Chapter 9

Asynchronous Sequential Logic

9.1 INTRODUCTION

A sequential circuit is specified by a time sequence of inputs, outputs, and internal states. In synchronous sequential circuits, the change of internal state occurs in response to the synchronized clock pulses. Asynchronous sequential circuits do not use clock pulses. The change of internal state occurs when there is a change in the input variables. The memory elements in synchronous sequential circuits are clocked flip-flops. The memory elements in asynchronous sequential circuits are either unclocked flip-flops or time-delay elements. The memory capability of a time-delay device depends on the finite amount of time it takes for the signal to propagate through digital gates. An asynchronous sequential circuit quite often resembles a combinational circuit with feedback.

The design of asynchronous sequential circuits is more difficult than that of synchronous circuits because of the timing problems involved in the feedback path. In a properly designed synchronous system, timing problems are eliminated by triggering all flip-flops with the pulse edge. The change from one state to the next occurs during the short time of the pulse transition. Since the asynchronous circuit does not use a clock, the state of the system is allowed to change immediately after the input changes. Care must be taken to ensure that each new state keeps the circuit in a stable condition even though a feedback path exists.

Asynchronous sequential circuits are useful in a variety of applications. They are used when speed of operation is important, especially in those cases where the digital system must respond quickly without having to wait for a clock pulse. They are more economical to use in small independent systems that require only a few components, as it may not be practical to go to the expense of providing a circuit for generating clock pulses. Asynchronous circuits are useful in applications where the input signals to the system may change at any time, independently of an internal clock. The communication between two units, each having its own independent clock, must be done with asynchronous circuits. Digital designers often produce a mixed system in which some part of the synchronous system has the characteristics of an asynchronous circuit. Knowledge of asynchronous sequential logic behavior is helpful in verifying that the total digital system is operating in the proper manner.

Figure 9.1 shows the block diagram of an asynchronous sequential circuit that consists of a combinational circuit and delay elements connected to form feedback loops. There are n input variables, m output variables, and k internal states. The delay elements can be visualized as providing short-term memory for the sequential circuit. In a gate-type circuit, the propagation delay that exists in the combinational circuit path from input to output provides sufficient delay along the feedback loop so that no specific delay elements are actually inserted into the feedback path. The present-state and next-state variables in asynchronous sequential circuits are customarily called *secondary variables* and *excitation variables*, respectively. The excitation variables should not be confused with the excitable table used in the design of clocked sequential circuits.

