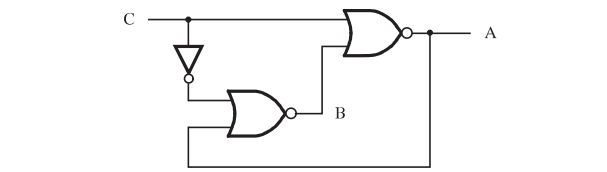
Figure 5.69 A modified version of the reaction-timer circuit.

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

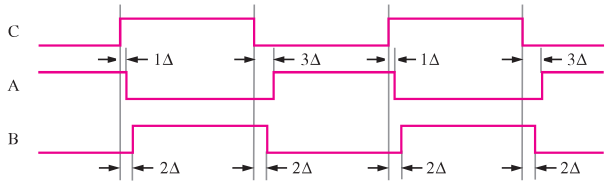
In the circuit of Figure 5.69 all flip-flops are clocked directly by the *Clock* signal. Therefore, the F_{max} and hold time analysis can now be applied. In general, it is a good design approach for sequential circuits to connect the clock inputs of all flip-flops to a common clock. We discuss such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

5.16 CONCLUDING REMARKS

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



(a) Circuit



(b) Timing diagram

Figure 5.70 Circuit for Example 5.18.

5.17 EXAMPLES OF SOLVED PROBLEMS

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

Example 5.18

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

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

Example 5.19

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

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

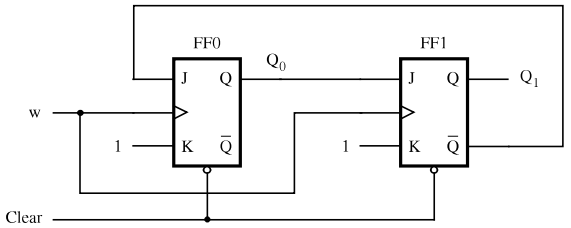


Figure 5.71 Circuit for Example 5.19.

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

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

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

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

Example 5.20

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

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

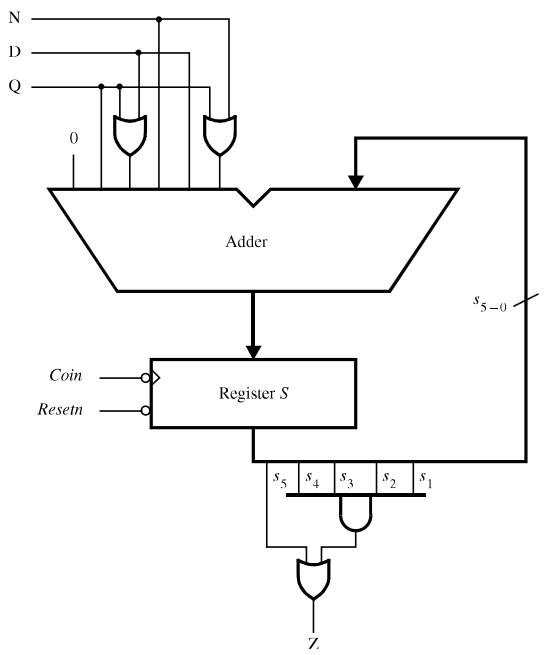


Figure 5.73 Circuit for Example 5.20.

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

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

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

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


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

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

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

endmodule
```

Figure 5.74 Code for Example 5.21.

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

Solution: Figure 5.74 gives the desired code.

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

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

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

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

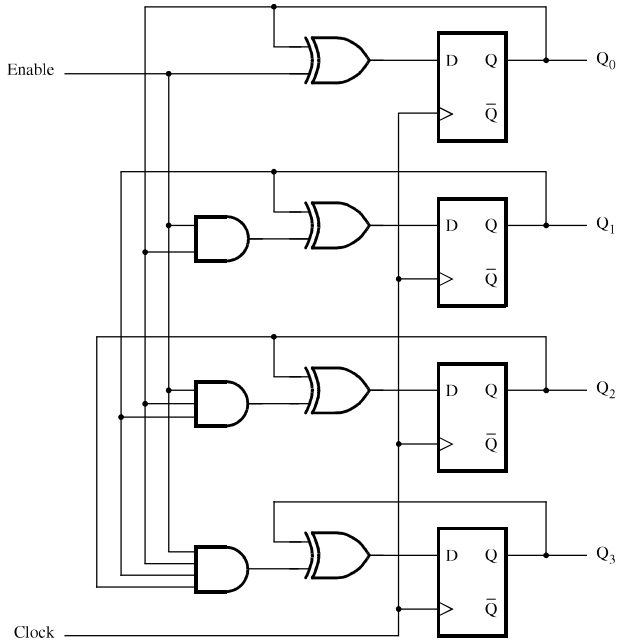


Figure 5.75 A faster 4-bit counter.

Problem: In Example 5.17 we showed how to perform timing analysis when there may be different delays associated with the clock signal at each of the flip-flops in a circuit. The circuit in Figure 5.76 includes three flip-flops, Q_1, Q_2, and Q_3, with corresponding clock delays Δ_1, Δ_2, and Δ_3. The flip-flop timing parameters are $t_{su} = 0.6$ ns, $t_h = 0.4$ ns, and $0.8 \leq t_{cQ} \leq 1$ ns. Also, the delay through a logic gate is given by $1 + 0.1k$, where k is the number of inputs to the gate. **Example 5.23**

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

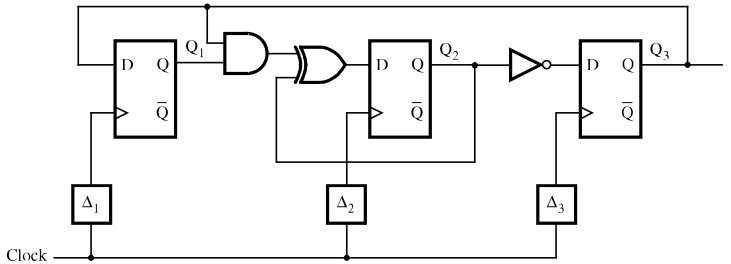


Figure 5.76 A circuit with clock skews.

Solution: For the values $\Delta_1 = \Delta_2 = \Delta_3 = 0$, there is no clock skew. Let the path delay between two flip-flops, Q_i and Q_j, be $T_{Q_i \to Q_j}$. The longest path delays are calculated by including the maximum value of t_{cQ} at flip-flop Q_i, and the setup time at flip-flop Q_j, as follows

$$T_{Q_1 \to Q_2} = t_{cQ} + t_{XOR} + t_{AND} + t_{su} = 1 + 1.2 + 1.2 + 0.6 = 4 \text{ ns}$$
$$T_{Q_2 \to Q_2} = t_{cQ} + t_{AND} + t_{su} = 1 + 1.2 + 0.6 = 2.8 \text{ ns}$$
$$T_{Q_2 \to Q_3} = t_{cQ} + t_{NOT} + t_{su} = 1 + 1.1 + 0.6 = 2.7 \text{ ns}$$
$$T_{Q_3 \to Q_1} = t_{cQ} + t_{su} = 1 + 0.6 = 1.6 \text{ ns}$$
$$T_{Q_3 \to Q_2} = t_{cQ} + t_{XOR} + t_{AND} + t_{su} = 1 + 1.2 + 1.2 + 0.6 = 4 \text{ ns}$$

Since the critical path delay $T_{Q_1 \to Q_2} = T_{Q_3 \to Q_2} = 4$ ns, then $F_{max} = 1/(4 \times 10^{-9}) = 250$ MHz.

For the values $\Delta_1 = \Delta_3 = 0$, and $\Delta_2 = 0.7$ ns, there is clock skew for the paths between flip-flops Q_1 and Q_2, as well as Q_3 and Q_2. Adjusting the longest delays calculated above to account for clock skew, we have

$$T_{Q_1 \to Q_2} = 4 - t_{skew} = 4 - (\Delta_2 - \Delta_1) = 4 - 0.7 = 3.3 \text{ ns}$$
$$T_{Q_2 \to Q_3} = 2.7 - t_{skew} = 2.7 - (\Delta_3 - \Delta_2) = 2.7 - (0 - 0.7) = 3.4 \text{ ns}$$
$$T_{Q_3 \to Q_2} = 4 - t_{skew} = 4 - (\Delta_2 - \Delta_3) = 4 - 0.7 = 3.3 \text{ ns}$$

The critical path delay now starts at flip-flop Q_2 and ends at flip-flop Q_3. Since $T_{Q_2 \to Q_3} = 3.4$ ns, then $F_{max} = 1/(3.4 \times 10^{-9}) = 294$ MHz.

For the values $\Delta_1 = 1$, $\Delta_2 = 0$, and $\Delta_3 = 0.5$ ns, there is clock skew for the paths between all flip-flops. Adjusting the longest delays calculated for the case with no clock skew, we have

$$T_{Q_1 \to Q_2} = 4 - t_{skew} = 4 - (\Delta_2 - \Delta_1) = 4 - (0 - 1) = 5 \text{ ns}$$
$$T_{Q_2 \to Q_3} = 2.7 - t_{skew} = 2.7 - (\Delta_3 - \Delta_2) = 2.7 - (0.5) = 2.2 \text{ ns}$$
$$T_{Q_3 \to Q_1} = 1.6 - t_{skew} = 1.6 - (\Delta_1 - \Delta_3) = 1.6 - (1 - 0.5) = 1.1 \text{ ns}$$
$$T_{Q_3 \to Q_2} = 4 - t_{skew} = 4 - (\Delta_2 - \Delta_3) = 4 - (0 - 0.5) = 4.5 \text{ ns}$$

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

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

$$T_{Q_1 \to Q_2} = t_{cQ} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns}$$
$$T_{Q_2 \to Q_2} = t_{cQ} + t_{AND} = 0.8 + 1.2 = 2 \text{ ns}$$
$$T_{Q_2 \to Q_3} = t_{cQ} + t_{NOT} = 0.8 + 1.1 = 1.9 \text{ ns}$$
$$T_{Q_3 \to Q_1} = t_{cQ} = 0.8 \text{ ns}$$
$$T_{Q_3 \to Q_2} = t_{cQ} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2 \text{ ns}$$

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

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

$$T_{Q_1 \to Q_2} = 3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_1) = 3.2 - (0 - 1) = 4.2 \text{ ns}$$
$$T_{Q_2 \to Q_3} = 1.9 - t_{skew} = 1.9 - (\Delta_3 - \Delta_2) = 1.9 - 0.5 = 1.4 \text{ ns}$$
$$T_{Q_3 \to Q_1} = 0.8 - t_{skew} = 0.8 - (\Delta_1 - \Delta_3) = 0.8 - (1 - 0.5) = 0.3 \text{ ns}$$
$$T_{Q_3 \to Q_2} = 3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_3) = 3.2 - (0 - 0.5) = 3.7 \text{ ns}$$

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

PROBLEMS

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

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

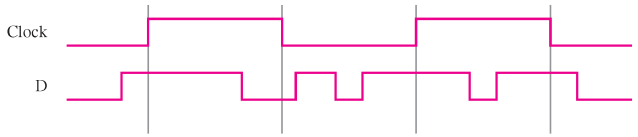


Figure P5.1 Timing diagram for Problem 5.1.

5.2 Figure 5.4 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its characteristic table and show its timing diagram.

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

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

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

5.6 The gated SR latch in Figure 5.5*a* has unpredictable behavior if the *S* and *R* inputs are both equal to 1 when the *Clk* changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition $S = R = 1$ causes the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.

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

***5.8** Consider the circuit in Figure P5.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?

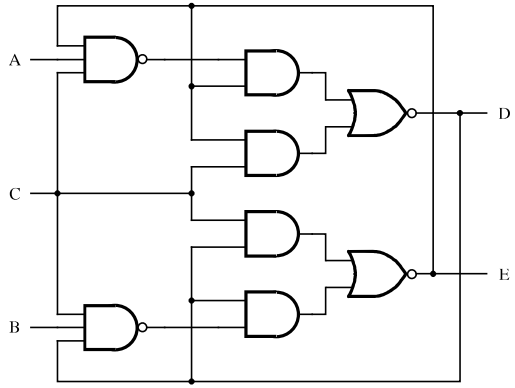


Figure P5.2 Circuit for Problem 5.8.

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

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

5.11 Synthesize a circuit for the code written for Problem 5.10 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.

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

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

5.14 Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in Section 5.9.3.

***5.15** Design a three-bit up/down counter using T flip-flops. It should include a control input called $\overline{\text{Up}}$/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.

5.16 Repeat Problem 5.15 using D flip-flops.

***5.17** The circuit in Figure P5.3 looks like a counter. What is the counting sequence of this circuit?

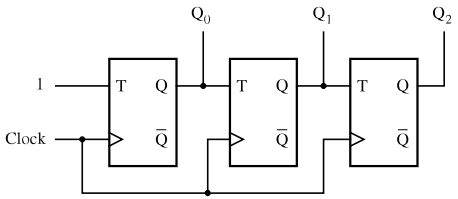


Figure P5.3 The circuit for Problem 5.17.

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

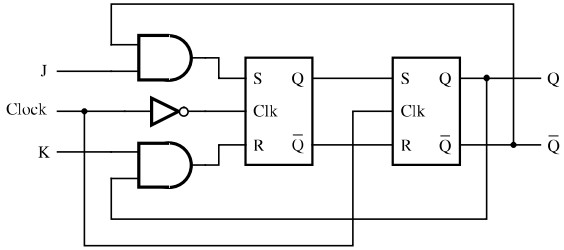


Figure P5.4 Circuit for Problem 5.18.

5.19 Construct a NOR-gate circuit, similar to the one in Figure 5.11a, which implements a negative-edge-triggered D flip-flop.

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

***5.21** For the flip-flops in the counter in Figure 5.24, assume that $t_{su} = 3$ ns, $t_h = 1$ ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?

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

5.23 Write Verilog code in the style shown in Figure 5.51 that represents a ring counter. Your code should have a parameter n that sets the number of flip-flops in the counter.

5.24 A ring oscillator is a circuit that has an odd number, n, of inverters connected in a ringlike structure, as shown in Figure P5.5. The output of each inverter is a periodic signal with a certain period.

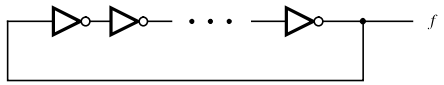


Figure P5.5 A ring oscillator.

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

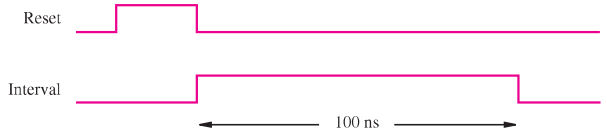
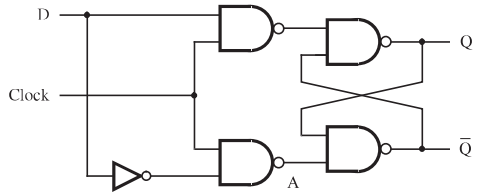


Figure P5.6 Timing of signals for Problem 5.24.

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



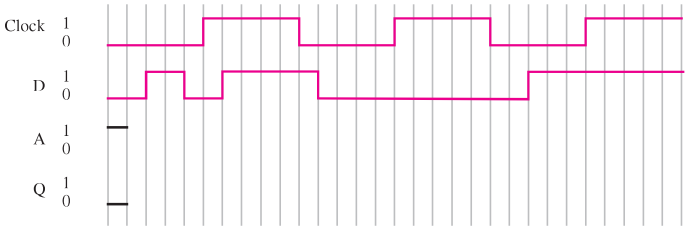


Figure P5.7 Circuit and timing diagram for Problem 5.25.

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

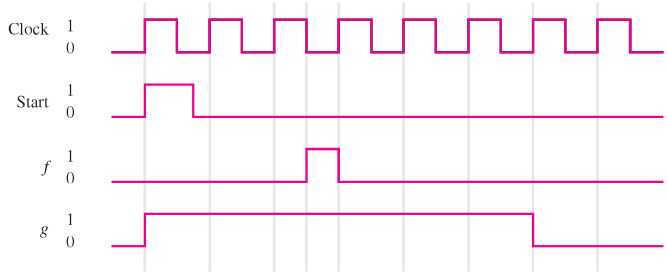


Figure P5.8 Timing diagram for Problem 5.26.

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

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

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

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

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

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

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

endmodule
```

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

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

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

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

endmodule
```

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

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

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

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

endmodule
```

Figure P5.11 Code for Problem 5.30.

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

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

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

endmodule
```

Figure P5.12 Code for Problem 5.31.

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

REFERENCES

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

This page intentionally left blank

chapter

6

SYNCHRONOUS SEQUENTIAL CIRCUITS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Design techniques for circuits that use flip-flops
- The concept of states and their implementation with flip-flops
- Synchronous control by using a clock signal
- Sequential behavior of digital circuits
- A complete procedure for designing synchronous sequential circuits
- Verilog specification of sequential circuits
- The concept of finite state machines

In preceding chapters we considered combinational logic circuits in which outputs are determined fully by the present values of inputs. We also discussed how simple storage elements can be implemented in the form of flip-flops. The output of a flip-flop depends on the state of the flip-flop rather than the value of its inputs at any given time; the inputs cause changes in the state.

In this chapter we deal with a general class of circuits in which the outputs depend on the past behavior of the circuit, as well as on the present values of inputs. They are called *sequential circuits*. In most cases a clock signal is used to control the operation of a sequential circuit; such a circuit is called a *synchronous sequential circuit*. The alternative, in which no clock signal is used, is called an *asynchronous sequential circuit*. Synchronous circuits are easier to design and are used in a vast majority of practical applications; they are the topic of this chapter. Asynchronous circuits will be discussed in Chapter 9.

Synchronous sequential circuits are realized using combinational logic and one or more flip-flops. The general structure of such a circuit is shown in Figure 6.1. The circuit has a set of primary inputs, W, and produces a set of outputs, Z. The stored values in the flip-flops are referred to as the *state*, Q, of the circuit. Under control of the clock signal, the flip-flops change their state as determined by the combinational logic that feeds the inputs of these flip-flops. Thus the circuit moves from one state to another. To ensure that only one transition from one state to another takes place during one clock cycle, the flip-flops have to be of the edge-triggered type. They can be triggered either by the positive (0 to 1 transition) or by the negative (1 to 0 transition) edge of the clock. We will use the term *active clock edge* to refer to the clock edge that causes the change in state.

The combinational logic that provides the input signals to the flip-flops has two sources: the primary inputs, W, and the present (current) state of the flip-flops, Q. Thus changes in state depend on both the present state and the values of the primary inputs.

Figure 6.1 indicates that the outputs of the sequential circuit are generated by another combinational circuit, such that the outputs are a function of the present state of the flip-flops and of the primary inputs. Although the outputs always depend on the present state, they do not necessarily have to depend directly on the primary inputs. Thus the connection shown in blue in the figure may or may not exist. To distinguish between these two possibilities, it is customary to say that sequential circuits whose outputs depend only on the state of the circuit are of *Moore* type, while those whose outputs depend on both the state and the primary inputs are of *Mealy* type. These names are in honor of Edward Moore and George Mealy, who investigated the behavior of such circuits in the 1950s.

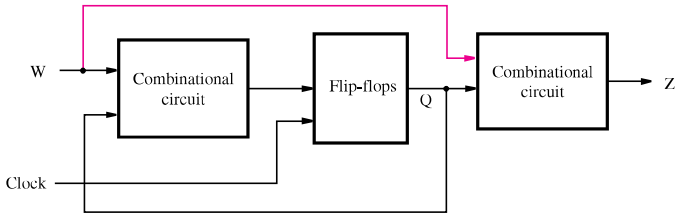


Figure 6.1 The general form of a sequential circuit.

Sequential circuits are also called *finite state machines (FSMs)*, which is a more formal name that is often found in technical literature. The name derives from the fact that the functional behavior of these circuits can be represented using a finite number of states. In this chapter we will often use the term *finite state machine*, or simply *machine*, when referring to sequential circuits.

6.1 BASIC DESIGN STEPS

Sequential circuits are often used to control the operation of physical systems. We will introduce the techniques for designing such circuits by means of a simple example.

Consider an application where the speed of an automatically-controlled vehicle has to be regulated as follows. The vehicle is designed to run at some predetermined speed. However, due to some operational conditions the speed may exceed the desirable limit, in which case the vehicle has to be slowed down. To determine when such action is needed, the speed is measured at regular intervals. Let a binary signal w indicate whether the speed exceeds the required limit, such that $w = 0$ means that the speed is within acceptable range and $w = 1$ indicates excessive speed. The desired control strategy is that if $w = 1$ during two or more consecutive measurements, a control signal z must be asserted to cause the vehicle to slow down. Thus, $z = 0$ allows the current speed to be maintained, while $z = 1$ reduces the speed. Let a signal *Clock* define the required timing intervals, such that the speed is measured once during each clock cycle. Therefore, we wish to design a circuit that meets the following specification:

1. The circuit has one input, w, and one output, z.
2. All changes in the circuit occur on the positive edge of the clock signal.
3. The output z is equal to 1 if during two immediately preceding clock cycles the input w was equal to 1. Otherwise, the value of z is equal to 0.

From this specification it is apparent that the output z depends on both present and past values of w. Consider the sequence of values of the w and z signals for the clock cycles shown in Figure 6.2. The values of w are assumed to be generated by the vehicle being controlled; the values of z correspond to our specification. Since $w = 1$ in clock cycle t_3 and remains equal to 1 in cycle t_4, then z should be set to 1 in cycle t_5. Similarly, z should be set to 1 in cycle t_8 because $w = 1$ in cycles t_6 and t_7, and it should also be set to 1 in t_9 because $w = 1$ in t_7 and t_8. As seen in the figure. for a given value of input w the output z may be either 0 or 1. For example, $w = 0$ during clock cycles t_2 and t_5, but $z = 0$ during t_2 and $z = 1$ during t_5. Similarly, $w = 1$ during t_1 and t_8, but $z = 0$ during t_1 and $z = 1$ during t_8. This means that z is not determined only by the present value of w, so there must exist different states in the circuit that determine the value of z.

6.1.1 STATE DIAGRAM

The first step in designing a finite state machine is to determine how many states are needed and which transitions are possible from one state to another. There is no set procedure for

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Figure 6.2 Sequences of input and output signals.

this task. The designer must think carefully about what the machine has to accomplish. A good way to begin is to select one particular state as a *starting* state; this is the state that the circuit should enter when power is first turned on or when a *reset* signal is applied. For our example let us assume that the starting state is called state A. As long as the input w is 0, the circuit need not do anything, and so each active clock edge should result in the circuit remaining in state A. When w becomes equal to 1, the machine should recognize this, and move to a different state, which we will call state B. This transition takes place on the next active clock edge after w has become equal to 1. In state B, as in state A, the circuit should keep the value of output z at 0, because it has not yet seen $w = 1$ for two consecutive clock cycles. When in state B, if w is 0 at the next active clock edge, the circuit should move back to state A. However, if $w = 1$ when in state B, the circuit should change at the next active clock edge to a third state, called C, and it should then generate an output $z = 1$. The circuit should remain in state C as long as $w = 1$ and should continue to maintain $z = 1$. When w becomes 0, the machine should move back to state A. Since the preceding description handles all possible values of input w that the machine can encounter in its various states, we can conclude that three states are needed to implement the desired machine.

Now that we have determined in an informal way the possible transitions between states, we will describe a more formal procedure that can be used to design the corresponding sequential circuit. Behavior of a sequential circuit can be described in several different ways. The conceptually simplest method is to use a pictorial representation in the form of a *state diagram*, which is a graph that depicts states of the circuit as nodes (circles) and transitions between states as directed arcs. The state diagram in Figure 6.3 defines the behavior that corresponds to our specification. States A, B, and C appear as nodes in the diagram. Node A represents the starting state, and it is also the state that the circuit will reach *after* an input $w = 0$ is applied. In this state the output z should be 0, which is indicated as $A/z = 0$ in the node. The circuit should remain in state A as long as $w = 0$, which is indicated by an arc with a label $w = 0$ that originates and terminates at this node. The first occurrence of $w = 1$ (following the condition $w = 0$) is recorded by moving from state A to state B. This transition is indicated on the graph by an arc originating at A and terminating at B. The label $w = 1$ on this arc denotes the input value that causes the transition. In state B the output remains at 0, which is indicated as $B/z = 0$ in the node.

When the circuit is in state B, it will change to state C if w is still equal to 1 at the next active clock edge. In state C the output z becomes equal to 1. If w stays at 1 during subsequent clock cycles, the circuit will remain in state C maintaining $z = 1$. However, if w becomes 0 when the circuit is either in state B or in state C, the next active clock edge will cause a transition to state A to take place.

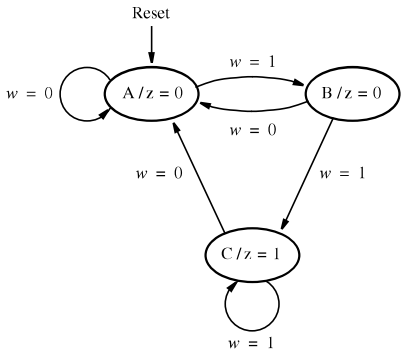


Figure 6.3 State diagram of a simple sequential circuit.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

Figure 6.4 State table corresponding to Figure 6.3.

In the diagram we indicated that the *Reset* input is used to force the circuit into state *A*. We could treat *Reset* as just another input to the circuit, and show a transition from each state to the starting state *A* under control of the input *Reset*. This would complicate the diagram unnecessarily. Instead, we use a single arrow with the *Reset* label, as shown in Figure 6.3, to indicate that the *Reset* input causes a change to the starting state regardless of what state the circuit happens to be in.

6.1.2 STATE TABLE

Although the state diagram provides a description of the behavior of a sequential circuit that is easy to understand, to proceed with the implementation of the circuit it is convenient to translate the information contained in the state diagram into a tabular form. Figure 6.4 shows the *state table* for our sequential circuit. The table indicates all transitions from

each *present state* to the *next state* for different values of the input signal. Note that the output z is specified with respect to the present state, namely, the state that the circuit is in at present time. Note also that we did not include the *Reset* input; instead, we made an implicit assumption that the first state in the table is the starting state.

We now show the design steps that will produce the final circuit. To explain the basic design concepts, we first go through a traditional process of manually performing each design step. This is followed by a discussion of automated design techniques that use modern computer aided design (CAD) tools.

6.1.3 STATE ASSIGNMENT

The state table in Figure 6.4 defines the three states in terms of letters *A*, *B*, and *C*. When implemented in a logic circuit, each state is represented by a particular valuation (combination of values) of *state variables*. Each state variable may be implemented in the form of a flip-flop. Since three states have to be realized, it is sufficient to use two state variables. Let these variables be y_1 and y_2.

Now we can adapt the general block diagram in Figure 6.1 to our example as shown in Figure 6.5, to indicate the structure of the circuit that implements the required finite state machine. Two flip-flops represent the state variables. In the figure we have not specified the type of flip-flops to be used; this issue is addressed in the next subsection. From the specification in Figures 6.3 and 6.4, the output z is determined only by the present state of the circuit. Thus the block diagram in Figure 6.5 shows that z is a function of only y_1 and y_2; our design is of Moore type. We need to design a combinational circuit that uses y_1 and y_2 as input signals and generates a correct output signal z for all possible valuations of these inputs.

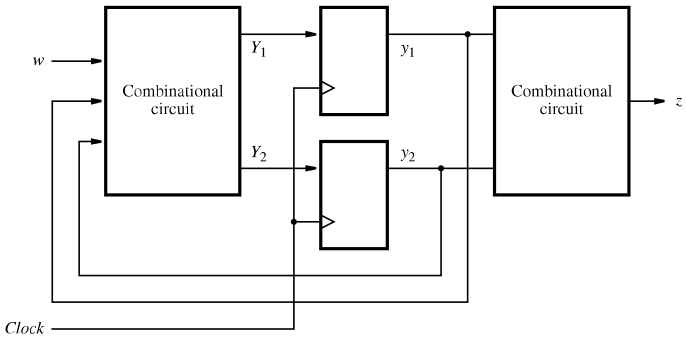


Figure 6.5 A general sequential circuit with input w, output z, and two state flip-flops.

	Present state	Next state $w = 0$	Next state $w = 1$	Output
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	dd	dd	d

Figure 6.6 State-assigned table corresponding to Figure 6.4.

The signals y_1 and y_2 are also fed back to the combinational circuit that determines the next state of the FSM. This circuit also uses the primary input signal w. Its outputs are two signals, Y_1 and Y_2, which are used to set the state of the flip-flops. Each active edge of the clock will cause the flip-flops to change their state according to the values of Y_1 and Y_2 at that time. Therefore, Y_1 and Y_2 are called the *next-state variables*, and y_1 and y_2 are called the *present-state variables*. We need to design a combinational circuit with inputs w, y_1, and y_2, such that for all valuations of these inputs the outputs Y_1 and Y_2 will cause the machine to move to the next state that satisfies our specification. The next step in the design process is to create a truth table that defines this circuit, as well as the circuit that generates z.

To produce the desired truth table, we assign a specific valuation of variables y_1 and y_2 to each state. One possible assignment is given in Figure 6.6, where the states A, B, and C are represented by $y_2y_1 = 00$, 01, and 10, respectively. The fourth valuation, $y_2y_1 = 11$, is not needed in this case.

The type of table given in Figure 6.6 is usually called a *state-assigned table*. This table can serve directly as a truth table for the output z with the inputs y_1 and y_2. Although for the next-state functions Y_1 and Y_2 the table does not have the appearance of a normal truth table, because there are two separate columns in the table for each value of w, it is obvious that the table includes all of the information that defines Y_1 and Y_2 in terms of valuations of inputs w, y_1, and y_2.

6.1.4 CHOICE OF FLIP-FLOPS AND DERIVATION OF NEXT-STATE AND OUTPUT EXPRESSIONS

From the state-assigned table in Figure 6.6, we can derive the logic expressions for the next-state and output functions. But first we have to decide on the type of flip-flops that will be used in the circuit. The most straightforward choice is to use D-type flip-flops, because in this case the values of Y_1 and Y_2 are simply clocked into the flip-flops to become the new values of y_1 and y_2. In other words, if the inputs to the flip-flops are called D_1 and D_2, then these signals are the same as Y_1 and Y_2. Note that the diagram in Figure 6.5

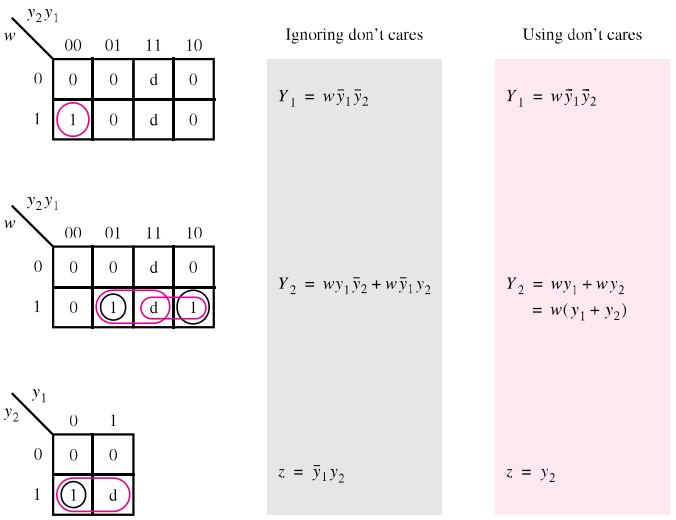


Figure 6.7 Derivation of logic expressions for the table in Figure 6.6.

corresponds exactly to this use of D-type flip-flops. For other types of flip-flops, such as JK type, the relationship between the next-state variable and inputs to a flip-flop is not as straightforward; we will consider this situation in Section 6.7.

The required logic expressions can be derived as shown in Figure 6.7. We use Karnaugh maps to make it easy for the reader to verify the validity of the expressions. Recall that in Figure 6.6 we needed only three of the four possible binary valuations to represent the states. The fourth valuation, $y_2y_1 = 11$, will not occur in the circuit because the circuit is constrained to move only within states A, B, and C; therefore, we may choose to treat this valuation as a don't-care condition. The resulting don't-care squares in the Karnaugh maps are denoted by d's. Using the don't cares to simplify the expressions, we obtain

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = w(y_1 + y_2)$$
$$z = y_2$$

If we do not use don't cares, then the resulting expressions are slightly more complex; they are shown in the gray-shaded area of Figure 6.7.

Since $D_1 = Y_1$ and $D_2 = Y_2$, the logic circuit that corresponds to the preceding expressions is implemented as shown in Figure 6.8. Observe that a clock signal is included, and

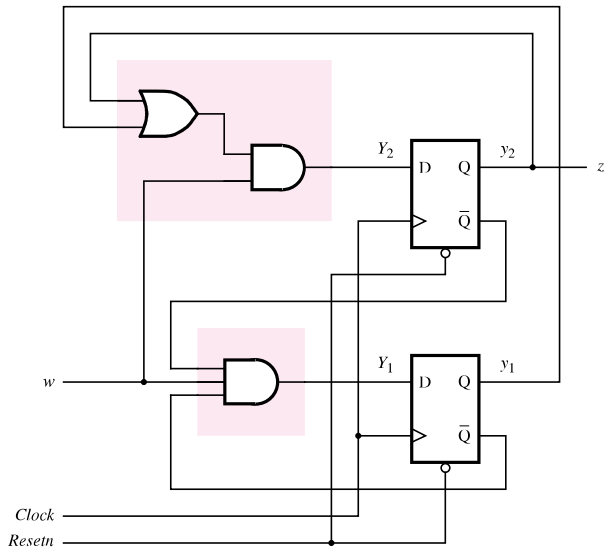


Figure 6.8 Final implementation of the sequential circuit.

the circuit is provided with an active-low reset capability. Connecting the clear input on the flip-flops to an external *Resetn* signal, as shown in the figure, provides a simple means for forcing the circuit into a known state. If we apply the signal $Resetn = 0$ to the circuit, then both flip-flops will be cleared to 0, placing the FSM into the state $y_2y_1 = 00$.

6.1.5 TIMING DIAGRAM

To understand fully the operation of the circuit in Figure 6.8, let us consider its timing diagram presented in Figure 6.9. The diagram depicts the signal waveforms that correspond to the sequences of values in Figure 6.2.

Because we are using positive-edge-triggered flip-flops, all changes in the signals occur shortly after the positive edge of the clock. The amount of delay from the clock edge depends on the propagation delays through the flip-flops. Note that the input signal w is also shown to change slightly after the active edge of the clock. This is a good assumption because in a typical digital system an input such as w would be just an output of another circuit that is synchronized by the same clock. We discuss the synchronization of input signals with the clock signal in Chapter 7.

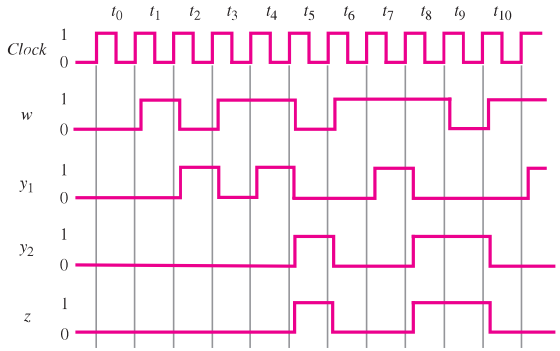


Figure 6.9 Timing diagram for the circuit in Figure 6.8.

A key point to observe is that even though w changes slightly after the active clock edge, and thus the value of w is equal to 1 (or 0) for almost the entire clock cycle, no change in the circuit will occur until the beginning of the next clock cycle when the positive edge causes the flip-flops to change their state. Thus the value of w must be equal to 1 for two clock cycles if the circuit is to reach state C and generate the output $z = 1$.

6.1.6 SUMMARY OF DESIGN STEPS

We can summarize the steps involved in designing a synchronous sequential circuit as follows:

1. Obtain the specification of the desired circuit.
2. Derive the states for the machine by first selecting a starting state. Then, given the specification of the circuit, consider all valuations of the inputs to the circuit and create new states as needed for the machine to respond to these inputs. To keep track of the states as they are visited, create a state diagram. When completed, the state diagram shows all states in the machine and gives the conditions under which the circuit moves from one state to another.
3. Create a state table from the state diagram. Alternatively, it may be convenient to directly create the state table in step 2, rather than first creating a state diagram.
4. In our sequential circuit example, there were only three states; hence it was a simple matter to create the state table that does not contain more states than necessary. However, in practice it is common to deal with circuits that have a large number of states. In such cases it is unlikely that the first attempt at deriving a state table will

produce optimal results, so that we may have more states than is really necessary. This can be corrected by a procedure that minimizes the number of states. We will discuss the process of state minimization in Section 6.6.

5. Decide on the number of state variables needed to represent all states and perform the state assignment. There are many different state assignments possible for a given sequential circuit. Some assignments may be better than others. In the preceding example we used what seemed to be a natural state assignment. We will return to this example in Section 6.2 and show that a different assignment may lead to a simpler circuit.
6. Choose the type of flip-flops to be used in the circuit. Derive the next-state logic expressions to control the inputs to all flip-flops and then derive logic expressions for the outputs of the circuit. So far we have used only D-type flip-flops. We will consider other types of flip-flops in Section 6.7.
7. Implement the circuit as indicated by the logic expressions.

We introduced the steps used in the design of sequential circuits by using an example of a simple controller for a vehicle. In this case the value of w reflected the speed of the vehicle. But, we could consider this example in a more general sense, in that the value of w in each clock cycle provides a *sequence* of symbols over time, where each symbol has the value 0 or 1. Our circuit can detect sequences of symbols that consist of consecutive 1s, and indicate this by generating $z = 1$ following each occurrence of two consecutive 1s. Beacause the circuit detects a specific sequence of symbols, we can say that it acts as a *sequence detector*. Similar circuits can be designed to detect a variety of different sequences.

Example 6.1

We have illustrated the design steps using a very simple sequential circuit. We will now consider a slightly more complex example that is closely tied to application in computer systems.

A computer system usually contains a number of registers that hold data during various operations. Sometimes it is necessary to swap the contents of two registers. Typically, this is done by using a temporary location, which is usually a third register. For example, suppose that we want to swap the contents of registers $R1$ and $R2$. We can achieve this by first transferring the contents of $R2$ into the third register, say $R3$. Next, we transfer the contents of $R1$ into $R2$. Finally, we transfer the contents of $R3$ into $R1$.