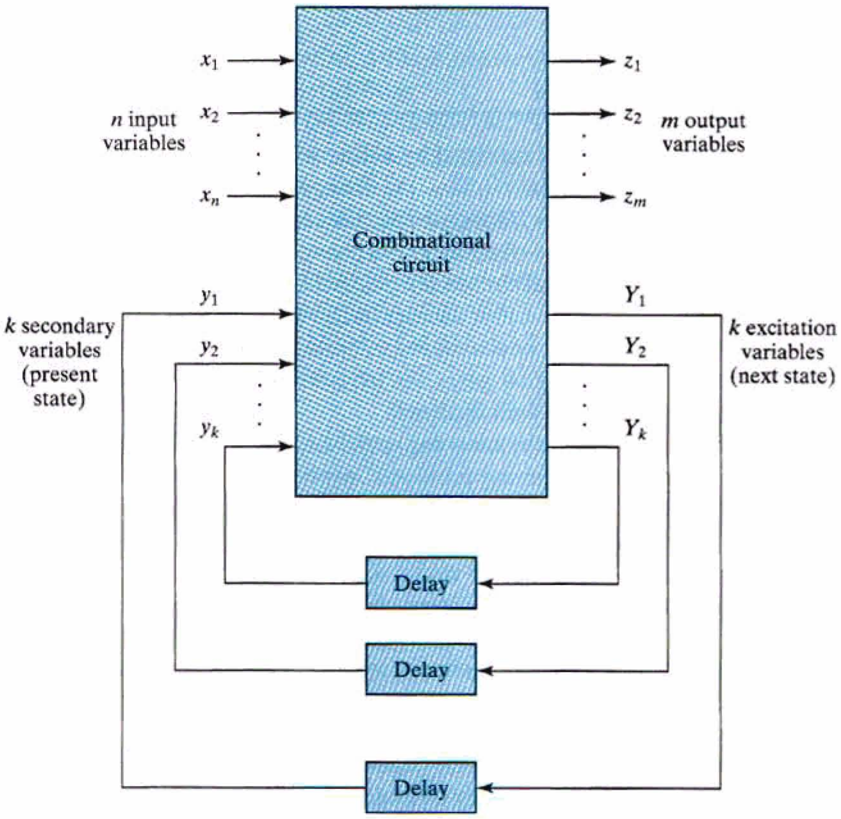


FIGURE 9.1
Block diagram of an asynchronous sequential circuit

When an input variable changes in value, the y secondary variables do not change instantaneously. It takes a certain amount of time for the signal to propagate from the input terminals, through the combinational circuit, to the Y excitation variables, which generate new values for the next state. These values propagate through the delay elements and become the new present state for the secondary variables. Note the distinction between the y's and the Y's. In the steady-state condition, they are the same, but during transition they are not. For a given value of input variables, the system is stable if the circuit reaches a steady-state condition with $y_i = Y_i$ for $i = 1, 2, \ldots, k$. Otherwise, the circuit is in a continuous transition and is said to be unstable. It is important to realize that a transition from one stable state to another occurs only in response to a change in an input variable. This is in contrast to synchronous systems, in which state transitions occur in response to the application of a clock pulse.

To ensure proper operation, asynchronous sequential circuits must be allowed to attain a stable state before the input is changed to a new value. Because of delays in the wires and the gate circuits, it is impossible to have two or more input variables change at exactly the same instant of time without an uncertainty as to which one changes first. Therefore, simultaneous changes of two or more variables are usually prohibited. This restriction means that only one input variable can change at any one time and the time between two input changes must be longer than the time it takes the circuit to reach a stable state. Such operation, defined as *fundamental mode*, assumes that the input signals change one at a time and only when the circuit is in a stable condition.

9.2 ANALYSIS PROCEDURE

The analysis of asynchronous sequential circuits consists of obtaining a table or a diagram that describes the sequence of internal states and outputs as a function of changes in the input variables. A logic diagram manifests the behavior of an asynchronous sequential circuit if it has one or more feedback loops or if it includes unclocked flip-flops. In this section, we will investigate the behavior of asynchronous sequential circuits that have feedback paths without employing flip-flops. Unclocked flip-flops are called *latches*, and their use in asynchronous sequential circuits will be explained in the next section.

The analysis procedure will be presented by means of three specific examples. The first example introduces the transition table, the second defines the flow table, and the third investigates the stability of asynchronous sequential circuits.

Transition Table

An example of an asynchronous sequential circuit with only gates is shown in Fig. 9.2. The diagram clearly shows two feedback loops from the OR gate outputs back to the AND gate inputs. The circuit consists of one input variable x and two internal states. The internal states have two excitation variables, Y_1 and Y_2, and two secondary variables, y_1 and y_2. The delay associated with each feedback loop is obtained from the propagation delay between each y input and its corresponding Y output. Each logic gate in the path introduces a propagation delay of about 2 to 10 ns. The wires that conduct electrical signals introduce approximately a 1-ns delay for each foot of wire. Thus, no additional external delay elements are necessary when the combinational circuit and the wires in the feedback path provide sufficient delay.