Registers in a computer system are connected via an interconnection network, as shown in Figure 6.10. In addition to the wires that connect to the network, each register has two control signals. The Rk_{out} signal causes the contents of register Rk to be placed into the interconnection network. The Rk_{in} signal causes the data from the network to be loaded into Rk. The Rk_{out} and Rk_{in} signals are generated by a control circuit, which is a finite state machine. For our example, we will design a control circuit that swaps the contents of $R1$ and $R2$, in response to an input $w = 1$. Therefore, the inputs to the control circuit will be w and *Clock*. The outputs will be $R1_{out}$, $R1_{in}$, $R2_{out}$, $R2_{in}$, $R3_{out}$, $R3_{in}$, and *Done* which indicates the completion of the required transfers.

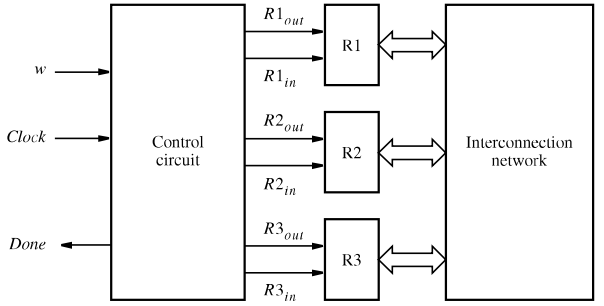


Figure 6.10 System for Example 6.1.

The desired swapping operation will be performed as follows. The contents of $R2$ are first loaded into $R3$, using the control signals $R2_{out} = 1$ and $R3_{in} = 1$. Then the contents of $R1$ are transferred into $R2$, using $R1_{out} = 1$ and $R2_{in} = 1$. Finally, the contents of $R3$ (which are the previous contents of $R2$) are transferred into $R1$, using $R3_{out} = 1$ and $R1_{in} = 1$. Since this step completes the required swap, we will set the signal $Done = 1$. Assume that the swapping is performed in response to a pulse on the input w, which has a duration of one clock cycle. Figure 6.11 gives a state diagram for a sequential circuit that generates the output control signals in the required sequence. Note that to keep the diagram simple, we have indicated the output signals only when they are equal to 1. In all other cases the output signals are equal to 0.

In the starting state, A, no transfer is indicated, and all output signals are 0. The circuit remains in this state until a request to swap arrives in the form of w changing to 1. In state B the signals required to transfer the contents of $R2$ into $R3$ are asserted. The next active clock edge places these contents into $R3$. This clock edge also causes the circuit to change to state C, regardless of whether w is equal to 0 or 1. In this state the signals for transferring $R1$ into $R2$ are asserted. The transfer takes place at the next active clock edge, and the circuit changes to state D regardless of the value of w. The final transfer, from $R3$ to $R1$, is performed on the clock edge that leaves state D, which also causes the circuit to return to state A.

Figure 6.12 presents the same information in a state table. Since there are four states, we can use two state variables, y_2 and y_1. A straightforward state assignment where the states A, B, C, and D are assigned the valuations $y_2y_1 = 00$, 01, 10, and 11, respectively, leads to the state-assigned table in Figure 6.13. Using this assignment and D-type flip-flops, the next-state expressions can be derived as shown in Figure 6.14. They are

$$Y_1 = w\bar{y}_1 + \bar{y}_1y_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2$$

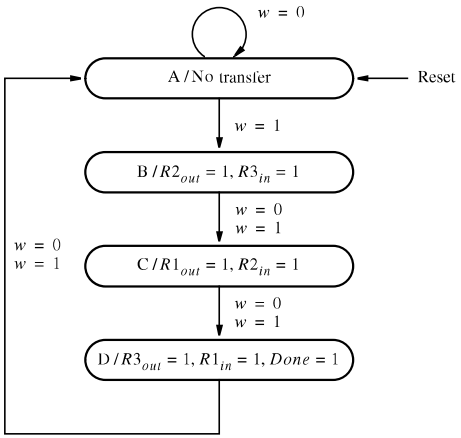


Figure 6.11 State diagram for Example 6.1.

Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

Figure 6.12 State table for Example 6.1.

The output control signals are derived as

$$R1_{out} = R2_{in} = \bar{y}_1 y_2$$
$$R1_{in} = R3_{out} = Done = y_1 y_2$$
$$R2_{out} = R3_{in} = y_1 \bar{y}_2$$

These expressions lead to the circuit in Figure 6.15.

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 0	0 0	0 1	0	0	0	0	0	0	0
B	0 1	1 0	1 0	0	0	1	0	0	1	0
C	1 0	1 1	1 1	1	0	0	1	0	0	0
D	1 1	0 0	0 0	0	1	0	0	1	0	1

Figure 6.13 State-assigned table corresponding to Figure 6.12.

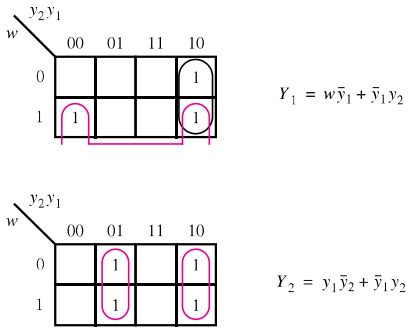


Figure 6.14 Derivation of next-state expressions for Figure 6.13.

6.2 STATE-ASSIGNMENT PROBLEM

Having introduced the basic concepts involved in the design of sequential circuits, we should revisit some details where alternative choices are possible. In Section 6.1.6 we suggested that some state assignments may be better than others. To illustrate this we can reconsider the example in Figure 6.4. We already know that the state assignment in Figure 6.6 leads to a simple-looking circuit in Figure 6.8. But can the FSM of Figure 6.4 be implemented with an even simpler circuit by using a different state assignment?

Figure 6.16 gives one possible alternative. In this case we represent the states A, B, and C with the valuations $y_2y_1 = 00$, 01, and 11, respectively. The remaining valuation, $y_2y_1 = 10$, is not needed, and we will treat it as a don't-care condition. The next-state and

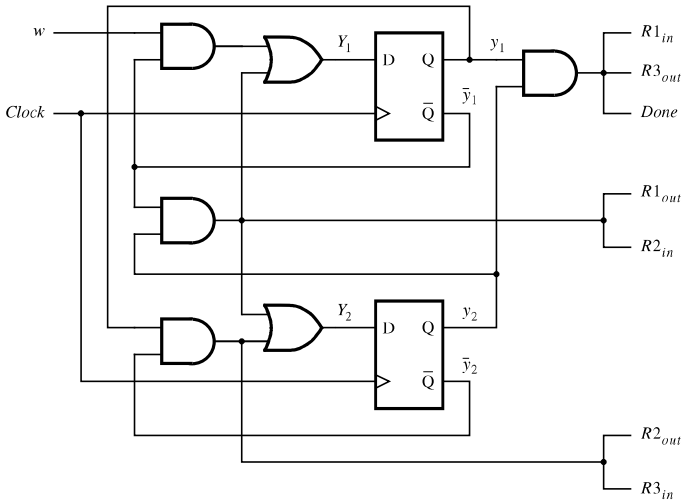


Figure 6.15 Final implementation of the sequential circuit for Example 6.1.

	Present state	Next state $w = 0$	Next state $w = 1$	Output
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	dd	dd	d

Figure 6.16 Improved state assignment for the state table in Figure 6.4.

output expressions derived from the figure will be

$$Y_1 = D_1 = w$$
$$Y_2 = D_2 = wy_1$$
$$z = y_2$$

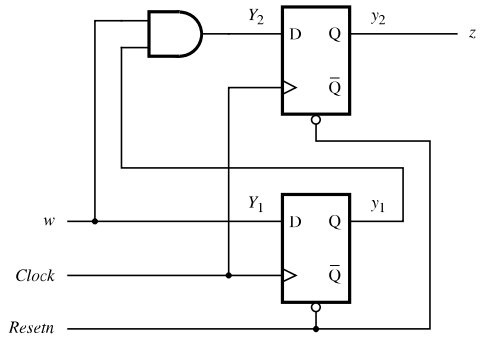


Figure 6.17 Final circuit for the improved state assignment in Figure 6.16.

These expressions define the circuit shown in Figure 6.17. Comparing this circuit with the one in Figure 6.8, we see that the cost of the new circuit is lower because it requires fewer gates.

In general, circuits are much larger than our example, and different state assignments can have a substantial effect on the cost of the final implementation. While highly desirable, it is often impossible to find the best state assignment for a large circuit. The exhaustive approach of trying all possible state assignments is not practical because the number of available state assignments is huge. CAD tools usually perform the state assignment using heuristic techniques. These techniques are usually proprietary, and their details are seldom published.

Example 6.2 In Figure 6.13 we used a straightforward state assignment for the sequential circuit in Figure 6.12. Consider now the effect of interchanging the valuations assigned to states C and D, as shown in Figure 6.18. Then the next-state expressions are

$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$
$$Y_2 = y_1$$

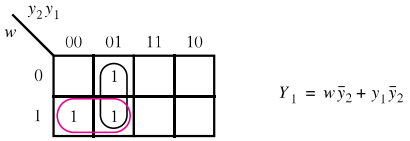
as derived in Figure 6.19. The output expressions are

$$R1_{out} = R2_{in} = y_1 y_2$$
$$R1_{in} = R3_{out} = Done = \bar{y}_1 y_2$$
$$R2_{out} = R3_{in} = y_1 \bar{y}_2$$

These expressions lead to a slightly simpler circuit than the one given in Figure 6.15.

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	01	0	0	0	0	0	0	0
B	01	11	11	0	0	1	0	0	1	0
C	11	10	10	1	0	0	1	0	0	0
D	10	00	00	0	1	0	0	1	0	1

Figure 6.18 Improved state assignment for the state table in Figure 6.12.



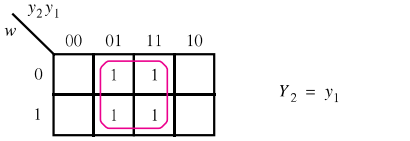


Figure 6.19 Derivation of next-state expressions for Figure 6.18.

6.2.1 ONE-HOT ENCODING

In previous examples we used the minimum number of flip-flops to represent the states of the FSM. Another interesting possibility is to use as many state variables as there are states. In this method, for each state all but one of the state variables are equal to 0. The variable whose value is 1 is deemed to be "hot." The approach is known as the *one-hot encoding* method.

Figure 6.20 shows how one-hot state assignment can be applied to the sequential circuit of Figure 6.4. Because there are three states, it is necessary to use three state variables. The chosen assignment is to represent the states A, B, and C using the valuations $y_3y_2y_1 = 001$, 010, and 100, respectively. The remaining five valuations of the state variables are not used. They can be treated as don't cares in the derivation of the next-state and output expressions.

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	z
A	001	001	010	0
B	010	001	100	0
C	100	001	100	1

Figure 6.20 One-hot state assignment for the state table in Figure 6.4.

Using this assignment, the resulting expressions are

$$Y_1 = \overline{w}$$
$$Y_2 = wy_1$$
$$Y_3 = w\overline{y}_1$$
$$z = y_3$$

Note that none of the next-state variables depends on the present-state variable y_2. This suggests that the second flip-flop and the expression $Y_2 = wy_1$ are not needed. (CAD tools detect and eliminate such redundancies!) But even then, the derived expressions are not simpler than those obtained using the state assignment in Figure 6.16. Although in this case the one-hot assignment is not advantageous, there are many cases where this approach is attractive.

Example 6.3 The one-hot state assignment can be applied to the sequential circuit of Figure 6.12 as indicated in Figure 6.21. Four state variables are needed, and the states A, B, C, and D are encoded as $y_4y_3y_2y_1 = 0001$, 0010, 0100, and 1000, respectively. Treating the remaining 12 valuations of the state variables as don't cares, the next-state expressions are

$$Y_1 = \overline{w}y_1 + y_4$$
$$Y_2 = wy_1$$
$$Y_3 = y_2$$
$$Y_4 = y_3$$

It is instructive to note that we can derive these expressions simply by inspecting the state diagram in Figure 6.11. There are two arcs leading to state A (not including the arc for the *Reset* signal). These arcs mean that flip-flop y_1 should be set to 1 if the FSM is already in state A and $w = 0$, or if the FSM is in state D; hence $Y_1 = \overline{w}y_1 + y_4$. Similarly, flip-flop y_2 should be set to 1 if the present state is A and $w = 1$; hence $Y_2 = wy_1$. Flip-flops y_3 and y_4 should be set to 1 if the FSM is presently in state B or C, respectively; hence $Y_3 = y_2$ and $Y_4 = y_3$.

	Present state	Next state $w = 0$	Next state $w = 1$	Outputs						
	$y_4y_3y_2y_1$	$Y_4Y_3Y_2Y_1$	$Y_4Y_3Y_2Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 0 0 1	0 0 0 1	0 0 1 0	0	0	0	0	0	0	0
B	0 0 1 0	0 1 0 0	0 1 0 0	0	0	1	0	0	1	0
C	0 1 0 0	1 0 0 0	1 0 0 0	1	0	0	1	0	0	0
D	1 0 0 0	0 0 0 1	0 0 0 1	0	1	0	0	1	0	1

Figure 6.21 One-hot state assignment for the state table in Figure 6.12.

The output expressions are just the outputs of the flip-flops, such that

$$R1_{out} = R2_{in} = y_3$$
$$R1_{in} = R3_{out} = Done = y_4$$
$$R2_{out} = R3_{in} = y_2$$

These expressions are simpler than those derived in Example 6.2, but four flip-flops are needed, rather than two.

An important feature of the one-hot state assignment is that it often leads to simpler output expressions than do assignments with the minimal number of state variables. Simpler output expressions may lead to a faster circuit. For instance, if the outputs of the sequential circuit are just the outputs of the flip-flops, as is the case in our example, then these output signals are valid as soon as the flip-flops change their states. If more complex output expressions are involved, then the propagation delay through the gates that implement these expressions must be taken into account. We will consider this issue in Section 6.8.2.

The examples considered to this point show that there are many ways to implement a given finite state machine as a sequential circuit. Each implementation is likely to have a different cost and different timing characteristics. In the next section we introduce another way of modeling FSMs that leads to even more possibilities.

6.3 MEALY STATE MODEL

Our introductory examples were sequential circuits in which each state had specific values of the output signals associated with it. As we explained at the beginning of the chapter, such finite state machines are said to be of Moore type. We will now explore the concept of Mealy-type machines in which the output values are generated based on both the state of the circuit and the present values of its inputs. This provides additional flexibility in the

design of sequential circuits. We will introduce the Mealy-type machines, using a slightly altered version of a previous example.

The essence of the first sequential circuit in Section 6.1 is to generate an output $z = 1$ whenever a second occurrence of the input $w = 1$ is detected in consecutive clock cycles. The specification requires that the output z be equal to 1 in the clock cycle that follows the detection of the second occurrence of $w = 1$. Suppose now that we eliminate this latter requirement and specify instead that the output z should be equal to 1 in the same clock cycle when the second occurrence of $w = 1$ is detected. Then a suitable input-output sequence may be as shown in Figure 6.22. To see how we can realize the behavior given in this table, we begin by selecting a starting state, A. As long as $w = 0$, the machine should remain in state A, producing an output $z = 0$. When $w = 1$, the machine has to move to a new state, B, to record the fact that an input of 1 has occurred. If w remains equal to 1 when the machine is in state B, which happens if $w = 1$ for at least two consecutive clock cycles, the machine should remain in state B and produce an output $z = 1$. As soon as w becomes 0, z should immediately become 0 and the machine should move back to state A at the next active edge of the clock. Thus the behavior specified in Figure 6.22 can be achieved with a two-state machine, which has a state diagram shown in Figure 6.23. Only two states are needed because we have allowed the output value to depend on the present value of the input as well as the present state of the machine. The diagram indicates that if the machine is in state A, it will remain in state A if $w = 0$ and the output will be 0. This is indicated by an arc with the label $w = 0/z = 0$. When w becomes 1, the output stays at 0 until the machine moves to state B at the next active clock edge. This is denoted by the arc from A to B with the label $w = 1/z = 0$. In state B the output will be 1 if $w = 1$, and the machine will remain in state B, as indicated by the label $w = 1/z = 1$ on the corresponding arc. However, if $w = 0$ in state B, then the output will be 0 and a transition to state A will take place at the

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

Figure 6.22 Sequences of input and output signals.

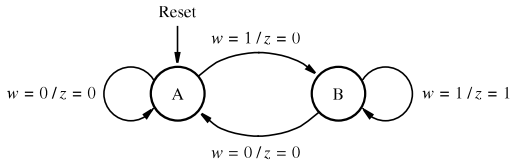


Figure 6.23 State diagram of an FSM that realizes the task in Figure 6.22.

next active clock edge. A key point to understand is that during the present clock cycle the output value corresponds to the label on the arc emanating from the present-state node.

We can implement the FSM in Figure 6.23, using the same design steps as in Section 6.1. The state table is shown in Figure 6.24. The table shows that the output z depends on the present value of input w and not just on the present state. Figure 6.25 gives the state-assigned table. Because there are only two states, it is sufficient to use a single state variable, y. Assuming that y is realized as a D-type flip-flop, the required next-state and output expressions are

$$Y = D = w$$
$$z = wy$$

The resulting circuit is presented in Figure 6.26 along with a timing diagram. The timing diagram corresponds to the input-output sequences in Figure 6.22.

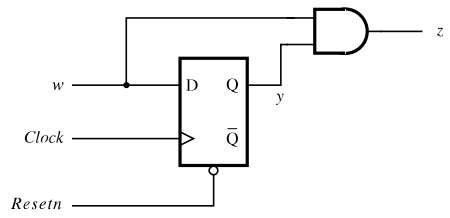
The greater flexibility of Mealy-type FSMs often leads to simpler circuit realizations. This certainly seems to be the case in our examples that produced the circuits in Figures 6.8, 6.17, and 6.26, assuming that the design requirement is only to detect two consecutive occurrences of input w being equal to 1. We should note, however, that the circuit in Figure 6.26 is not the same in terms of output behavior as the circuits in Figures 6.8 and 6.17. The difference is a shift of one clock cycle in the output signal in Figure 6.26*b*. If we wanted to produce exactly the same output behavior using the Mealy approach, we could modify the circuit in Figure 6.26*a* by adding another flip-flop as shown in Figure 6.27. This flip-flop merely delays the output signal, Z, by one clock cycle with respect to z, as indicated

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

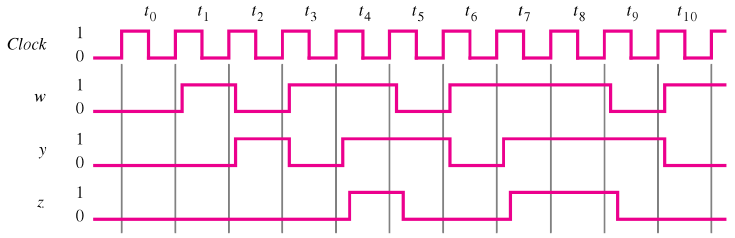
Figure 6.24 State table for the FSM in Figure 6.23.

	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

Figure 6.25 State-assigned table for the FSM in Figure 6.24.



(a) Circuit



(b) Timing diagram

Figure 6.26 Implementation of the FSM in Figure 6.25.

in the timing diagram. By making this change, we effectively turn the Mealy-type circuit into a Moore-type circuit with output Z. Note that the circuit in Figure 6.27 is essentially the same as the circuit in Figure 6.17.

Example 6.4 In Example 6.1 we considered the control circuit needed to swap the contents of two registers, implemented as a Moore-type finite state machine. The same task can be achieved using a Mealy-type FSM, as indicated in Figure 6.28. State A still serves as the reset state. But as soon as w changes from 0 to 1, the output control signals $R2_{out}$ and $R3_{in}$ are asserted. They remain asserted until the beginning of the next clock cycle, when the circuit will leave state A and change to B. In state B the outputs $R1_{out}$ and $R2_{in}$ are asserted for both $w = 0$ and $w = 1$. Finally, in state C the swap is completed by asserting $R3_{out}$ and $R1_{in}$.

The Mealy-type realization of the control circuit requires three states. This does not necessarily imply a simpler circuit because two flip-flops are still needed to implement the state variables. The most important difference in comparison with the Moore-type realization is the timing of output signals. A circuit that implements the FSM in Figure 6.28

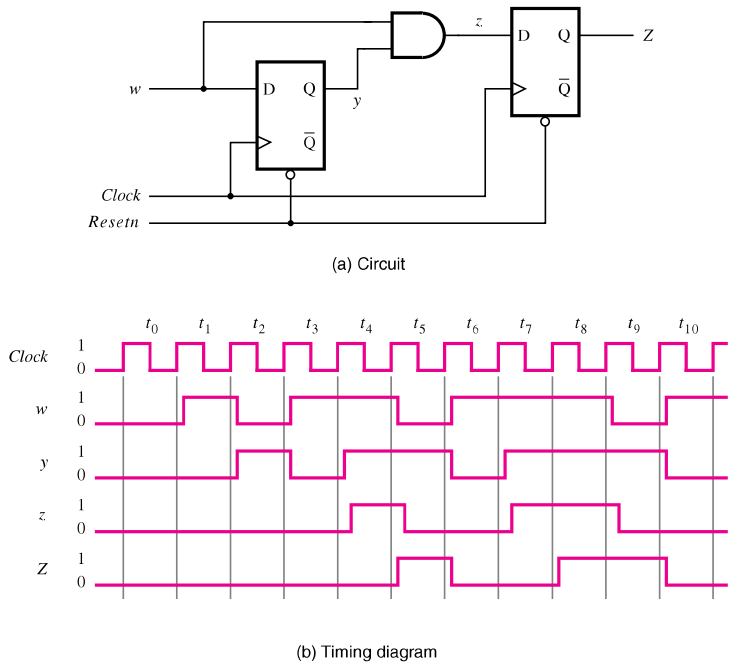


Figure 6.27 Circuit that implements the specification in Figure 6.2.

generates the output control signals one clock cycle sooner than the circuits derived in Examples 6.1 and 6.2.

Note also that using the FSM in Figure 6.28, the entire process of swapping the contents of $R1$ and $R2$ takes three clock cycles, starting and finishing in state A. Using the Moore-type FSM in Example 6.1, the swapping process involves four clock cycles before the circuit returns to state A.

Suppose that we wish to implement this FSM using one-hot encoding. Then three flip-flops are needed, and the states A, B, and C may be assigned the valuations $y_3y_2y_1 =$ 001, 010, and 100, respectively. Examining the state diagram in Figure 6.28, we can derive the next-state equations by inspection. The input to flip-flop y_1 should have the value 1 if the FSM is in state A and $w = 0$ or if the FSM is in state C; hence $Y_1 = \overline{w}y_1 + y_3$.

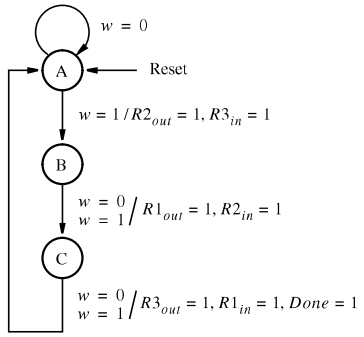


Figure 6.28 State diagram for Example 6.4.

Flip-flop y_2 should be set to 1 if the FSM is in state A and $w = 1$; hence $Y_2 = wy_1$. Flip-flop y_3 should be set to 1 if the present state is B; hence $Y_3 = y_2$. The derivation of the output expressions, which we leave as an exercise for the reader, can also be done by inspection.

The preceding discussion deals with the basic principles involved in the design of sequential circuits. Although it is essential to understand these principles, the manual approach used in the examples is difficult and tedious when large circuits are involved. We will now show how CAD tools are used to greatly simplify the design task.

6.4 DESIGN OF FINITE STATE MACHINES USING CAD TOOLS

Sophisticated CAD tools are available for finite state machine design, and we introduce them in this section. A rudimentary way of using CAD tools for FSM design could be as follows: The designer employs the manual techniques described previously to derive a circuit that contains flip-flops and logic gates from a state diagram. This circuit is entered into the CAD system by drawing a schematic diagram or by writing structural hardware description language (HDL) code. The designer then uses the CAD system to simulate the behavior of the circuit and uses the CAD tools to automatically implement the circuit in a chip, such as a PLD.

It is tedious to manually synthesize a circuit from a state diagram. Since CAD tools are meant to obviate this type of task, more attractive ways of utilizing CAD tools for FSM design have been developed. A better approach is to directly enter the state diagram into the

CAD system and perform the entire synthesis process automatically. CAD tools support this approach in two main ways. One method is to allow the designer to draw the state diagram using a graphical tool similar to the schematic capture tool. The designer draws circles to represent states and arcs to represent state transitions and indicates the outputs that the machine should generate. Another and more popular approach is to write HDL code that represents the state diagram, as described below.

Many HDLs provide constructs that allow the designer to represent a state diagram. To show how this is done, we will provide Verilog code that represents the simple machine designed manually as the first example in Section 6.1. Then we will use the CAD tools to synthesize a circuit that implements the machine in a chip.

6.4.1 VERILOG CODE FOR MOORE-TYPE FSMS

Verilog does not define a standard way of describing a finite state machine. Hence while adhering to the required Verilog syntax, there is more than one way to describe a given FSM. An example of Verilog code for the FSM of Figure 6.3 is given in Figure 6.29. The code reflects directly the FSM structure in Figure 6.5. The module *simple* has inputs *Clock*, *Resetn*, and w, and output z. Two-bit vectors y and Y represent the present and the next state of the machine, respectively. The **parameter** statement associates the present-state valuations used in Figure 6.6 with the symbolic names A, B, and C.

The state transitions are specified by two separate **always** blocks. The first block describes the required combinational circuit. It uses a **case** statement to give the value of the next state Y for each value of the present state y. Each case alternative corresponds to a present state of the machine, and the associated **if-else** statement specifies the next state to be reached according to the value of w. This portion of the code corresponds to the combinational circuit on the left side of Figure 6.5.

Since there are only three states in our FSM, the **case** statement in Figure 6.29 includes a **default** clause that covers the unused valuation $y = 11$. This clause specifies Y = 2'bxx, which indicates to the Verilog compiler that the value of Y can be treated as a don't-care when $y = 11$. It is important to consider the effect on the compiler if we did not include the **default** clause. In that case, the value of Y would not be specified at all for $y = 11$. As explained in Chapter 5, the Verilog compiler assumes that a signal's value must be kept unchanged whenever the value is not specified by an assignment, and a memory element should be synthesized. For the code in Figure 6.29 this means that a latch will be generated in the combinational circuit for Y if the **default** clause is not included. In general, it is always necessary to include a **default** clause when using a **case** statement that does not cover all of its alternatives.

The second **always** block introduces flip-flops into the circuit. Its sensitivity list comprises the reset and clock signals. Asynchronous reset is performed when the *Resetn* input goes to 0, placing the FSM into state A. The **else** clause stipulates that after each positive clock edge the y signal should take the value of the Y signal, thus implementing the state changes.


```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A: if (w)    Y = B;
            else      Y = A;
         B: if (w)    Y = C;
            else      Y = A;
         C: if (w)    Y = C;
            else      Y = A;
         default:     Y = 2'bxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define output
   assign  z = (y == C);

endmodule
```

Figure 6.29 Verilog code for the FSM in Figure 6.3.

This is a Moore-type FSM in which the output z is equal to 1 only in state C. The output is conveniently defined in a conditional assignment statement that sets $z = 1$ if $y = C$. This realizes the combinational circuit on the right side of Figure 6.5.

6.4.2 SYNTHESIS OF VERILOG CODE

To give an example of the circuit produced by a synthesis tool, we synthesised the code in Figure 6.29 for implementation in a CPLD, such as those described in Appendix B. The synthesis resulted in two flip-flops, with inputs Y_1 and Y_2, and outputs y_1 and y_2. The next-state expressions generated by the synthesis tool are

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = wy_1 + wy_2$$

The output expression is

$$z = y_2$$

These expressions correspond to the case in Figure 6.7 when the unused state pattern $y_2y_1 = 11$ is treated as don't-cares in the Karnaugh maps for Y_1, Y_2, and z.

Figure 6.30 depicts a part of the FSM circuit implemented in a CPLD. To keep the figure simple, only the logic resources used for the two macrocells that implement y_1, y_2, and z are shown. The parts of the macrocells used for the circuit are highlighted in blue.

The w input to the circuit is shown connected to one of the interconnection wires in the CPLD. The source node in the chip that generates w is not shown. It could be either an input pin, or else w might be the output of another macrocell, assuming that the CPLD may

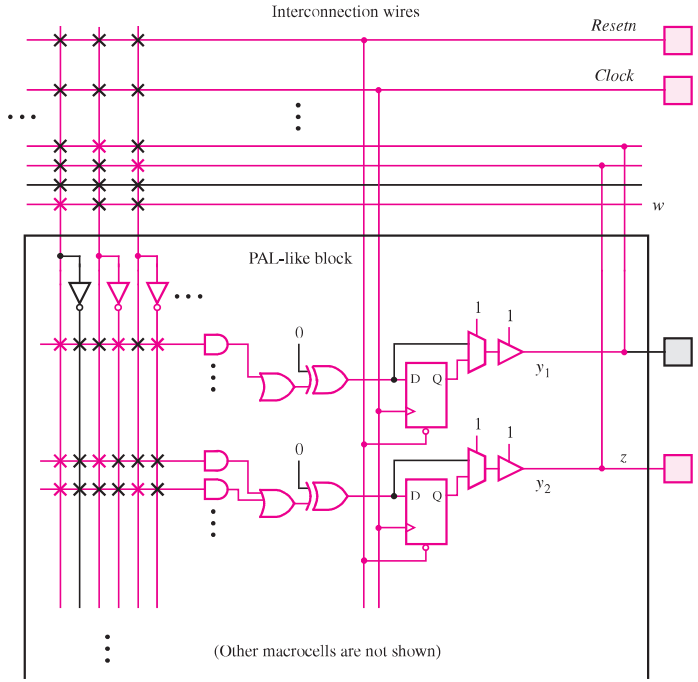


Figure 6.30 Implementation of the FSM of Figure 6.3 in a CPLD.

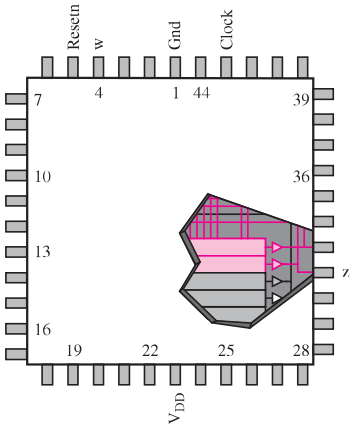


Figure 6.31 The circuit from Figure 6.30 in a small CPLD.

contain other circuitry that is connected to our FSM. The *Clock* signal is assigned to a pin on the chip that is dedicated for use by clock signals. From this dedicated pin a *global wire* distributes the clock signal to all of the flip-flops in the chip. The global wire distributes the clock signal to the flip-flops such that the difference in the arrival time, or *clock skew*, of the clock signal at each flip-flop is minimized. The concept of clock skew is discussed in Chapter 5. A global wire is also used for the reset signal.

Figure 6.31 illustrates how the circuit might be assigned to the pins on a small CPLD. The figure is drawn with a part of the top of the chip package cut away, revealing a conceptual view of the two macrocells from Figure 6.30, which are indicated in blue. Our simple circuit uses only a small portion of the device.

6.4.3 SIMULATING AND TESTING THE CIRCUIT

The behavior of the circuit implemented in the CPLD chip can be tested using timing simulation, as depicted in Figure 6.32. The figure gives the waveforms that correspond to the timing diagram in Figure 6.9, assuming that a 100 ns clock period is used. The *Resetn* signal is set to 0 at the beginning of the simulation and then set to 1. The circuit produces the output $z = 1$ for one clock cycle after w has been equal to 1 for two successive clock cycles. When w is 1 for three clock cycles, z becomes 1 for two clock cycles, as it should be. We show the changes in state by using the letters A, B, and C for readability purposes.

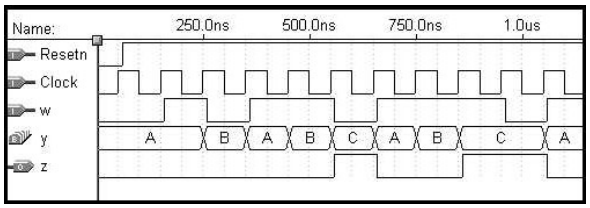


Figure 6.32 Simulation results for the circuit in Figure 6.30.

Having examined the simulation output, we should consider the question of whether we can conclude that the circuit functions correctly and satisfies all requirements. For our simple example it is not difficult to answer this question because the circuit has only one input and its behavior is straightforward. It is easy to see that the circuit works properly. However, in general it is difficult to ascertain with a high degree of confidence whether a sequential circuit will work properly for all possible input sequences, because a very large number of input patterns may be possible. For large finite state machines, the designer must think carefully about patterns of inputs that may be used in simulation for testing purposes.

6.4.4 ALTERNATIVE STYLES OF VERILOG CODE

We mentioned earlier in this section that Verilog does not specify a standard way for writing code that represents a finite state machine. The code given in Figure 6.29 is only one possibility. A slightly different version of code for our simple machine is given in Figure 6.33. In this case, we specified the output z inside the **always** block that defines the required combinational circuit. The effect is the same as in Figure 6.29.

A different approach is taken in Figure 6.34. A single **always** block is used. The states are represented by a two-bit vector y. The required state transitions are given in the **always** block with the sensitivity list that comprises the reset and clock signals. Asynchronous reset is specified when *Resetn* goes to 0. Other transitions are defined in the **case** statement to correspond directly to those in Figure 6.3. The **default** clause indicates that the valuation $y = y_2y_1 = 11$ can be treated as a don't-care condition.

The assignment statement that defines z is placed outside the **always** block. This assignment cannot be made inside the **always** block as done in Figure 6.33. Doing so would infer a separate flip-flop for z, which would delay the changes in z with respect to y_2 by one clock cycle.

We have shown three styles of Verilog code for our FSM example. The circuit produced by the Verilog compiler for each version of the code may be somewhat different because, as the reader is well aware by this point, there are many ways to implement a given logic function. However, the circuits produced from the three versions of the code provide identical functionality.


```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output reg z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the next state and output combinational circuits
   always @(w, y)
   begin
      case (y)
         A: if (w)    Y = B;
            else      Y = A;
         B: if (w)    Y = C;
            else      Y = A;
         C: if (w)    Y = C;
            else      Y = A;
         default:     Y = 2'bxx;
      endcase
      z = (y == C);   // Define output
   end

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)   y <= A;
      else   y <= Y;

endmodule
```

Figure 6.33 Second version of code for the FSM in Figure 6.3.

Example 6.5 Figure 6.35 shows how the FSM in Figure 6.11 can be specified in Verilog using the style illustrated in Figure 6.29. This FSM has four states, which are encoded using all four possible valuations of the state variables, hence there is no need for a **default** clause in the **case** statement.

6.4.5 SUMMARY OF DESIGN STEPS WHEN USING CAD TOOLS

In Section 6.1.6 we summarized the design steps needed to derive sequential circuits manually. We have now seen that CAD tools can automatically perform much of the work. However, it is important to realize that the CAD tools have not replaced *all* manual steps.

```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [2:1] y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else
         case (y)
            A:  if (w)    y <= B;
                else      y <= A;
            B:  if (w)    y <= C;
                else      y <= A;
            C:  if (w)    y <= C;
                else      y <= A;
            default:      y <= 2'bxx;
         endcase

   // Define output
   assign z = (y == C);

endmodule
```

Figure 6.34 Third version of code for the FSM in Figure 6.3.

With reference to the list given in Section 6.1.6, the first two steps, in which the machine specification is obtained and a state diagram is derived, still have to be done manually. Given the state diagram information as input, the CAD tools then automatically perform the tasks needed to generate a circuit with logic gates and flip-flops. In addition to the design steps given in Section 6.1.6, we should add the testing and simulation stage. We will defer detailed discussion of this issue until Chapter 11.

6.4.6 SPECIFYING THE STATE ASSIGNMENT IN VERILOG CODE

In Section 6.2 we saw that the state assignment may have an impact on the complexity of the designed circuit. An obvious objective of the state-assignment process is to minimize the cost of implementation. The cost function that should be optimized may be simply the number of gates and flip-flops. But it could also be based on other considerations that may be representative of the type of chip used to implement the design.


```
module control (Clock, Resetn, w, R1in, R1out, R2in, R2out, R3in, R3out, Done);
   input Clock, Resetn, w;
   output R1in, R1out, R2in, R2out, R3in, R3out, Done;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A:  if (w)    Y = B;
             else      Y = A;
         B:            Y = C;
         C:            Y = D;
         D:            Y = A;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define outputs
   assign R2out = (y == B);
   assign R3in = (y == B);
   assign R1out = (y == C);
   assign R2in = (y == C);
   assign R3out = (y == D);
   assign R1in = (y == D);
   assign Done = (y == D);

endmodule
```

Figure 6.35 Verilog code for the FSM in Figure 6.11.

A particular state assignment can be specified in Verilog code by means of a **parameter** statement as done in Figures 6.29 through 6.35. However, Verilog compilers usually have a capability to search for different assignments that may give better results. Most compilers are able to recognize a finite state machine when they encounter its specification in the code that conforms to typical styles, such as those used in this chapter. When the compiler detects an FSM, it can try to optimize its implementation by applying certain strategies such as looking for a better state assignment, attempting to use the one-hot encoding, and exploiting specific features of the target device. The user can either allow the compiler to use its FSM-handling capability, or surpress it in which case the compiler simply deals with the Verilog statements in the usual way.

6.4.7 SPECIFICATION OF MEALY FSMS USING VERILOG

A Mealy-type FSM can be specified in a similar manner as a Moore-type FSM. Figure 6.36 gives Verilog code for the FSM in Figure 6.23. The state transitions are described in the same way as in our first Verilog example in Figure 6.29. The variables y and Y represent the present and next states, which can have values A and B. Compared to the code in Figure 6.29, the main difference is the way in which the code for the output is written. In Figure 6.36 the output z is defined within the **case** statement that also defines the state transitions. When the FSM is in state A, z should be 0, but when in state B, z should take the value of w. Since the sensitivity list for the **always** block includes w, a change in w will immediately reflect itself in the value of z if the machine is in state B, which meets the requirements of the Mealy-type FSM.

Implementing the FSM specified in Figure 6.36 in a CPLD chip yields the same equations as we derived manually in Section 6.3. Simulation results for the synthesized circuit appear in Figure 6.37. The input waveform for w is the same as the one we used for the Moore-type machine in Figure 6.32. Our Mealy-type machine behaves correctly, with z becoming 1 just after the start of the second consecutive clock cycle in which w is 1.

In the simulation results we have given in this section, all changes in the input w occur immediately following a positive clock edge. This is based on the assumption, stated in Section 6.1.5, that in a real circuit w would be synchronized with respect to the clock that controls the FSM. In Figure 6.38 we illustrate a problem that may arise if w does not meet this specification. In this case we have assumed that the changes in w take place at the negative edge of the clock, rather than at the positive edge when the FSM changes its state. The first pulse on the w input is 100 ns long. This should not cause the output z to become equal to 1. But the circuit does not behave in this manner. After the signal w becomes equal to 1, the first positive edge of the clock causes the FSM to change from state A to state B. As soon as the circuit reaches the state B, the w input is still equal to 1 for another 50 ns, which causes z to go to 1. When w returns to 0, the z signal does likewise. Thus an erroneous 50-ns pulse is generated on the output z.

We should pursue the consequences of this problem a little further. If z is used to drive another circuit that is not controlled by the same clock, then the extraneous pulse is likely to cause big problems. But if z is used as an input to a circuit (perhaps another FSM) that is controlled by the same clock, then the 50-ns pulse will be ignored by this circuit if $z = 0$ before the next positive edge of the clock (accounting for the setup time).

6.5 SERIAL ADDER EXAMPLE

We will now present another simple example that illustrates the complete design process. In Chapter 3 we discussed the addition of binary numbers in detail. We explained several schemes that can be used to add two n-bit numbers in parallel, ranging from carry-ripple to carry-lookahead adders. In these schemes the speed of the adder unit is an important design parameter. Fast adders are more complex and thus more expensive. If speed is not of great importance, then a cost-effective option is to use a *serial adder*, in which bits are added a pair at a time.


```
module mealy (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output reg z;
   reg y, Y;
   parameter A = 1'b0, B = 1'b1;

   // Define the next state and output combinational circuits
   always @(w, y)
      case (y)
         A: if (w)
               begin
                  z = 0;
                  Y = B;
               end
               else
               begin
                  z = 0;
                  Y = A;
               end
         B: if (w)
               begin
                  z = 1;
                  Y = B;
               end
               else
               begin
                  z = 0;
                  Y = A;
               end
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

endmodule
```

Figure 6.36 Verilog code for the Mealy machine of Figure 6.23.

6.5.1 MEALY-TYPE FSM FOR SERIAL ADDER

Let $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$ be two unsigned numbers that have to be added to produce $Sum = s_{n-1}s_{n-2} \cdots s_0$. Our task is to design a circuit that will perform serial addition, dealing with a pair of bits in one clock cycle. The process starts by adding bits a_0 and b_0. In the next clock cycle, bits a_1 and b_1 are added, including a possible

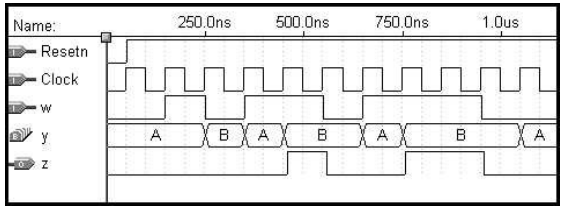


Figure 6.37 Simulation results for the Mealy machine.

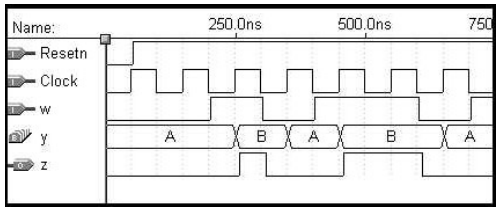


Figure 6.38 Potential problem with asynchronous inputs to a Mealy FSM.

carry from the bit-position 0, and so on. Figure 6.39 shows a block diagram of a possible implementation. It includes three shift registers that are used to hold A, B, and *Sum* as the computation proceeds. Assuming that the input shift registers have parallel-load capability, as depicted in Figure 5.18, the addition task begins by loading the values of A and B into these registers. Then in each clock cycle, a pair of bits is added by the adder FSM, and at the end of the cycle the resulting sum bit is shifted into the *Sum* register. We will use positive-edge-triggered flip-flops in which case all changes take place soon after the positive edge of the clock, depending on the propagation delays within the various flip-flops. At this time the contents of all three shift registers are shifted to the right; this shifts the existing sum bit into *Sum*, and it presents the next pair of input bits a_i and b_i to the adder FSM.

Now we are ready to design the required FSM. This cannot be a combinational circuit because different actions will have to be taken, depending on the value of the carry from the previous bit position. Hence two states are needed: let G and H denote the states where the carry-in values are 0 and 1, respectively. Figure 6.40 gives a suitable state diagram, defined as a Mealy model. The output value, s, depends on both the state and the present value of the inputs a and b. Each transition is labeled using the notation ab/s, which indicates the value of s for a given valuation ab. In state G the input valuation 00 will produce $s = 0$, and the FSM will remain in the same state. For input valuations 01 and 10, the output will be $s = 1$, and the FSM will remain in G. But for 11, $s = 0$ is generated, and the machine

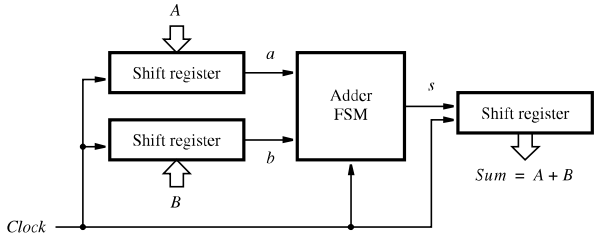


Figure 6.39 Block diagram for the serial adder.

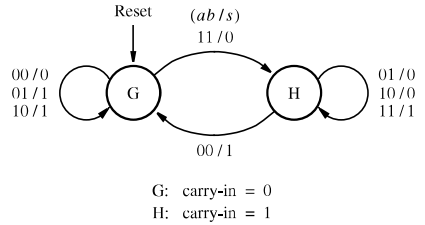


Figure 6.40 State diagram for the serial adder FSM.

moves to state H. In state H valuations 01 and 10 cause $s = 0$, while 11 causes $s = 1$. In all three of these cases, the machine remains in H. However, when the valuation 00 occurs, the output of 1 is produced and a change into state G takes place.

The corresponding state table is presented in Figure 6.41. A single flip-flop is needed to represent the two states. The state assignment can be done as indicated in Figure 6.42. This assignment leads to the following next-state and output equations

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

Comparing these expressions with those for the full-adder in Section 3.2, it is obvious that y is the carry-in, Y is the carry-out, and s is the sum of the full-adder. Therefore, the adder FSM box in Figure 6.39 consists of the circuit shown in Figure 6.43. The flip-flop can be cleared by the *Reset* signal at the start of the addition operation.

The serial adder is a simple circuit that can be used to add numbers of any length. The structure in Figure 6.39 is limited in length only by the size of the shift registers.

Present state	Next state				Output s			
	$ab = 00$	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Figure 6.41 State table for the serial adder FSM.

Present state	Next state				Output			
	$ab = 00$	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Figure 6.42 State-assigned table for Figure 6.41.

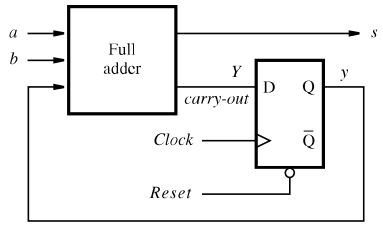


Figure 6.43 Circuit for the adder FSM in Figure 6.39.

6.5.2 MOORE-TYPE FSM FOR SERIAL ADDER

In the preceding example we saw that a Mealy-type FSM nicely meets the requirement for implementing the serial adder. Now we will try to achieve the same objective using a Moore-type FSM. A good starting point is the state diagram in Figure 6.40. In a Moore-type FSM, the output must depend only on the state of the machine. Since in both states, G and H, it is possible to produce two different outputs depending on the valuations of the inputs

a and b, a Moore-type FSM will need more than two states. We can derive a suitable state diagram by splitting both G and H into two states. Instead of G, we will use G_0 and G_1 to denote the fact that the carry is 0 and that the sum is either 0 or 1, respectively. Similarly, instead of H, we will use H_0 and H_1. Then the information in Figure 6.40 can be mapped into the Moore-type state diagram in Figure 6.44 in a straightforward manner.

The corresponding state table is given in Figure 6.45 and the state-assigned table in Figure 6.46. The next-state and output expressions are

$$Y_1 = a \oplus b \oplus y_2$$
$$Y_2 = ab + ay_2 + by_2$$
$$s = y_1$$

The expressions for Y_1 and Y_2 correspond to the sum and carry-out expressions in the full-adder circuit. The FSM is implemented as shown in Figure 6.47. It is interesting to observe that this circuit is very similar to the circuit in Figure 6.43. The only difference is that in the Moore-type circuit, the output signal, s, is passed through an extra flip-flop and thus delayed by one clock cycle with respect to the Mealy-type sequential circuit. Recall that we observed the same difference in our previous example, as depicted in Figures 6.26 and 6.27.

A key difference between the Mealy and Moore types of FSMs is that in the former a change in inputs reflects itself immediately in the outputs, while in the latter the outputs do not change until the change in inputs forces the machine into a new state, which takes place one clock cycle later. We encourage the reader to draw the timing diagrams for the circuits in Figures 6.43 and 6.47, which will exemplify further this key difference between the two types of FSMs.

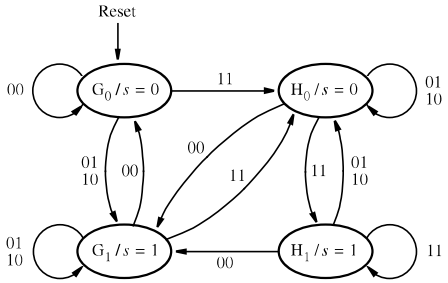


Figure 6.44 State diagram for the Moore-type serial adder FSM.

Present state	Next state				Output
	$ab = 00$	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Figure 6.45 State table for the Moore-type serial adder FSM.

Present state y_2y_1	Next state				Output
	$ab = 00$	01	10	11	s
	Y_2Y_1				
0 0	0 0	0 1	0 1	1 0	0
0 1	0 0	0 1	0 1	1 0	1
1 0	0 1	1 0	1 0	1 1	0
1 1	0 1	1 0	1 0	1 1	1

Figure 6.46 State-assigned table for Figure 6.45.

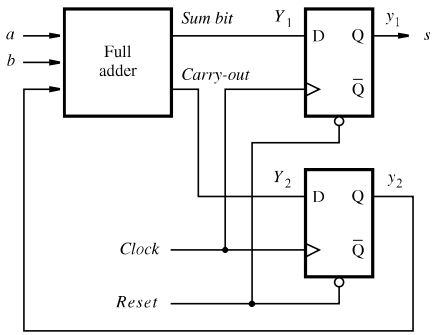


Figure 6.47 Circuit for the Moore-type serial adder FSM.

6.5.3 VERILOG CODE FOR THE SERIAL ADDER

The serial adder can be described in Verilog by writing code for the shift registers and the adder FSM. We will first design the shift register and then use it as a subcircuit in the serial adder.

Shift Register Subcircuit

In the serial adder it is beneficial to have the ability to prevent the shift register contents from changing when an active clock edge occurs. Figure 6.48 gives the code for a shift register named *shiftrne*, which has an enable input, E. When $E = 1$, the contents of the register are shifted left-to-right on the positive edge of the clock. Setting $E = 0$ prevents the contents of the shift register from changing.

Complete Code

The code for the serial adder is shown in Figure 6.49. It instantiates three shift registers for the inputs A and B and the output *Sum*. The shift registers are loaded with parallel data when the circuit is reset. The state diagram for the adder FSM is described by two **always** blocks, using the style of code in Figure 6.36. In addition to the components of the serial adder shown in Figure 6.39, the Verilog code includes a down-counter to determine when the adder should be halted because all n bits of the required sum are present in the output shift register. When the circuit is reset, the counter is loaded with the number of bits in the serial adder, n. The counter counts down to 0, and then stops and disables further changes in the output shift register.

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

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

endmodule
```

Figure 6.48 Code for a left-to-right shift register with an enable input.

```
module serial_adder(A, B, Reset, Clock, Sum);
   input [7:0] A, B;
   input Reset, Clock;
   output wire [7:0] Sum;
   reg [3:0] Count;
   reg s, y, Y;
   wire [7:0] QA, QB;
   wire Run;
   parameter G = 1'b0, H = 1'b1;

   shiftrne shift_A (A, Reset, 1'b1, 1'b0, Clock, QA);
   shiftrne shift_B (B, Reset, 1'b1, 1'b0, Clock, QB);
   shiftrne shift_Sum (8'b0, Reset, Run, s, Clock, Sum);

   // Adder FSM
   // Output and next state combinational circuit
   always @(QA, QB, y)
      case (y)
         G: begin
               s = QA[0] ^ QB[0];
               if (QA[0] & QB[0])  Y = H;
               else  Y = G;
            end
         H: begin
               s = QA[0] ~^ QB[0];
               if (~QA[0] & ~QB[0])  Y = G;
               else  Y = H;
            end
         default:  Y = G;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Reset)  y <= G;
      else  y <= Y;

   // Control the shifting process
   always @(posedge Clock)
      if (Reset)  Count = 8;
      else if (Run)  Count = Count - 1;
   assign Run = |Count;

endmodule
```

Figure 6.49 Verilog code for the serial adder.