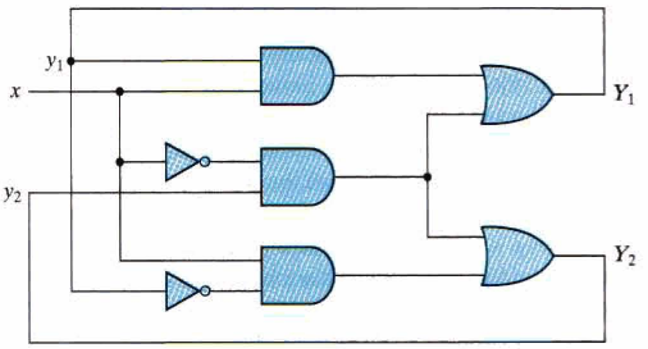


FIGURE 9.2
Example of an asynchronous sequential circuit

The analysis of the circuit starts with a consideration of the excitation variables as outputs and the secondary variables as inputs. We then derive the Boolean expressions for the excitation variables as a function of the input and secondary variables. These expressions, readily obtained from the logic diagram, are

$$Y_1 = xy_1 + x'y_2$$
$$Y_2 = xy_1' + x'y_2$$

The next step is to plot the Y_1 and Y_2 functions in a map, as shown in Fig. 9.3(a) and (b). The encoded binary values of the y variables are used for labeling the rows, and the input x variable is used to designate the columns. This configuration results in a slightly different three-variable map from the one used in previous chapters. However, it is still a valid map, and such

y_1y_2 \ x	0	1
00	0	0
01	1	0
11	1	1
10	0	1

(a) Map for $Y_1 = xy_1 + x'y_2$

y_1y_2 \ x	0	1
00	0	1
01	1	1
11	1	0
10	0	0

(b) Map for $Y_2 = xy'_1 + x'y_2$

y_1y_2 \ x	0	1
00	00	01
01	11	01
11	11	10
10	00	10

(c) Transition table

FIGURE 9.3
Maps and transition table for the circuit of Fig. 9.2

a configuration is more convenient in dealing with asynchronous sequential circuits. Note that, unlike what was done in previous chapters, the variables belonging to the appropriate squares are not marked along the sides of the map.

The transition table shown in Fig. 9.3(c) is obtained from the maps by combining the binary values in corresponding squares. The transition table shows the value of $Y = Y_1Y_2$ inside each square. The first bit of Y is obtained from the value of Y_1, and the second bit is obtained from the value of Y_2 in the same square position. For a state to be stable, the secondary variables must match the excitation variables (i.e., the value of Y must be the same as that of $y = y_1y_2$). Those entries in the transition table where $Y = y$ are circled to indicate a stable condition. An uncircled entry represents an unstable state.

Now consider the effect of a change in the input variable. The square for $x = 0$ and $y = 00$ in the transition table shows that $Y = 00$. Since Y represents the next value of y, this is a stable condition. If x changes from 0 to 1 while $y = 00$, the circuit changes the value of Y to 01. This represents a temporary unstable condition, because Y is not equal to the present value of y. What happens next is that as soon as the signal propagates to make $Y = 01$, the feedback path in the circuit causes a change in y to 01. This change is manifested in the transition table by a transition from the first row ($y = 00$) to the second row, where $y = 01$. Now that $y = Y$, the circuit reaches a stable condition with an input of $x = 1$. In general, if a change in the input takes the circuit to an unstable state, the value of y will change (while that of x remains the same) until it reaches a stable (circled) state. Using this type of analysis for the remaining squares of the transition table, we find that the circuit repeats the sequence of states 00, 01, 11, 10 when the input repeatedly alternates between 0 and 1.

Note the difference between a synchronous and an asynchronous sequential circuit. In a synchronous system, the present state is totally specified by the flip-flop values and does not change if the input changes while the clock pulse is inactive. In an asynchronous circuit, the internal state can change immediately after a change in the input. Because of this rapid change, it is sometimes convenient to combine the internal state with the input value together and call it the *total state* of the circuit. The circuit whose transition table is shown in Fig. 9.3(c) has four stable total states—$y_1y_2x = 000, 011, 110,$ and 101—and four unstable total states—001, 010, 111, and 100.