The code in Figure 6.49 implements a serial adder for eight-bit numbers. The wires QA and QB correspond to the parallel outputs of the shift registers with inputs A and B in Figure 6.39. The variable s represents the output of the adder FSM.

In Figure 6.39 the shift registers for inputs A and B do not use a serial input or an enable input. However, the *shiftrne* component, which is used for all three shift registers, includes these ports and so signals must be connected to them. The enable input for the two shift registers can be connected to logic value 1. The value shifted into the serial input does not matter, so it can be connected to either 1 or 0; we have chosen to connect it to 0. The shift registers are loaded in parallel by the *Reset* signal. We have chosen to use an active-high reset signal for the circuit. The output shift register does not need a parallel data input, so all 0s are connected to this input.

The first **always** block describes the state transitions and the output of the adder FSM in Figure 6.40. The output definition follows from observing in Figure 6.40 that when the FSM is in state G, the sum is $s = a \oplus b$, and when in state H, the sum is $s = \overline{a \oplus b}$. The second **always** block implements the flip-flop y and provides synchronous reset when *Reset* = 1.

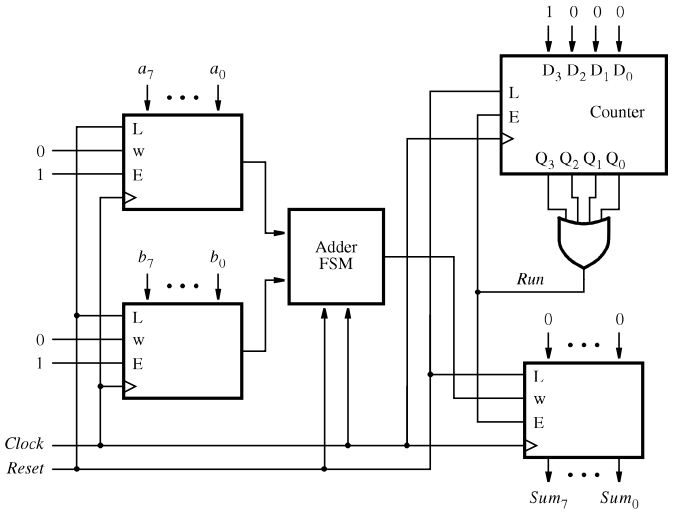
The enable input for the output shift register is named *Run*. It is derived from the outputs of the down-counter specified in the third **always** block. When *Reset* = 1, *Count* is initialized to the value 8. Then as long as *Run* = 1, *Count* is decremented in each clock cycle. *Run* is set to 0 when *Count* is equal to 0, which is detected by using the reduction OR operator.

Synthesis and Simulation of the Verilog Code

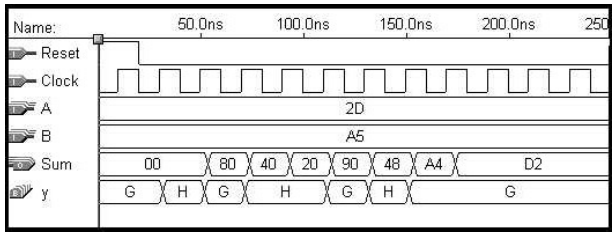
The results of synthesizing a circuit from the code in Figure 6.49 are illustrated in Figure 6.50*a*. The outputs of the counter are ORed to provide the *Run* signal, which enables clocking of both the output shift register and the counter. A sample of a timing simulation for the circuit is shown in Figure 6.50*b*. The circuit is first reset, resulting in the values of A and B being loaded into the input shift registers, and the value 8 loaded into the down-counter. After each clock cycle one pair of bits of the input numbers is added by the adder FSM, and the sum bit is shifted into the output shift register. After eight clock cycles the output shift register contains the correct sum, and shifting is halted by the *Run* signal becoming equal to 0.

6.6 STATE MINIMIZATION

Our introductory examples of finite state machines were so simple that it was easy to see that the number of states that we used was the minimum possible to perform the required function. When a designer has to design a more complex FSM, it is likely that the initial attempt will result in a machine that has more states than is actually required. Minimizing the number of states is of interest because fewer flips-flops may be needed to represent the states and the complexity of the combinational circuit needed in the FSM may be reduced.



(a) Circuit



(b) Simulation Results

Figure 6.50 Synthesized serial adder.

If the number of states in an FSM can be reduced, then some states in the original design must be equivalent to other states in their contribution to the overall behavior of the FSM. We can express this more formally in the following definition.

Definition 6.1 – *Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.*

It is possible to define a minimization procedure that searches for any states that are equivalent. Such a procedure is very tedious to perform manually, but it can be automated for use in CAD tools. We will not pursue it here, because of its tediousness. However, to provide some appreciation of the impact of state minimization, we will present an alternative approach, which is much more efficient but not quite as broad in scope.

Instead of trying to show that some states in a given FSM are equivalent, it is often easier to show that some states are definitely **not** equivalent. This idea can be exploited to define a simple minimization procedure.

6.6.1 PARTITIONING MINIMIZATION PROCEDURE

Suppose that a state machine has a single input w. Then if the input signal $w = 0$ is applied to this machine in state S_i and the result is that the machine moves to state S_u, we will say that S_u is a 0-*successor* of S_i. Similarly, if $w = 1$ is applied in the state S_i and it causes the machine to move to state S_v, we will say that S_v is a 1-*successor* of S_i. In general, we will refer to the successors of S_i as its k-successors. When the FSM has only one input, k can be either 0 or 1. But if there are multiple inputs to the FSM, then k represents the set of all possible combinations (valuations) of the inputs.

From Definition 6.1 it follows that if the states S_i and S_j are equivalent, then their corresponding k-successors (for all k) are also equivalent. Using this fact, we can formulate a minimization procedure that involves considering the states of the machine as a set and then breaking the set into *partitions* that comprise subsets that are definitely not equivalent.

Definition 6.2 – *A partition consists of one or more blocks, where each block comprises a subset of states that may be equivalent, but the states in a given block are definitely not equivalent to the states in other blocks.*

Let us assume initially that all states are equivalent; this forms the initial partition, P_1, in which all states are in the same block. As the next step, we will form the partition P_2 in which the set of states is partitioned into blocks such that the states in each block generate the same output values. Obviously, the states that generate different outputs cannot possibly be equivalent. Then we will continue to form new partitions by testing whether the k-successors of the states in each block are contained in one block. Those states whose k-successors are in different blocks cannot be in one block. Thus new blocks are formed in each new partition. The process ends when a new partition is the same as the previous partition. Then all states in any one block are equivalent. To illustrate the procedure, consider Example 6.6.

Example 6.6

Figure 6.51 shows a state table for a particular FSM. In an attempt to minimize the number of states, let us apply the partitioning procedure. The initial partition contains all states in a single block

$$P_1 = (ABCDEFG)$$

The next partition separates the states that have different outputs (note that this FSM is of Moore type), which means that the states A, B, and D must be different from the states C, E, F, and G. Thus the new partition has two blocks

$$P_2 = (ABD)(CEFG)$$

Now we must consider all 0- and 1-successors of the states in each block. For the block (ABD), the 0-successors are (BDB), respectively. Since all of these successor states are in the same block in P_2, we should still assume that the states A, B, and D may be equivalent. The 1-successors for these states are (CFG). Since these successors are also in the same block in P_2, we conclude that (ABD) should remain in one block of P_3. Next consider the block $(CEFG)$. Its 0-successors are $(FFEF)$, respectively. They are in the same block in P_2. The 1-successors are $(ECDG)$. Since these states are not in the same block in P_2, it means that at least one of the states in the block $(CEFG)$ is not equivalent to the others. In particular, the state F must be different from the states C, E, and G because its 1-successor is D, which is in a different block than C, E, and G. Hence

$$P_3 = (ABD)(CEG)(F)$$

Repeating the process yields the following. The 0-successors of (ABD) are (BDB), which are in the same block of P_3. The 1-successors are (CFG), which are not in the same block. Since F is in a different block than C and G, it follows that the state B cannot be equivalent to states A and D. The 0- and 1-successors of (CEG) are (FFF) and (ECG), respectively. Both of these subsets are accommodated in the blocks of P_3. Therefore

$$P_4 = (AD)(B)(CEG)(F)$$

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Figure 6.51 State table for Example 6.6.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

Figure 6.52 Minimized state table for Example 6.6.

If we follow the same approach to check the 0- and 1-successors of the blocks (AD) and (CEG), we find that

$$P_5 = (AD)(B)(CEG)(F)$$

Since $P_5 = P_4$ and no new blocks are generated, it follows that states in each block are equivalent. If the states in some block were not equivalent, then their k-successors would have to be in different blocks. Therefore, states A and D are equivalent, and C, E, and G are equivalent. Since each block can be represented by a single state, only four states are needed to implement the FSM defined by the state table in Figure 6.51. If we let the symbol A represent both the states A and D in the figure and the symbol C represent the states C, E, and G, then the state table reduces to the state table in Figure 6.52.

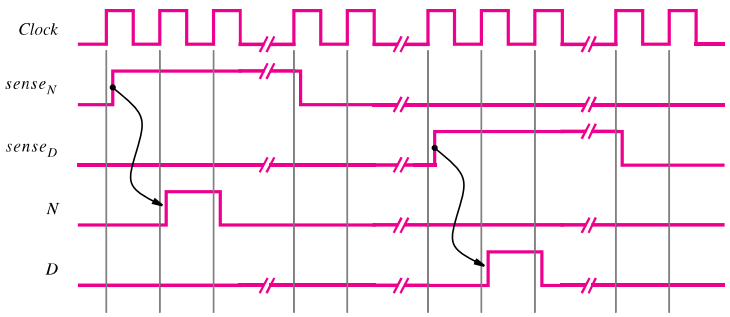
The effect of the minimization is that we have found a solution that requires only two flip-flops to realize the four states of the minimized state table, instead of needing three flip-flops for the original design. The expectation is that the FSM with fewer states will be simpler to implement, although this is not always the case.

The state minimization concept is based on the fact that two different FSMs may exhibit identical behavior in terms of the outputs produced in response to all possible inputs. Such machines are functionally equivalent, even though they are implemented with circuits that may be vastly different. In general, it is not easy to determine whether or not two arbitrary FSMs are equivalent. Our minimization procedure ensures that a simplified FSM is functionally equivalent to the original one. We encourage the reader to get an intuitive feeling that the FSMs in Figures 6.51 and 6.52 are indeed functionally equivalent by implementing both machines and simulating their behavior using the CAD tools.

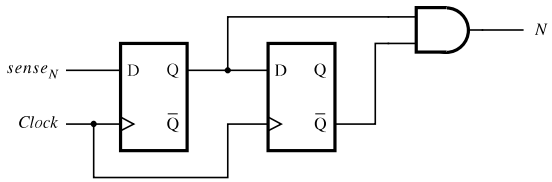
Example 6.7

As another example of minimization, we will consider the design of a sequential circuit that could control a vending machine. Suppose that a coin-operated vending machine dispenses candy under the following conditions:

- The machine accepts nickels and dimes.
- It takes 15 cents for a piece of candy to be released from the machine.



(a) Timing diagram



(b) Circuit that generates N

Figure 6.53 Signals for the vending machine.

- If 20 cents is deposited, the machine will not return the change, but it will credit the buyer with 5 cents and wait for the buyer to make a second purchase.

All electronic signals in the vending machine are synchronized to the positive edge of a clock signal, named *Clock*. The exact frequency of the clock signal is not important for our example, but we will assume a clock period of 100 ns. The vending machine's coin-receptor mechanism generates two signals, $sense_D$ and $sense_N$, which are asserted when a dime or a nickel is detected. Because the coin receptor is a mechanical device and thus very slow compared to an electronic circuit, inserting a coin causes $sense_D$ or $sense_N$ to be set to 1 for a large number of clock cycles. We will assume that the coin receptor also generates two other signals, named D and N. The D signal is set to 1 for one clock cycle after $sense_D$ becomes 1, and N is set to 1 for one clock cycle after $sense_N$ becomes 1. The timing relationships between *Clock*, $sense_D$, $sense_N$, D, and N are illustrated in Figure 6.53*a*. The hash marks

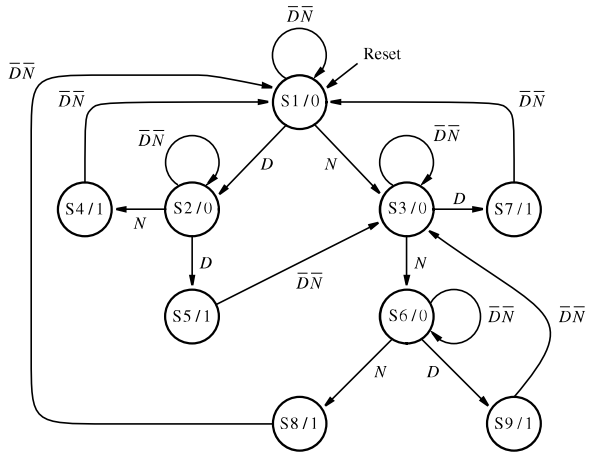


Figure 6.54 State diagram for Example 6.7.

on the waveforms indicate that $sense_D$ or $sense_N$ may be 1 for many clock cycles. Also, there may be an arbitrarily long time between the insertion of two consecutive coins. Note that since the coin receptor can accept only one coin at a time, it is not possible to have both D and N set to 1 at once. Figure 6.53*b* illustrates how the N signal may be generated from the $sense_N$ signal.

Based on these assumptions, we can develop an initial state diagram in a fairly straightforward manner, as indicated in Figure 6.54. The inputs to the FSM are D and N, and the starting state is $S1$. As long as $D = N = 0$, the machine remains in state $S1$, which is indicated by the arc labeled $\overline{D} \cdot \overline{N} = 1$. Inserting a dime leads to state $S2$, while inserting a nickel leads to state $S3$. In both cases the deposited amount is less than 15 cents, which is not sufficient to release the candy. This is indicated by the output, z, being equal to 0, as in $S2/0$ and $S3/0$. The machine will remain in state $S2$ or $S3$ until another coin is deposited because $D = N = 0$. In state $S2$ a nickel will cause a transition to $S4$ and a dime to $S5$. In both of these states, sufficient money is deposited to activate the output mechanism that releases the candy; hence the state nodes have the labels $S4/1$ and $S5/1$. In $S4$ the deposited amount is 15 cents, which means that on the next active clock edge the machine should return to the reset state $S1$. The condition $\overline{D} \cdot \overline{N}$ on the arc leaving $S4$ is guaranteed to be true because the machine remains in state $S4$ for only 100 ns, which is far too short a time for a new coin to have been deposited.

Present state	Next state				Output
	DN = 00	01	10	11	z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

Figure 6.55 State table for Example 6.7.

The state $S5$ denotes that an amount of 20 cents has been deposited. The candy is released, and on the next clock edge the FSM makes a transition to state $S3$, which represents a credit of 5 cents. A similar reasoning when the machine is in state $S3$ leads to states $S6$ through $S9$. This completes the state diagram for the desired FSM. A state table version of the same information is given in Figure 6.55.

Note that the condition $D = N = 1$ is denoted as don't care in the table. Note also other don't cares in states $S4$, $S5$, $S7$, $S8$, and $S9$. They correspond to cases where there is no need to check the D and N signals because the machine changes to another state in an amount of time that is too short for a new coin to have been inserted.

Using the minimization procedure, we obtain the following partitions

$$P_1 = (S1, S2, S3, S4, S5, S6, S7, S8, S9)$$
$$P_2 = (S1, S2, S3, S6)(S4, S5, S7, S8, S9)$$
$$P_3 = (S1)(S3)(S2, S6)(S4, S5, S7, S8, S9)$$
$$P_4 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$
$$P_5 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$

The final partition has five blocks. Let $S2$ denote its equivalence to $S6$, let $S4$ denote the same with respect to $S7$ and $S8$, and let $S5$ represent $S9$. This leads to the minimized state table in Figure 6.56. The actual circuit that implements this table can be designed as explained in the previous sections.

In this example we used a straightforward approach to derive the original state diagram, which we then minimized using the partitioning procedure. Figure 6.57 presents the information in the state table of Figure 6.56 in the form of a state diagram. Looking at this diagram, the reader can probably see that it may have been quite feasible to derive the optimized diagram directly, using the following reasoning. Suppose that the states correspond to the various amounts of money deposited. In particular, the states, $S1$, $S3$, $S2$,

Present state	Next state				Output
	DN = 00	01	10	11	z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S2	S4	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1

Figure 6.56 Minimized state table for Example 6.7.

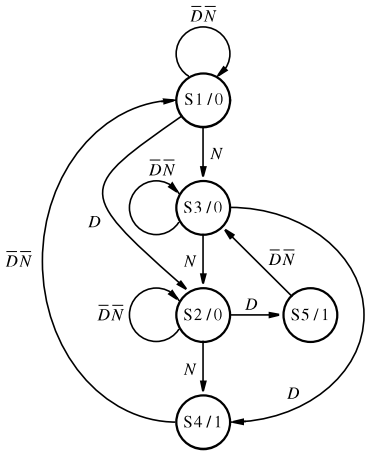


Figure 6.57 Minimized state diagram for Example 6.7.

$S4$, and $S5$ correspond to the amounts of 0, 5, 10, 15, and 20 cents, respectively. With this interpretation of the states, it is not difficult to derive the transition arcs that define the desired FSM. In practice, the designer can often produce initial designs that do not have a large number of superfluous states.

We have found a solution that requires five states, which is the minimum number of states for a Moore-type FSM that realizes the desired vending control task. From Section 6.3 we know that Mealy-type FSMs may need fewer states than Moore-type machines.

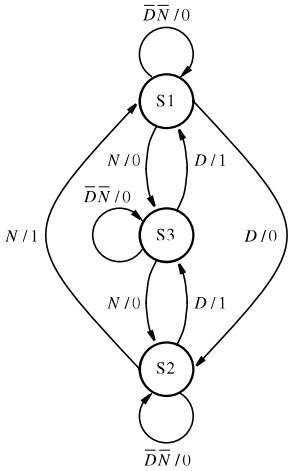


Figure 6.58 Mealy-type FSM for Example 6.7.

although they do not necessarily lead to simpler overall implementations. If we use the Mealy model, we can eliminate states $S4$ and $S5$ in Figure 6.57. The result is shown in Figure 6.58. This version requires only three states, but the output functions become more complicated. The reader is encouraged to compare the complexity of implementations by completing the design steps for the FSMs in Figures 6.57 and 6.58.

6.6.2 INCOMPLETELY SPECIFIED FSMS

The partitioning scheme for minimization of states works well when all entries in the state table are specified. Such is the case for the FSM defined in Figure 6.51. FSMs of this type are said to be *completely specified*. If one or more entries in the state table are not specified, corresponding to don't-care conditions, then the FSM is said to be *incompletely specified*. An example of such an FSM is given in Figure 6.55. As seen in Example 6.7, the partitioning scheme works well for this FSM also. But in general, the partitioning scheme is less useful when incompletely specified FSMs are involved, as illustrated by Example 6.8.

Example 6.8

Consider the FSM in Figure 6.59 which has four unspecified entries, because we have assumed that the input $w = 1$ will not occur when the machine is in states B or G. Accordingly,

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Figure 6.59 Incompletely specified state table for Example 6.8.

neither a state transition nor an output value is specified for these two cases. An important difference between this FSM and the one in Figure 6.55 is that some outputs in this FSM are unspecified, whereas in the other FSM all outputs are specified.

The partitioning minimization procedure can be applied to Mealy-type FSMs in the same way as for Moore-type FSMs illustrated in Examples 6.6 and 6.7. Two states are considered equivalent, and are thus placed in the same block of a partition, if their outputs are equal for all corresponding input valuations. To perform the partitioning process, we can assume that the unspecified outputs have a specific value. Not knowing whether these values should be 0 or 1, let us first assume that both unspecified outputs have a value of 0. Then the first two partitions are

$$P_1 = (ABCDEFG)$$
$$P_2 = (ABDG)(CEF)$$

Note that the states A, B, D, and G are in the same block because their outputs are equal to 0 for both $w = 0$ and $w = 1$. Also, the states C, E, and F are in one block because they have the same output behavior: they all generate $z = 0$ if $w = 0$, and $z = 1$ if $w = 1$. Continuing the partitioning procedure gives the remaining partitions

$$P_3 = (AB)(D)(G)(CE)(F)$$
$$P_4 = (A)(B)(D)(G)(CE)(F)$$
$$P_5 = P_4$$

The result is an FSM that is specified by six states.