The transition table of asynchronous sequential circuits is similar to the state table used for synchronous circuits. If we regard the secondary variables as the present state and the excitation variables as the next state, we obtain the state table shown in Table 9.1. This table provides the same information as the transition table. There is one restriction that applies to the

Table 9.1
State Table for the Circuit of Fig. 9.2

Present State		Next State			
		$x = 0$		$x = 1$	
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	1	1	0

asynchronous case, but not the synchronous case: In the asynchronous transition table, there usually is at least one next-state entry that is the same as the present-state value in each row. Otherwise, all the total states in that row will be unstable.

The procedure for obtaining a transition table from the circuit diagram of an asynchronous sequential circuit is as follows:

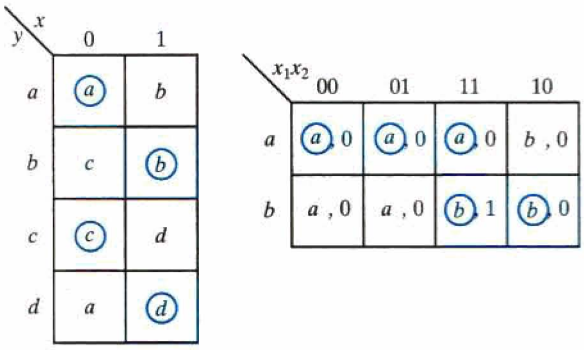
1. Determine all feedback loops in the circuit.
2. Designate the output of each feedback loop with variable Y_i and its corresponding input with y_i for $i = 1, 2, \ldots, k$, where k is the number of feedback loops in the circuit.
3. Derive the Boolean functions of all Y's as a function of the external inputs and the y's.
4. Plot each Y function in a map, using the y variables for the rows and the external inputs for the columns.
5. Combine all the maps into one table showing the value of $Y = Y_1Y_2 \cdots Y_k$ inside each square.
6. Circle those values of Y in each square that are equal to the value of $y = y_1y_2 \cdots y_k$ in the same row.

Once the transition table is available, the behavior of the circuit can be analyzed by observing the state transition as a function of changes in the input variables.

Flow Table

During the design of asynchronous sequential circuits, it is more convenient to name the states by letter symbols without making specific reference to their binary values. Such a table is called a *flow table* and is similar to a transition table, except that the internal states are symbolized with letters rather than binary numbers. The flow table also includes the output values of the circuit for each stable state.

Examples of flow tables are shown in Fig. 9.4. The one in Fig. 9.4(a) has four states, designated by the letters a, b, c, and d. It reduces to the transition table of Fig. 9.3(c) if we assign



$y \backslash x$	0	1
a	$\textcircled{a}$	b
b	c	$\textcircled{b}$
c	$\textcircled{c}$	d
d	a	$\textcircled{d}$

(a) Four states with one input

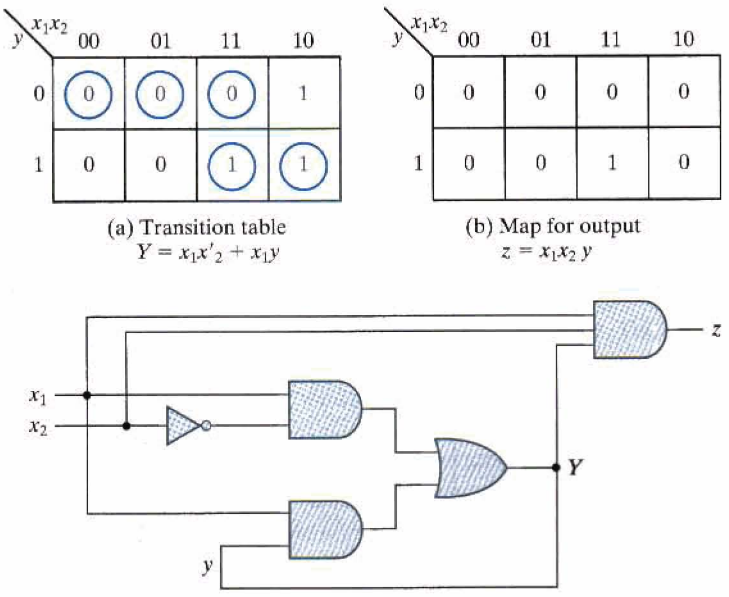
x_1x_2	00	01	11	10
a	$\textcircled{a}, 0$	$\textcircled{a}, 0$	$\textcircled{a}, 0$	$b, 0$
b	$a, 0$	$a, 0$	$\textcircled{b}, 1$	$\textcircled{b}, 0$