Next consider the alternative of assuming that both unspecified outputs in Figure 6.59 have a value of 1. This would lead to the partitions

$$P_1 = (ABCDEFG)$$
$$P_2 = (AD)(BCEFG)$$
$$P_3 = (AD)(B)(CEFG)$$
$$P_4 = (AD)(B)(CEG)(F)$$
$$P_5 = P_4$$

This solution involves four states. Evidently, the choice of values assigned to unspecified outputs is of considerable importance.

We will not pursue the issue of state minimization of incompletely specified FSMs any further. As we already mentioned, it is possible to develop a minimization technique that searches for equivalent states based directly on Definition 6.1. This approach is described in many books on logic design [2, 8–10, 12–14].

Finally, it is important to mention that reducing the number of states in a given FSM will not necessarily lead to a simpler implementation. Interestingly, the effect of state assignment, discussed in Section 6.2, may have a greater influence on the simplicity of implementation than does the state minimization. In a modern design environment, the designer relies on the CAD tools to implement state machines efficiently.

6.7 DESIGN OF A COUNTER USING THE SEQUENTIAL CIRCUIT APPROACH

In this section we discuss the design of a counter circuit as a finite state machine. From Chapter 5 we already know that counters can be realized as cascaded stages of flip-flops and some gating logic, where each stage divides the number of incoming pulses by two. To keep our example simple, we choose a counter of small size but also show how the design can be extended to larger sizes. The specification for the counter is

- The counting sequence is 0, 1, 2, . . . , 6, 7, 0, 1, . . .
- There exists an input signal w. The value of this signal is considered during each clock cycle. If $w = 0$, the present count remains the same; if $w = 1$, the count is incremented.

The counter can be designed as a synchronous sequential circuit using the design techniques introduced in the previous sections.

6.7.1 STATE DIAGRAM AND STATE TABLE FOR A MODULO-8 COUNTER

Figure 6.60 gives a state diagram for the desired counter. There is a state associated with each count. In the diagram state A corresponds to count 0, state B to count 1, and so on. We show the transitions between the states needed to implement the counting sequence. Note that the output signals are specified as depending only on the state of the counter at a given time, which is the Moore model of sequential circuits.

The state diagram may be represented in the state-table form as shown in Figure 6.61.

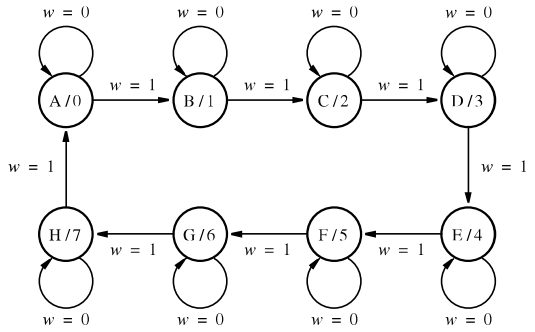


Figure 6.60 State diagram for the counter.

Present state	Next state		Output
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figure 6.61 State table for the counter.

6.7.2 STATE ASSIGNMENT

Three state variables are needed to represent the eight states. Let these variables, denoting the present state, be called y_2, y_1, and y_0. Let Y_2, Y_1, and Y_0 denote the corresponding next-state functions. The most convenient (and simplest) state assignment is to encode each state with the binary number that the counter should give as output in that state. Then the required output signals will be the same as the signals that represent the state variables. This leads to the state-assigned table in Figure 6.62.

	Present state $y_2y_1y_0$	Next state $w = 0$ $Y_2Y_1Y_0$	Next state $w = 1$ $Y_2Y_1Y_0$	Count $z_2z_1z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Figure 6.62 State-assigned table for the counter.

The final step in the design is to choose the type of flip-flops and derive the expressions that control the flip-flop inputs. The most straightforward choice is to use D-type flip-flops. We pursue this approach first. Then we show the alternative of using JK-type flip-flops.

6.7.3 IMPLEMENTATION USING D-TYPE FLIP-FLOPS

When using D-type flip-flops to realize the finite state machine, each next-state function, Y_i, is connected to the D input of the flip-flop that implements the state variable y_i. The next-state functions are derived from the information in Figure 6.62. Using Karnaugh maps in Figure 6.63, we obtain the following implementation

$$D_0 = Y_0 = \overline{w}y_0 + w\overline{y}_0$$
$$D_1 = Y_1 = \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1$$
$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2$$

The resulting circuit is given in Figure 6.64. It is not obvious how to extend this circuit to implement a larger counter, because no clear pattern is discernible in the expressions for D_0, D_1, and D_2. However, we can rewrite these expressions as follows

$$\begin{aligned} D_0 &= \overline{w}y_0 + w\overline{y}_0 \\ &= w \oplus y_0 \end{aligned}$$

$$\begin{aligned} D_1 &= \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1 \\ &= (\overline{w} + \overline{y}_0)y_1 + wy_0\overline{y}_1 \\ &= \overline{wy_0}y_1 + wy_0\overline{y}_1 \\ &= wy_0 \oplus y_1 \end{aligned}$$

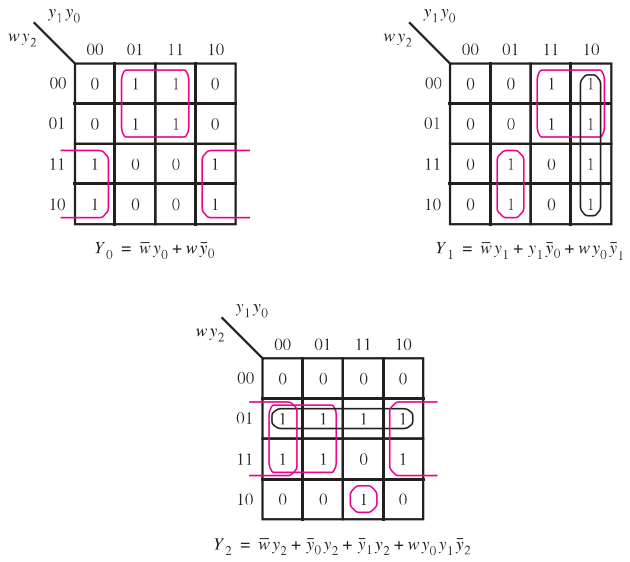


Figure 6.63 Karnaugh maps for D flip-flops for the counter.

$$\begin{aligned} D_2 &= \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2 \\ &= (\overline{w} + \overline{y}_0 + \overline{y}_1)y_2 + wy_0y_1\overline{y}_2 \\ &= \overline{wy_0y_1}y_2 + wy_0y_1\overline{y}_2 \\ &= wy_0y_1 \oplus y_2 \end{aligned}$$

Then an obvious pattern emerges, which leads to the circuit in Figure 5.23.

6.7.4 IMPLEMENTATION USING JK-TYPE FLIP-FLOPS

JK-type flip-flops provide an attractive alternative. Using these flip-flops to implement the sequential circuit specified in Figure 6.62 requires derivation of J and K inputs for each flip-flop. The following control is needed:

- If a flip-flop in state 0 is to remain in state 0, then $J = 0$ and $K = d$ (where d means that K can be equal to either 0 or 1).
- If a flip-flop in state 0 is to change to state 1, then $J = 1$ and $K = d$.

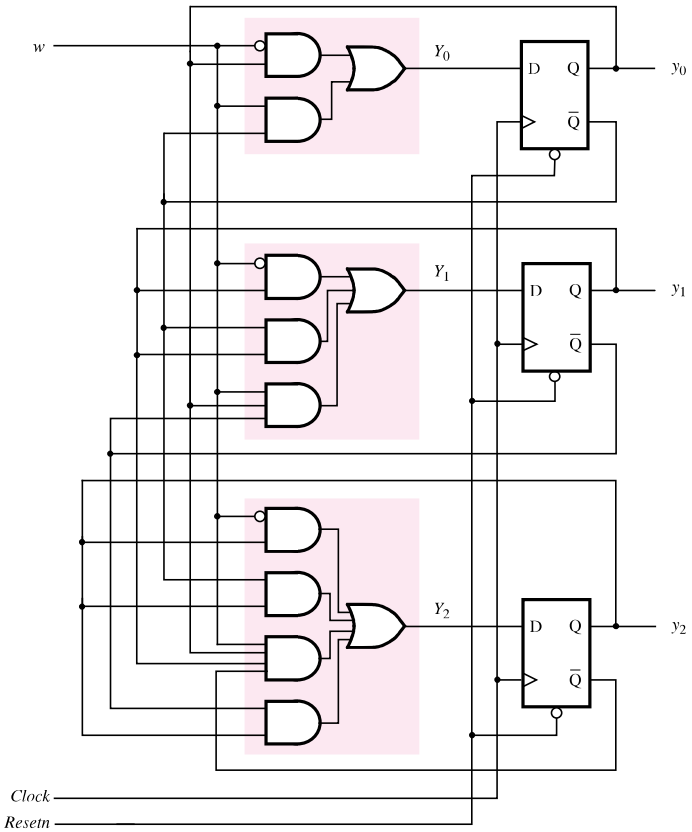


Figure 6.64 Circuit diagram for the counter implemented with D flip-flops.

- If a flip-flop in state 1 is to remain in state 1, then $J = d$ and $K = 0$.
- If a flip-flop in state 1 is to change to state 0, then $J = d$ and $K = 1$.

Following these guidelines, we can create a truth table that specifies the required values of the J and K inputs for the three flip-flops in our design. Figure 6.65 shows a modified version of the state-assigned table in Figure 6.62, with the J and K input functions included. To see how this table is derived, consider the first row in which the present state is

	Present state $y_2y_1y_0$	Flip-flop inputs								Count $z_2z_1z_0$
		$w = 0$				$w = 1$				
		$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	
A	000	000	$0d$	$0d$	$0d$	001	$0d$	$0d$	$1d$	000
B	001	001	$0d$	$0d$	$d0$	010	$0d$	$1d$	$d1$	001
C	010	010	$0d$	$d0$	$0d$	011	$0d$	$d0$	$1d$	010
D	011	011	$0d$	$d0$	$d0$	100	$1d$	$d1$	$d1$	011
E	100	100	$d0$	$0d$	$0d$	101	$d0$	$0d$	$1d$	100
F	101	101	$d0$	$0d$	$d0$	110	$d0$	$1d$	$d1$	101
G	110	110	$d0$	$d0$	$0d$	111	$d0$	$d0$	$1d$	110
H	111	111	$d0$	$d0$	$d0$	000	$d1$	$d1$	$d1$	111

Figure 6.65 Excitation table for the counter with JK flip-flops.

$y_2y_1y_0 = 000$. If $w = 0$, then the next state is also $Y_2Y_1Y_0 = 000$. Thus the present value of each flip-flop is 0, and it should remain 0. This implies the control $J = 0$ and $K = d$ for all three flip-flops. Continuing with the first row, if $w = 1$, the next state will be $Y_2Y_1Y_0 = 001$. Thus flip-flops y_2 and y_1 still remain at 0 and have the control $J = 0$ and $K = d$. However, flip-flop y_0 must change from 0 to 1, which is accomplished with $J = 1$ and $K = d$. The rest of the table is derived in the same manner by considering each present state $y_2y_1y_0$ and providing the necessary control signals to reach the new state $Y_2Y_1Y_0$.

A state-assigned table is essentially the state table in which each state is encoded using the state variables. When D flip-flops are used to implement an FSM, the next-state entries in the state-assigned table correspond directly to the signals that must be applied to the D inputs. This is not the case if some other type of flip-flops is used. A table that gives the state information in the form of the flip-flop inputs that must be "excited" to cause the transitions to the next states is usually called an *excitation table*. The excitation table in Figure 6.65 indicates how JK flip-flops can be used. In many books the term excitation table is used even when D flip-flops are involved, in which case it is synonymous with the state-assigned table.

Once the table in Figure 6.65 has been derived, it provides a truth table with inputs y_2, y_1, y_0, and w, and outputs J_2, K_2, J_1, K_1, J_0, and K_0. We can then derive expressions for these outputs as shown in Figure 6.66. The resulting expressions are

$$J_0 = K_0 = w$$
$$J_1 = K_1 = wy_0$$
$$J_2 = K_2 = wy_0y_1$$

This leads to the circuit shown in Figure 6.67. It is apparent that this design can be extended easily to larger counters. The pattern $J_n = K_n = wy_0y_1 \cdots y_{n-1}$ defines the circuit for each stage in the counter. Note that the size of the AND gate that implements the product term

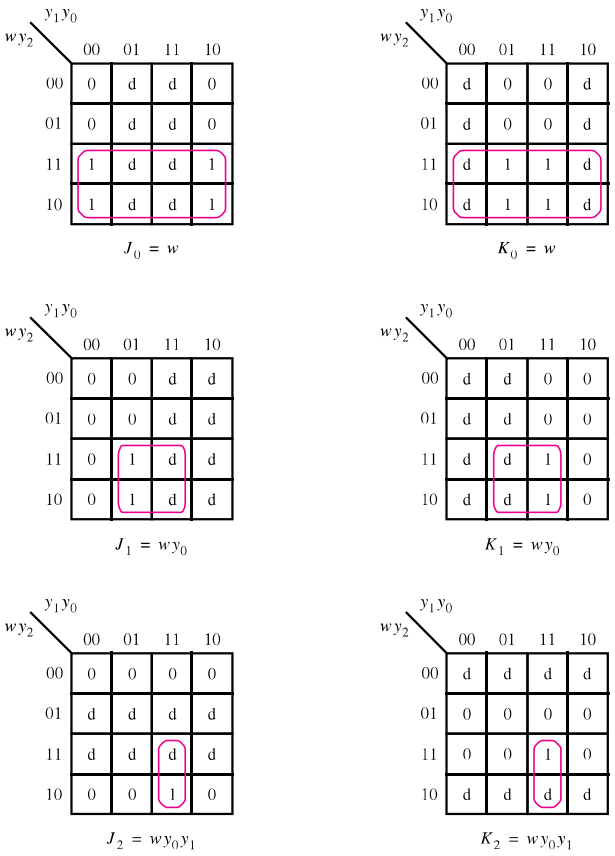


Figure 6.66 Karnaugh maps for JK flip-flops in the counter.

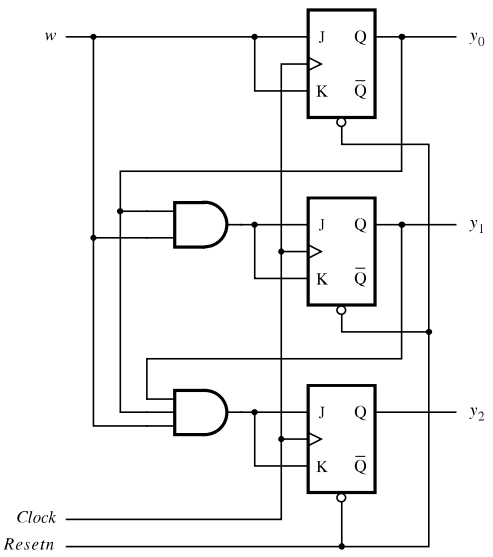


Figure 6.67 Circuit diagram using JK flip-flops.

$y_0y_1 \cdots y_{n-1}$ grows with successive stages. A circuit with a more regular structure can be obtained by factoring out the previously needed terms as we progress through the stages of the counter. This gives

$$J_2 = K_2 = (wy_0)y_1 \qquad = J_1y_1$$
$$J_n = K_n = (wy_0 \cdots y_{n-2})y_{n-1} = J_{n-1}y_{n-1}$$

Using the factored form, the counter circuit can be realized as indicated in Figure 6.68. In this circuit all stages (except the first) look the same. Note that this circuit has the same structure as the circuit in Figure 5.22 because connecting the J and K inputs of a flip-flop together turns the flip-flop into a T flip-flop.

6.7.5 EXAMPLE—A DIFFERENT COUNTER

Having considered the design of an ordinary counter, we will now apply this knowledge to design a slightly different counterlike circuit. Suppose that we wish to derive a three-bit counter that counts the pulses on an input line, w. But instead of displaying the count as 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, . . . , this counter must display the count in the sequence

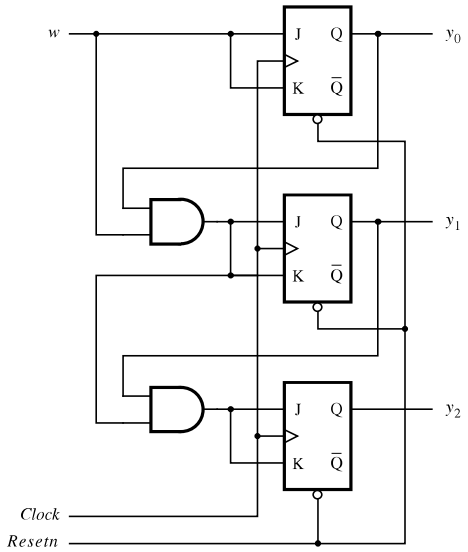


Figure 6.68 Factored-form implementation of the counter.

0, 4, 2, 6, 1, 5, 3, 7, 0, 4, and so on. The count is to be represented directly by the flip-flop values themselves, without using any extra gates. Namely, Count = $Q_2Q_1Q_0$.

Since we wish to count the pulses on the input line w, it makes sense to use w as the clock input to the flip-flops. Thus the counter circuit should always be enabled, and it should change its state whenever the next pulse on the w line appears. The desired counter can be designed in a straightforward manner using the FSM approach. Figures 6.69 and 6.70 give the required state table and a suitable state assignment. Using D flip-flops, we obtain the next-state equations

$$\begin{aligned} D_2 &= Y_2 = \overline{y}_2 \\ D_1 &= Y_1 = y_1 \oplus y_2 \\ D_0 &= Y_0 = y_0\overline{y}_1 + y_0\overline{y}_2 + \overline{y}_0y_1y_2 \\ &= y_0(\overline{y}_1 + \overline{y}_2) + \overline{y}_0y_1y_2 \\ &= y_0 \oplus y_1y_2 \end{aligned}$$

This leads to the circuit in Figure 6.71.

Present state	Next state	Output $z_2z_1z_0$
A	B	0 0 0
B	C	1 0 0
C	D	0 1 0
D	E	1 1 0
E	F	0 0 1
F	G	1 0 1
G	H	0 1 1
H	A	1 1 1

Figure 6.69 State table for the counterlike example.

Present state $y_2y_1y_0$	Next state $Y_2Y_1Y_0$	Output $z_2z_1z_0$
0 0 0	1 0 0	0 0 0
1 0 0	0 1 0	1 0 0
0 1 0	1 1 0	0 1 0
1 1 0	0 0 1	1 1 0
0 0 1	1 0 1	0 0 1
1 0 1	0 1 1	1 0 1
0 1 1	1 1 1	0 1 1
1 1 1	0 0 0	1 1 1

Figure 6.70 State-assigned table for Figure 6.69.

The reader should compare this circuit with the normal up-counter in Figure 5.23. Take the first three stages of that counter, set the *Enable* input to 1, and let *Clock* = w. Then the two circuits are essentially the same with one small difference in the order of bits in the count. In Figure 5.23 the top flip-flop corresponds to the least-significant bit of the count, whereas in Figure 6.71 the top flip-flop corresponds to the most-significant bit of the count. This is not just a coincidence. In Figure 6.70 the required count is defined as *Count* = $y_2y_1y_0$. However, if the bit patterns that define the states are viewed in the reverse order and interpreted as binary numbers, such that *Count* = $y_0y_1y_2$, then the states

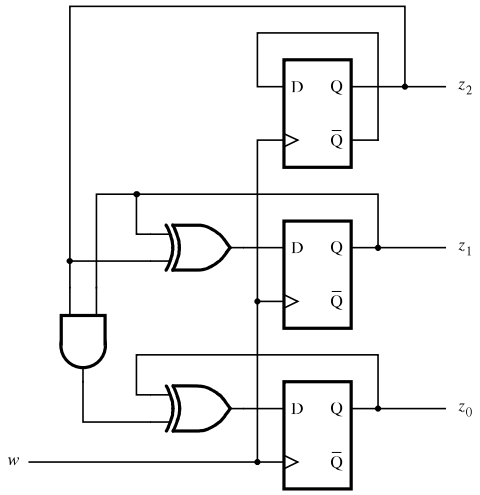


Figure 6.71 Circuit for Figure 6.70.

$A, B, C, \ldots, H$ have the values 0, 1, 2, ..., 7. These values are the same as the values that are associated with the normal three-bit up-counter.

6.8 FSM AS AN ARBITER CIRCUIT

In this section we present the design of an FSM that is slightly more complex than the previous examples. The purpose of the machine is to control access by various devices to a shared resource in a given system. Only one device can use the resource at a time. Assume that all signals in the system can change values only following the positive edge of the clock signal. Each device provides one input to the FSM, called a *request*, and the FSM produces a separate output for each device, called a *grant*. A device indicates its need to use the resource by asserting its request signal. Whenever the shared resource is not already in use, the FSM considers all requests that are active. Based on a priority scheme, it selects one of the requesting devices and asserts its grant signal. When the device is finished using the resource, it deasserts its request signal.

We will assume that there are three devices in the system, called device 1, device 2, and device 3. It is easy to see how the FSM can be extended to handle more devices. The request signals are named r_1, r_2, and r_3, and the grant signals are called g_1, g_2, and g_3. The devices are assigned a priority level such that device 1 has the highest priority, device 2 has the next highest, and device 3 has the lowest priority. Thus if more than one request signal is asserted when the FSM assigns a grant, the grant is given to the requesting device that has the highest priority.

A state diagram for the desired FSM, designed as a Moore-type machine, is depicted in Figure 6.72. Initially, on reset the machine is in the state called *Idle*. No grant signals are asserted, and the shared resource is not in use. There are three other states, called *gnt*1, *gnt*2, and *gnt*3. Each of these states asserts the grant signal for one of the devices.

The FSM remains in the *Idle* state as long as all of the request signals are 0. In the state diagram the condition $r_1r_2r_3 = 000$ is indicated by the arc labeled 000. When one or more request signals become 1, the machine moves to one of the grant states, according to the priority scheme. If r_1 is asserted, then device 1 will receive the grant because it has the highest priority. This is indicated by the arc labeled 1xx that leads to state *gnt*1, which sets $g_1 = 1$. The meaning of 1xx is that the request signal r_1 is 1, and the values of signals

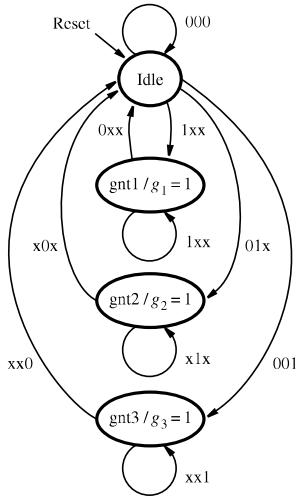


Figure 6.72 State diagram for the arbiter.

r_2 and r_3 are irrelevant because of the priority scheme. As before, we use the symbol x to indicate that the value of the corresponding variable can be either 0 or 1. The machine stays in state *gnt*1 as long as r_1 is 1. When $r_1 = 0$, the arc labeled 0xx causes a change on the next positive clock edge back to state *Idle*. and g_1 is deasserted. If other requests are active at this time, then the FSM will change to a new grant state after the next clock edge.

The arc that causes a change to state *gnt*2 is labeled 01x. This label adheres to the priority scheme because it represents the condition that $r_2 = 1$, but $r_1 = 0$. Similarly, the condition for entering state *gnt*3 is given as 001, which indicates that the only request signal asserted is r_3.

The state diagram is repeated in Figure 6.73. The only difference between this diagram and Figure 6.72 is the way in which the arcs are labeled. Figure 6.73 uses a simpler labeling scheme that is more intuitive. For the condition that leads from state *Idle* to state *gnt*1, the arc is labeled as r_1, instead of 1xx. This label means that if $r_1 = 1$, the FSM changes to state *gnt*1, regardless of the other inputs. The arc with the label $\bar{r}_1 r_2$ that leads from state *Idle* to *gnt*2 represents the condition $r_1 r_2 = 01$, while the value of r_3 is irrelevant. There is no standardized scheme for labeling the arcs in state diagrams. Some designers prefer the style of Figure 6.72, while others prefer a style more similar to Figure 6.73.

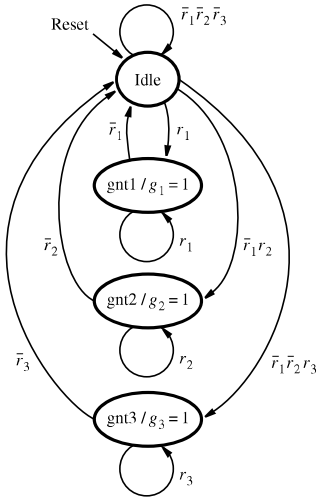


Figure 6.73 Alternative style of state diagram for the arbiter.

Figure 6.74 gives the Verilog code for the machine. The three request and grant signals are specified as three-bit vectors r and g. The FSM transitions are described using a **case** statement in the style used for Figure 6.29. In the state *Idle*, the required priority scheme is described by using a nested **casex** statement. If $r_1 = 1$, then the next state for the machine is *gnt*1. If r_1 is not asserted, then the alternative is evaluated, which stipulates that if $r_2 = 1$,

```
module arbiter (r, Resetn, Clock, g);
   input [1:3] r;
   input Resetn, Clock;
   output wire [1:3] g;
   reg [2:1] y, Y;
   parameter Idle = 2'b00, gnt1 = 2'b01, gnt2 = 2'b10, gnt3 = 2'b11;

   // Next state combinational circuit
   always @(r, y)
      case (y)
         Idle:  casex (r)
                   3'b000:  Y = Idle;
                   3'b1xx:  Y = gnt1;
                   3'b01x:  Y = gnt2;
                   3'b001:  Y = gnt3;
                   default: Y = Idle;
                endcase
         gnt1:  if (r[1])  Y = gnt1;
                else       Y = Idle;
         gnt2:  if (r[2])  Y = gnt2;
                else       Y = Idle;
         gnt3:  if (r[3])  Y = gnt3;
                else       Y = Idle;
         default:          Y = Idle;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Resetn == 0)  y <= Idle;
      else  y <= Y;

   // Define output
   assign g[1] = (y == gnt1);
   assign g[2] = (y == gnt2);
   assign g[3] = (y == gnt3);

endmodule
```

Figure 6.74 Verilog code for the arbiter.

then the next state will be *gnt*2. Each successive alternative considers a lower-priority request signal only if all of the higher-priority request signals are not asserted.

The transitions for each grant state are straightforward. The FSM stays in state *gnt*1 as long as $r_1 = 1$; when $r_1 = 0$, the next state is *Idle*. The other grant states have the same structure.

The grant signals, g_1, g_2, and g_3 are defined at the end. The value of g_1 is set to 1 when the machine is in state *gnt*1, and otherwise g_1 is set to 0. Similarly, each of the other grant signals is 1 only in the appropriate grant state.

6.9 ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUITS

In addition to knowing how to design a synchronous sequential circuit, the designer has to be able to analyze the behavior of an existing circuit. The analysis task is much simpler than the synthesis task. In this section we will show how analysis may be performed.

To analyze a circuit, we simply reverse the steps of the synthesis process. The outputs of the flip-flops represent the present-state variables. Their inputs determine the next state that the circuit will enter. From this information we can construct the state-assigned table for the circuit. This table leads to a state table and the corresponding state diagram by giving a name to each state. The type of flip-flops used in the circuit is a factor, as we will see in the examples that follow.

Example 6.9

D-TYPE FLIP-FLOPS Figure 6.75 gives an FSM that has two D flip-flops. Let y_1 and y_2 be the present-state variables and Y_1 and Y_2 the next-state variables. The next-state and output expressions are

$$Y_1 = w\bar{y}_1 + wy_2$$
$$Y_2 = wy_1 + wy_2$$
$$z = y_1y_2$$

Since there are two flip-flops, the FSM has at most four states. A good starting point in the analysis is to assume an initial state of the flip-flops such as $y_1 = y_2 = 0$. From the expressions for Y_1 and Y_2, we can derive the state-assigned table in Figure 6.76*a*. For example, in the first row of the table $y_1 = y_2 = 0$. Then $w = 0$ causes $Y_1 = Y_2 = 0$, and $w = 1$ causes $Y_1 = 1$ and $Y_2 = 0$. The output for this state is $z = 0$. The other rows are derived in the same manner. Labeling the states as *A*, *B*, *C*, and *D* yields the state table in Figure 6.76*b*. From this table it is apparent that following the reset condition the FSM produces the output $z = 1$ whenever three consecutive 1s occur on the input w. Therefore, the FSM acts as a sequence detector for this pattern.

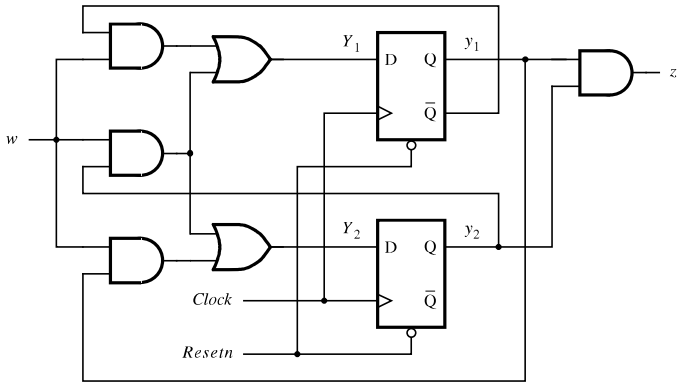


Figure 6.75 Circuit for Example 6.9.

Present state y_2y_1	Next state $w = 0$ Y_2Y_1	Next state $w = 1$ Y_2Y_1	Output z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	0 0	1 1	0
1 1	0 0	1 1	1

(a) State-assigned table

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

(b) State table

Figure 6.76 Tables for the circuit in Figure 6.75.

JK-TYPE FLIP-FLOPS Now consider the circuit in Figure 6.77, which has two JK flip-flops. **Example 6.10**
The expressions for the inputs to the flip-flops are

$$J_1 = w$$
$$K_1 = \overline{w} + \overline{y}_2$$
$$J_2 = wy_1$$
$$K_2 = \overline{w}$$

The output is given by $z = y_1 y_2$.

From these expressions we can derive the excitation table in Figure 6.78. Interpreting the entries in this table, we can construct the state-assigned table. For example, consider $y_2y_1 = 00$ and $w = 0$. Then, since $J_2 = J_1 = 0$ and $K_2 = K_1 = 1$, both flip-flops will remain in the 0 state; hence $Y_2 = Y_1 = 0$. If $y_2y_1 = 00$ and $w = 1$, then $J_2 = K_2 = 0$ and $J_1 = K_1 = 1$, which leaves the y_2 flip-flop unchanged and sets the y_1 flip-flop to 1; hence $Y_2 = 0$ and $Y_1 = 1$. If $y_2y_1 = 01$ and $w = 0$, then $J_2 = J_1 = 0$ and $K_2 = K_1 = 1$, which resets the y_1 flip-flop and results in the state $y_2y_1 = 00$; hence $Y_2 = Y_1 = 0$. Similarly, if $y_2y_1 = 01$ and $w = 1$, then $J_2 = 1$ and $K_2 = 0$ sets y_2 to 1; hence $Y_2 = 1$, while $J_1 = K_1 = 1$ toggles y_1; hence $Y_1 = 0$. This leads to the state $y_2y_1 = 10$. Completing this process, we find that the resulting state-assigned table is the same as the one in Figure 6.76*a*. The conclusion is that the circuits in Figures 6.75 and 6.77 implement the same FSM.

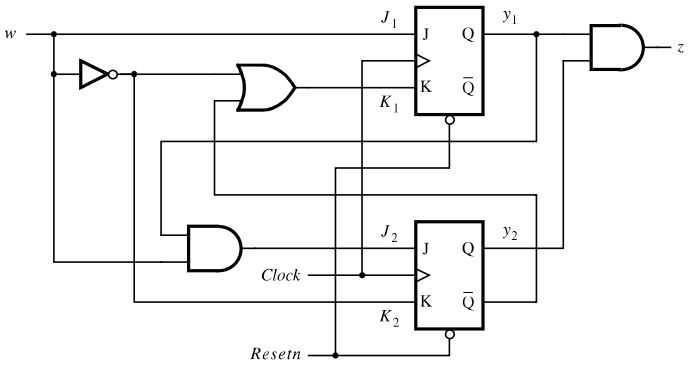


Figure 6.77 Circuit for Example 6.10.

Present state y_2y_1	Flip-flop inputs $w = 0$ J_2K_2	J_1K_1	$w = 1$ J_2K_2	J_1K_1	Output z
0 0	0 1	0 1	0 0	1 1	0
0 1	0 1	0 1	1 0	1 1	0
1 0	0 1	0 1	0 0	1 0	0
1 1	0 1	0 1	1 0	1 0	1

Figure 6.78 The excitation table for the circuit in Figure 6.77.

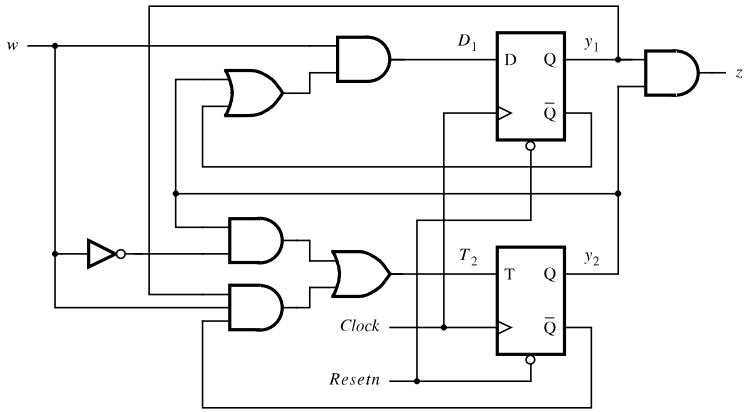


Figure 6.79 Circuit for Example 6.11.

Example 6.11 **MIXED FLIP-FLOPS** There is no reason why one cannot use a mixture of flip-flop types in one circuit. Figure 6.79 shows a circuit with one D and one T flip-flop. The expressions for this circuit are

$$D_1 = w(\overline{y}_1 + y_2)$$
$$T_2 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$z = y_1y_2$$

Present state y_2y_1	Flip-flop inputs $w = 0$ T_2D_1	Flip-flop inputs $w = 1$ T_2D_1	Output z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	1 0	0 1	0
1 1	1 0	0 1	1

Figure 6.80 The excitation table for the circuit in Figure 6.79.

From these expressions we derive the excitation table in Figure 6.80. Since it is a T flip-flop, y_2 changes its state only when $T_2 = 1$. Thus if $y_2y_1 = 00$ and $w = 0$, then because $T_2 = D_1 = 0$ the state of the circuit will not change. An example of where $T_2 = 1$ is when $y_2y_1 = 01$ and $w = 1$, which causes y_2 to change to 1; $D_1 = 0$ makes $y_1 = 0$, hence $Y_2 = 1$ and $Y_1 = 0$. The other cases where $T_2 = 1$ occur when $w = 0$ and $y_2y_1 = 10$ or 11. In both of these cases $D_1 = 0$. Hence the T flip-flop changes its state from 1 to 0, while the D flip-flop is cleared, which means that the next state is $Y_2Y_1 = 00$. Completing this analysis we again obtain the state-assigned table in Figure 6.76*a*. Thus this circuit is yet another implementation of the FSM represented by the state table in Figure 6.76*b*.

6.10 ALGORITHMIC STATE MACHINE (ASM) CHARTS

The state diagrams and tables used in this chapter are convenient for describing the behavior of FSMs that have only a few inputs and outputs. For larger machines the designers often use a different form of representation, called the *algorithmic state machine (ASM) chart*.

An ASM chart is a type of flowchart that can be used to represent the state transitions and generated outputs for an FSM. The three types of elements used in ASM charts are depicted in Figure 6.81.

- **State box** – A rectangle represents a state of the FSM. It is equivalent to a node in the state diagram or a row in the state table. The name of the state is indicated outside the box in the top-left corner. The Moore-type outputs are listed inside the box. These are the outputs that depend only on the values of the state variables that define the state; we will refer to them simply as *Moore outputs*. It is customary to write only the name of the signal that has to be asserted. Thus it is sufficient to write z, rather than $z = 1$, to indicate that the output z must have the value 1. Also, it may be useful to indicate an action that must be taken; for example, $Count \leftarrow Count + 1$ specifies that the contents of a counter have to be incremented by 1. Of course, this is just a simple way of saying that the control signal that causes the counter to be incremented must be asserted. We will use this way of specifying actions in larger systems that are discussed in Chapter 7.

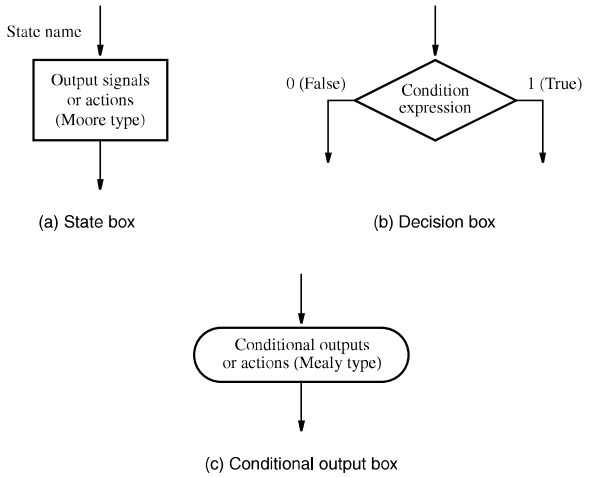


Figure 6.81 Elements used in ASM charts.

- **Decision box** – A diamond indicates that the stated condition expression is to be tested and the exit path is to be chosen accordingly. The condition expression consists of one or more inputs to the FSM. For example, w indicates that the decision is based on the value of the input w, whereas $w_1 \cdot w_2$ indicates that the true path is taken if $w_1 = w_2 = 1$ and the false path is taken otherwise.
- **Conditional output box** – An oval denotes the output signals that are of Mealy type. These outputs depend on the values of the state variables and the inputs of the FSM; we will refer to these outputs simply as *Mealy outputs*. The condition that determines whether such outputs are generated is specified in a decision box.

Figure 6.82 gives the ASM chart that represents the FSM in Figure 6.3. The transitions between state boxes depend on the decisions made by testing the value of the input variable w. In each case if $w = 0$, the exit path from a decision box leads to state A. If $w = 1$, then a transition from A to B or from B to C takes place. If $w = 1$ in state C, then the FSM stays in that state. The chart specifies a Moore output z, which is asserted only in state C,

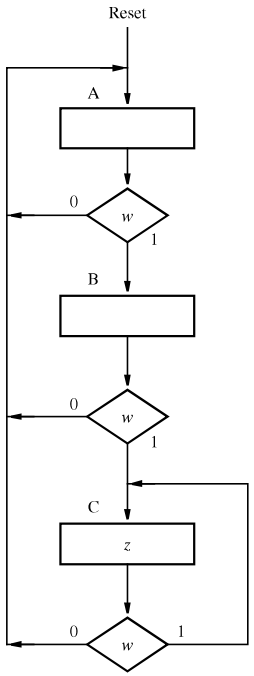


Figure 6.82 ASM chart for the FSM in Figure 6.3.

as indicated in the state box. In states A and B, the value of z is 0 (not asserted), which is implied by leaving the corresponding state boxes blank.

Figure 6.83 provides an example with Mealy outputs. This chart represents the FSM in Figure 6.23. The output, z, is equal to 1 when the machine is in state B and $w = 1$. This is indicated using the conditional output box. In all other cases the value of z is 0, which is implied by not specifying z as an output of state B for $w = 0$ and state A for w equal to 0 or 1.

Figure 6.84 gives the ASM chart for the arbiter FSM in Figure 6.73. The decision box drawn below the state box for *Idle* specifies that if $r_1 = 1$, then the FSM changes to state *gnt1*. In this state the FSM asserts the output signal g_1. The decision box to the right of

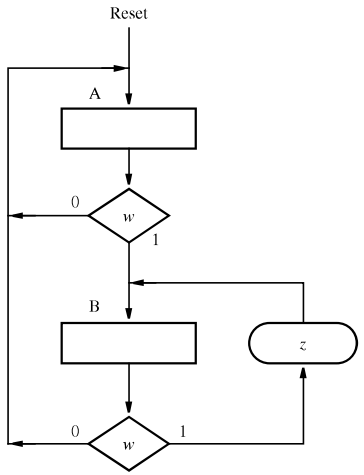


Figure 6.83 ASM chart for the FSM in Figure 6.23.

the state box for *gnt1* specifies that as long as $r_1 = 1$, the machine stays in state *gnt1*, and when $r_1 = 0$, it changes to state *Idle*. The decision box labeled r_2 that is drawn below the state box for *Idle* specifies that if $r_2 = 1$, then the FSM changes to state *gnt2*. This decision box can be reached only after first checking the value of r_1 and following the arrow that corresponds to $r_1 = 0$. Similarly, the decision box labeled r_3 can be reached only if both r_1 and r_2 have the value 0. Hence the ASM chart describes the required priority scheme for the arbiter.

ASM charts are similar to traditional flowcharts. Unlike a traditional flowchart, the ASM chart includes timing information because it implicitly specifies that the FSM changes (flows) from one state to another only after each active clock edge. The examples of ASM charts presented here are quite simple. We have used them to introduce the ASM chart terminology by giving examples of state, decision, and conditional-output boxes. Another term sometimes applied to ASM charts is *ASM block*, which refers to a single state box and any decision and conditional-output boxes that the state box may be connected to. The ASM charts can be used to describe complex circuits that include one or more finite state machines and other circuitry such as registers, shift registers, counters, adders, and multipliers. We will use ASM charts as an aid for designing more complex circuits in Chapter 7.

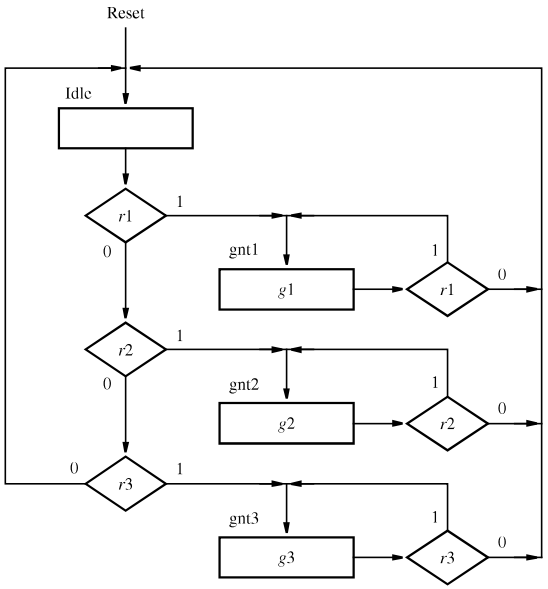


Figure 6.84 ASM chart for the arbiter FSM in Figure 6.73.

6.11 FORMAL MODEL FOR SEQUENTIAL CIRCUITS

This chapter has presented the synchronous sequential circuits using a rather informal approach because this is the easiest way to grasp the concepts that are essential in designing such circuits. The same topics can also be presented in a more formal manner, which has been the style adopted in many books that emphasize the switching theory aspects rather than the design using CAD tools. A formal model often gives a concise specification that is difficult to match in a more descriptive presentation. In this section we will describe a formal model that represents a general class of sequential circuits, including those of the synchronous type.

Figure 6.85 represents a general sequential circuit. The circuit has $W = \{w_1, w_2, \ldots, w_n\}$ inputs, $Z = \{z_1, z_2, \ldots, z_m\}$ outputs, $y = \{y_1, y_2, \ldots, y_k\}$ present-state variables, and $Y = \{Y_1, Y_2, \ldots, Y_k\}$ next-state variables. It can have up to 2^k states, $S = \{S_1, S_2, \ldots, S_{2^k}\}$.

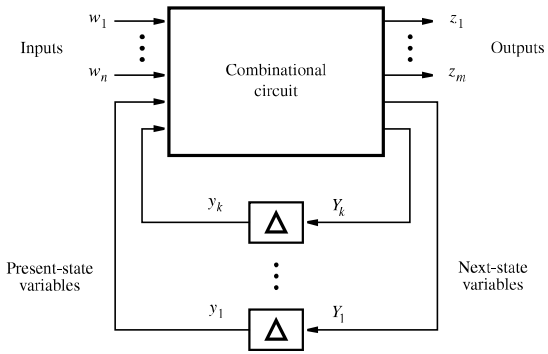


Figure 6.85 The general model for a sequential circuit.

There are delay elements in the feedback paths for the state-variables which ensure that y will take the values of Y after a time delay Δ. In the case of synchronous sequential circuits, the delay elements are flip-flops, which change their state on the active edge of a clock signal. Thus the delay Δ is determined by the clock period. The clock period must be long enough to allow for the propagation delay in the combinational circuit, in addition to the setup and hold parameters of the flip-flops.

Using the model in Figure 6.85, a synchronous sequential circuit, M, can be defined formally as a quintuple

$$M = (W, Z, S, \varphi, \lambda)$$

where

- W, Z, and S are finite, nonempty sets of inputs, outputs, and states, respectively.
- φ is the state transition function, such that $S(t+1) = \varphi[W(t), S(t)]$.
- λ is the output function, such that $\lambda(t) = \lambda[S(t)]$ for the Moore model and $\lambda(t) = \lambda[W(t), S(t)]$ for the Mealy model.

This definition assumes that the time between t and $t+1$ is one clock cycle.

We will see in Chapter 9 that the delay Δ need not be controlled by a clock. In asynchronous sequential circuits the delays are due solely to the propagation delays through various gates.

6.12 CONCLUDING REMARKS

The existence of closed loops and delays in a sequential circuit leads to a behavior that is characterized by the set of states that the circuit can reach. The present values of the inputs are not the sole determining factor in this behavior, because a given valuation of inputs may cause the circuit to behave differently in different states.

The propagation delays through a sequential circuit must be taken into account. The design techniques presented in this chapter are based on the assumption that all changes in the circuit are triggered by the active edge of a clock signal. Such circuits work correctly only if all internal signals are stable when the clock signal arrives. Thus the clock period must be longer than the longest propagation delay in the circuit.

Synchronous sequential circuits are used extensively in practical designs. They are supported by the commonly used CAD tools. All textbooks on the design of logic circuits devote considerable space to synchronous sequential circuits. Some of the more notable references are [1–14].

In Chapter 9 we will present a different class of sequential circuits, which do not use flip-flops to represent the states of the circuit and do not use clock pulses to trigger changes in the states.

6.13 EXAMPLES OF SOLVED PROBLEMS

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

Example 6.12

Problem: Design an FSM that has an input w and an output z. The machine is a sequence detector that produces $z = 1$ when the previous two values of w were 00 or 11; otherwise $z = 0$.

Solution: Section 6.1 presents the design of an FSM that detects the occurrence of consecutive 1s. Using the same approach, the desired FSM can be specified using the state diagram in Figure 6.86. State C denotes the occurrence of two or more 0s, and state E denotes two or more 1s. The corresponding state table is shown in Figure 6.87.

A straightforward state assignment leads to the state-assigned table in Figure 6.88. The codes $y_3y_2y_1 = 101, 110, 111$ can be treated as don't-care conditions. Then the next-state expressions are

$$Y_1 = w\overline{y}_1\overline{y}_3 + w\overline{y}_2\overline{y}_3 + \overline{w}y_1y_2 + \overline{w}y_1\overline{y}_2$$
$$Y_2 = y_1\overline{y}_2 + \overline{y}_1y_2 + w\overline{y}_2\overline{y}_3$$
$$Y_3 = wy_3 + wy_1y_2$$

The output expression is

$$z = y_3 + \overline{y}_1y_2$$

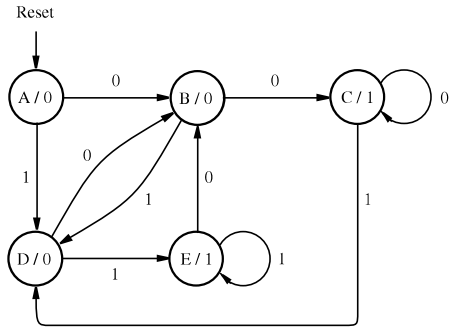


Figure 6.86 State diagram for Example 6.12.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	D	0
B	C	D	0
C	C	D	1
D	B	E	0
E	B	E	1

Figure 6.87 State table for the FSM in Figure 6.86.

These expressions seem to be unnecessarily complex, suggesting that we may attempt to find a better state assignment. Observe that state A is reached only when the machine is reset by means of the *Reset* input. So, it may be advantageous to assign the four codes in which $y_3 = 1$ to the states B, C, D, and E. The result is the state-assigned table in Figure 6.89. From it, the next-state and output expressions are

$$Y_1 = wy_2 + \overline{w}y_3\overline{y}_2$$
$$Y_2 = w$$
$$Y_3 = 1$$
$$z = y_1$$

Since $Y_3 = 1$, we do not need the flip-flop y_3, and the expression for Y_1 simplifies to $\overline{w \oplus y_2}$. This is a much better solution.

	Present state $y_3y_2y_1$	Next state $w = 0$ $Y_3Y_2Y_1$	Next state $w = 1$ $Y_3Y_2Y_1$	Output z
A	000	001	011	0
B	001	010	011	0
C	010	010	011	1
D	011	001	100	0
E	100	001	100	1

Figure 6.88 State-assigned table for the FSM in Figure 6.87.

	Present state $y_3y_2y_1$	Next state $w = 0$ $Y_3Y_2Y_1$	Next state $w = 1$ $Y_3Y_2Y_1$	Output z
A	000	100	110	0
B	100	101	110	0
C	101	101	110	1
D	110	100	111	0
E	111	100	111	1

Figure 6.89 An improved state assignment for the FSM in Figure 6.87.

Example 6.13

Problem: Implement the sequence detector of Example 6.12 by using two FSMs. One FSM detects the occurrence of consecutive 1s, while the other detects consecutive 0s.

Solution: A good realization of the FSM that detects consecutive 1s is given in Figure 6.17. The next-state and output expressions are

$$Y_1 = w$$
$$Y_2 = wy_1$$
$$z_{ones} = y_2$$

A similar FSM that detects consecutive 0s is defined in Figure 6.90. Its expressions are

$$Y_3 = \overline{w}$$
$$Y_4 = \overline{w}y_3$$
$$z_{zeros} = y_4$$

Present state	Next state $w = 0$	Next state $w = 1$	Output z_{zeros}
D	E	D	0
E	F	D	0
F	F	D	1

(a) State table

	Present state y_4y_3	Next state $w = 0$ Y_4Y_3	Next state $w = 1$ Y_4Y_3	Output z_{zeros}
D	00	01	00	0
E	01	11	00	0
F	11	11	00	1
	10	dd	dd	d

(b) State-assigned table

Figure 6.90 FSM that detects a sequence of two zeros.

The output of the combined circuit is

$$z = z_{ones} + z_{zeros}$$

Example 6.14

Problem: Derive a Mealy-type FSM that can act as a sequence detector described in Example 6.12.

Solution: A state diagram for the desired FSM is depicted in Figure 6.91. The corresponding state table is presented in Figure 6.92. Two flip-flops are needed to implement this FSM. A state-assigned table is given in Figure 6.93, which leads to the next-state and output expressions

$$Y_1 = 1$$
$$Y_2 = w$$
$$z = \overline{w}y_1\overline{y}_2 + wy_2$$

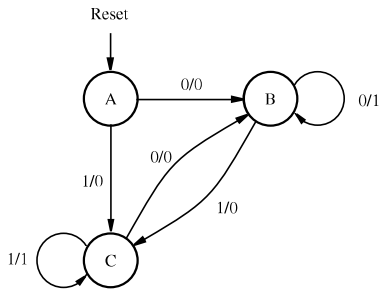


Figure 6.91 State diagram for Example 6.14.

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	B	C	1	0
C	B	C	0	1

Figure 6.92 State table for the FSM in Figure 6.91.

	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y_2y_1	Y_2Y_1	Y_2Y_1	z	z
A	00	01	11	0	0
B	01	01	11	1	0
C	11	01	11	0	1

Figure 6.93 State-assigned table for the FSM in Figure 6.92.

Example 6.15 **Problem:** Implement the state-assigned table in Figure 6.89 using JK-type flip-flops.

Solution: Figure 6.94 shows the excitation table. It results in the following next-state and output expressions

$$J_1 = wy_2 + \overline{w}y_3\overline{y}_2$$
$$K_1 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$J_2 = w$$
$$K_2 = \overline{w}$$
$$J_3 = 1$$
$$K_3 = 0$$
$$z = y_1$$

Example 6.16 **Problem:** Write Verilog code to implement the FSM in Figure 6.86.

Solution: Using the style of code given in Figure 6.29, the required FSM can be specified as shown in Figure 6.95.

Example 6.17 **Problem:** Write Verilog code to implement the FSM in Figure 6.91.

Solution: Using the style of code given in Figure 6.36, the Mealy-type FSM can be specified as shown in Figure 6.96.

Example 6.18 **Problem:** In computer systems it is often desirable to transmit data serially, namely, one bit at a time, to save on the cost of interconnecting cables. This means that parallel data at

	Present state $y_3y_2y_1$	Flip-flop inputs								Output z
		$w = 0$				$w = 1$				
		$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	
A	000	100	$1d$	$0d$	$0d$	110	$1d$	$1d$	$0d$	0
B	100	101	$d0$	$0d$	$1d$	110	$d0$	$1d$	$0d$	0
C	101	101	$d0$	$0d$	$d0$	110	$d0$	$1d$	$d1$	1
D	110	100	$d0$	$d1$	$0d$	111	$d0$	$d0$	$1d$	0
E	111	100	$d0$	$d1$	$d1$	111	$d0$	$d0$	$d0$	1

Figure 6.94 Excitation table for the FSM in Figure 6.89 with JK flip-flops.

```
module sequence (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [3:1] y, Y;
   parameter [3:1] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A: if (w)    Y = D;
            else      Y = B;
         B: if (w)    Y = D;
            else      Y = C;
         C: if (w)    Y = D;
            else      Y = C;
         D: if (w)    Y = E;
            else      Y = B;
         E: if (w)    Y = E;
            else      Y = B;
         default:     Y = 3'bxxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)   y <= A;
      else   y <= Y;

   // Define output
   assign z = (y == C) | (y == E);

endmodule
```

Figure 6.95 Verilog code for the FSM in Figure 6.86.

one end must be transmitted serially, and at the other end the received serial data has to be turned back into parallel form. Suppose that we wish to transmit ASCII characters in this manner. As explained in Chapter 1, the standard ASCII code uses seven bits to define each character. Usually, a character occupies one byte, in which case the eighth bit can either be set to 0 or it can be used to indicate the parity of the other bits to ensure a more reliable transmission.

Parallel-to-serial conversion can be done by means of a shift register. Assume that a circuit accepts parallel data, $B = b_7, b_6, \ldots, b_0$, representing ASCII characters. Assume also that bit b_7 is set to 0. The circuit is supposed to generate a parity bit, p, and send it instead of b_7 as a part of the serial transfer. Figure 6.97 gives a possible circuit. An FSM is


```
module seqmealy (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output reg z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b11;

   // Define the next state and output combinational circuits
   always @(w, y)
      case (y)
         A:  if (w)
             begin
                z = 0; Y = C;
             end
             else
             begin
                z = 0; Y = B;
             end
         B:  if (w)
             begin
                z = 0; Y = C;
             end
             else
             begin
                z = 1; Y = B;
             end
         C:  if (w)
             begin
                z = 1; Y = C;
             end
             else
             begin
                z = 0; Y = B;
             end
         default:
             begin
                z = 0; Y = 2'bxx;
             end
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)   y <= A;
      else   y <= Y;

endmodule
```

Figure 6.96 Verilog code for the FSM in Figure 6.91.

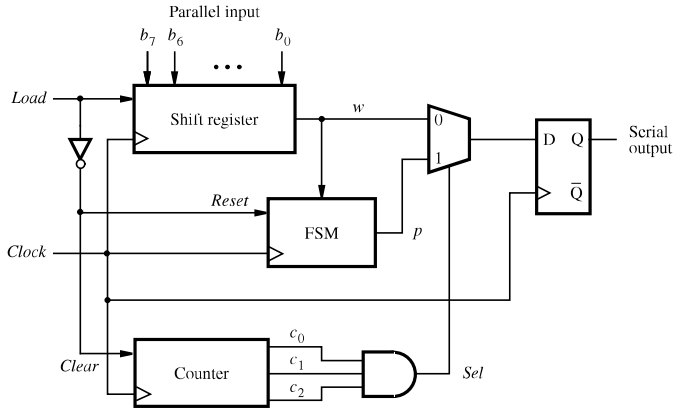


Figure 6.97 Parallel-to-serial converter.

used to generate the parity bit, which is included in the output stream by using a multiplexer. A three-bit counter is used to determine when the p bit is transmitted, which happens when the count reaches 7. Design the desired FSM.

Solution: As the bits are shifted out of the shift register, the FSM examines the bits and keeps track of whether there has been an even or odd number of 1s. It sets p to 1 if there is odd parity. Hence, the FSM must have two states. Figure 6.98 presents the state table, the state-assigned table, and the resulting circuit. The next state expression is

$$Y = \overline{w}y + w\overline{y}$$

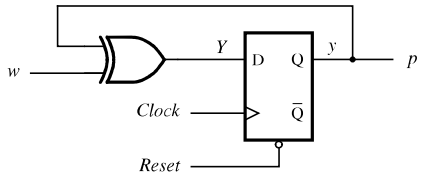
The output p is just equal to y.

Present state	Next state		Output
	$w = 0$	$w = 1$	p
S_{even}	S_{even}	S_{odd}	0
S_{odd}	S_{odd}	S_{even}	1

(a) State table

Present state	Next state		Output
	$w = 0$	$w = 1$	p
y	Y	Y	
0	0	1	0
1	1	0	1

(b) State-assigned table



(c) Circuit

Figure 6.98 FSM for parity generation.

PROBLEMS

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

***6.1** An FSM is defined by the state-assigned table in Figure P6.1. Derive a circuit that realizes this FSM using D flip-flops.

***6.2** Derive a circuit that realizes the FSM defined by the state-assigned table in Figure P6.1 using JK flip-flops.

Present state y_2y_1	Next state $w = 0$ Y_2Y_1	Next state $w = 1$ Y_2Y_1	Output z
0 0	1 0	1 1	0
0 1	0 1	0 0	0
1 0	1 1	0 0	0
1 1	1 0	0 1	1

Figure P6.1 State-assigned table for Problems 6.1 and 6.2.

6.3 Derive the state diagram for an FSM that has an input w and an output z. The machine has to generate $z = 1$ when the previous four values of w were 1001 or 1111; otherwise, $z = 0$. Overlapping input patterns are allowed. An example of the desired behavior is

$$w : 010111100110011111$$
$$z : 0000001001000010011$$

6.4 Write Verilog code for the FSM described in Problem 6.3.

***6.5** Derive a minimal state table for a single-input and single-output Moore-type FSM that produces an output of 1 if in the input sequence it detects either 110 or 101 patterns. Overlapping sequences should be detected.

***6.6** Repeat Problem 6.5 for a Mealy-type FSM.

6.7 Derive the circuits that implement the state tables in Figures 6.51 and 6.52. What is the effect of state minimization on the cost of implementation?

6.8 Derive the circuits that implement the state tables in Figures 6.55 and 6.56. Compare the costs of these circuits.

6.9 A sequential circuit has two inputs, w_1 and w_2, and an output, z. Its function is to compare the input sequences on the two inputs. If $w_1 = w_2$ during any four consecutive clock cycles, the circuit produces $z = 1$; otherwise, $z = 0$. For example

$$w_1 : 0110111000110$$
$$w_2 : 1110101000111$$
$$z : 0000100001110$$

Derive a suitable circuit.

6.10 Write Verilog code for the FSM described in Problem 6.9.

6.11 A given FSM has an input, w, and an output, z. During four consecutive clock pulses, a sequence of four values of the w signal is applied. Derive a state table for the FSM

that produces $z = 1$ when it detects that either the sequence $w : 0010$ or $w : 1110$ has been applied; otherwise, $z = 0$. After the fourth clock pulse, the machine has to be again in the reset state, ready for the next sequence. Minimize the number of states needed.

***6.12** Derive a minimal state table for an FSM that acts as a three-bit parity generator. For every three bits that are observed on the input w during three consecutive clock cycles, the FSM generates the parity bit $p = 1$ if and only if the number of 1s in the three-bit sequence is odd.

6.13 Write Verilog code for the FSM described in Problem 6.12.

6.14 Draw timing diagrams for the circuits in Figures 6.43 and 6.47, assuming the same changes in a and b signals for both circuits. Account for propagation delays.

***6.15** Show a state table for the state-assigned table in Figure P6.1, using A, B, C, D for the four rows in the table. Give a new state-assigned table using a one-hot encoding. For A use the code $y_4y_3y_2y_1 = 0001$. For states B, C, D use the codes 0010, 0100, and 1000, respectively. Synthesize a circuit using D flip-flops.

6.16 Show how the circuit derived in Problem 6.15 can be modified such that the code $y_4y_3y_2y_1 = 0000$ is used for the reset state, A, and the other codes for state B, C, D are changed as needed. (*Hint:* You do not have to resynthesize the circuit!)

***6.17** In Figure 6.59 assume that the unspecified outputs in states B and G are 0 and 1, respectively. Derive the minimized state table for this FSM.

6.18 In Figure 6.59 assume that the unspecified outputs in states B and G are 1 and 0, respectively. Derive the minimized state table for this FSM.

6.19 Derive circuits that implement the FSMs defined in Figures 6.57 and 6.58. Can you draw any conclusions about the complexity of circuits that implement Moore and Mealy types of machines?

6.20 Design a counter that counts pulses on line w and displays the count in the sequence 0, 2, 1, 3, 0, 2, ... Use D flip-flops in your circuit.

***6.21** Repeat Problem 6.20 using JK flip-flops.

***6.22** Repeat Problem 6.20 using T flip-flops.

6.23 Design a modulo-6 counter, which counts in the sequence 0, 1, 2, 3, 4, 5, 0, 1, ... The counter counts the clock pulses if its enable input, w, is equal to 1. Use D flip-flops in your circuit.

6.24 Repeat Problem 6.23 using JK flip-flops.

6.25 Repeat Problem 6.23 using T flip-flops.

6.26 Design a three-bit counterlike circuit controlled by the input w. If $w = 1$, then the counter adds 2 to its contents, wrapping around if the count reaches 8 or 9. Thus if the present state is 8 or 9, then the next state becomes 0 or 1, respectively. If $w = 0$, then the counter subtracts 1 from its contents, acting as a normal down-counter. Use D flip-flops in your circuit.

6.27 Repeat Problem 6.26 using JK flip-flops.

6.28 Repeat Problem 6.26 using T flip-flops.

***6.29** Derive the state table for the circuit in Figure P6.2. What sequence of input values on wire w is detected by this circuit?

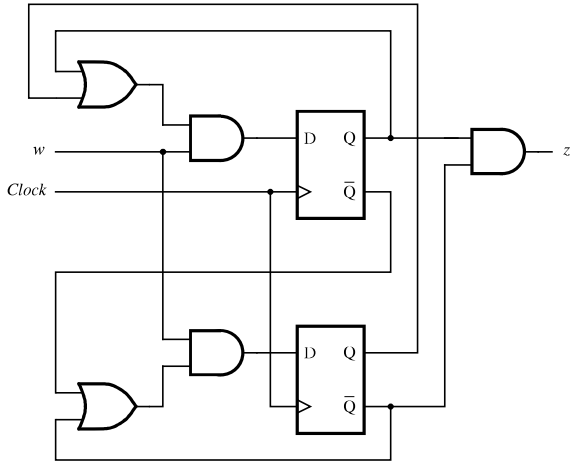


Figure P6.2 Circuit for Problem 6.29.

6.30 Write Verilog code for the FSM shown in Figure 6.57, using the style of code in Figure 6.29.

6.31 Repeat Problem 6.30, using the style of code in Figure 6.34.

6.32 Write Verilog code for the FSM shown in Figure 6.58, using the style of code in Figure 6.29.

6.33 Repeat Problem 6.32, using the style of code in Figure 6.34.

6.34 Write Verilog code for the FSM shown in Figure P6.1.

6.35 Represent the FSM in Figure 6.57 in form of an ASM chart.

6.36 Represent the FSM in Figure 6.58 in form of an ASM chart.

6.37 The arbiter FSM defined in Section 6.8 (Figure 6.72) may cause device 3 to never get serviced if devices 1 and 2 continuously keep raising requests, so that in the Idle state it always happens that either device 1 or device 2 has an outstanding request. Modify the proposed FSM to ensure that device 3 will get serviced, such that if it raises a request, the devices 1 and 2 will be serviced only once before the device 3 is granted its request.

6.38 Write Verilog code for the FSM designed in Problem 6.37.

6.39 Write Verilog code to specify the circuit in Figure 6.97.

6.40 Section 6.5 presents a design for the serial adder. Derive a similar circuit that functions as a serial subtractor which produces the difference of operands A and B. (*Hint:* Use the rule for finding 2's complements in Section 3.3.1 to generate the 2's complement of B.)

6.41 Write Verilog code that defines the serial subtractor designed in Problem 6.40.

6.42 In Section 6.2 we stated that trying all possible state assignments in order to find the best one is impractical. Determine the number of possible state assignments for an FSM that has n states and for which $k = log_2 n$ state variables are used.

REFERENCES

1. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
2. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
3. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
4. M. M. Mano, *Digital Design*, 3rd ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
5. A. Dewey, *Analysis and Design of Digital Systems with VHDL*, (PWS Publishing Co.: 1997).
6. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
7. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
8. 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).
9. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., (Wiley: New York, 1993).
10. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, MA, 1993).
11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. T. L. Booth, *Digital Networks and Computer Systems*, (Wiley: New York, 1971).
13. Z. Kohavi, *Switching and Finite Automata Theory*, (McGraw-Hill: New York, 1970).
14. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, (Prentice-Hall: Englewood Cliffs, NJ, 1966).

chapter

7

DIGITAL SYSTEM DESIGN

CHAPTER OBJECTIVES

In this chapter you will learn about aspects of digital system design, including

- Bus structure
- A simple processor
- Usage of ASM charts
- Clock synchronization and timing issues
- Flip-flop timing at the chip level