(b) Two states with two inputs and one output

FIGURE 9.4
Examples of flow tables

the following binary values to the states: $a = 00$, $b = 01$, $c = 11$, and $d = 10$. The table of Fig. 9.4(a) is called a *primitive* flow table because it has only one stable state in each row. Figure 9.4(b) shows a flow table with more than one stable state in the same row. It has two states, a and b; two inputs, x_1 and x_2; and one output, z. The binary value of the output variable is indicated inside the square next to the state symbol and is separated from the state symbol by a comma. From the flow table, we observe the following behavior of the circuit: If $x_1 = 0$, the circuit is in state a. If x_1 goes to 1 while x_2 is 0, the circuit goes to state b. With inputs $x_1x_2 = 11$, the circuit may be either in state a or in state b. If it is in state a, the output is 0, and if it is in state b, the output is 1. State b is maintained if the inputs change from 10 to 11. The circuit stays in state a if the inputs change from 01 to 11. Remember that in fundamental mode two input variables cannot change simultaneously; therefore, we do not allow a change of inputs from 00 to 11.

In order to obtain the circuit described by a flow table, it is necessary to assign a distinct binary value to each state. Such an assignment converts the flow table into a transition table from which we can derive the logic diagram. This is illustrated in Fig. 9.5 for the flow table of Fig. 9.4(b). We assign binary 0 to state a and binary 1 to state b. The result is the transition table of Fig. 9.5(a). The output map shown in Fig. 9.5(b) is obtained directly from the output values in the flow table. The excitation function Y and the output function z are simplified by means of the two maps. The logic diagram of the circuit is shown in Fig. 9.5(c).



$y \backslash x_1x_2$	00	01	11	10
0	⓪	⓪	⓪	1
1	0	0	①	①

(a) Transition table
$Y = x_1x'_2 + x_1y$

$y \backslash x_1x_2$	00	01	11	10
0	0	0	0	0
1	0	0	1	0

(b) Map for output
$z = x_1x_2\,y$

(c) Logic diagram

FIGURE 9.5
Derivation of a circuit specified by the flow table of Fig. 9.4(b)

This example demonstrates the procedure for obtaining the logic diagram from a given flow table. Doing that, however, is not always so simple. There are several difficulties associated with the binary state assignment and with the output assigned to the unstable states. These problems are discussed in detail next.

Race Conditions

A *race* condition is said to exist in an asynchronous sequential circuit when two or more binary state variables change value in response to a change in an input variable. When unequal delays are encountered, a race condition may cause the state variables to change in an unpredictable manner. For example, if the state variables must change from 00 to 11, the difference in delays may cause the first variable to change sooner than the second, with the result that the state variables change in sequence from 00 to 10 and then to 11. If the second variable changes sooner than the first, the state variables will change from 00 to 01 and then to 11. Thus, the order by which the state variables change may not be known in advance. If the final stable state that the circuit reaches does not depend on the order in which the state variables change, the race is called a *noncritical* race. If it is possible to end up in two or more different stable states, depending on the order in which the state variables change, then the race is a *critical* race. For proper operation, critical races must be avoided.

The two examples in Fig. 9.6 illustrate noncritical races. We start with the total stable state $y_1y_2x = 000$ and change the input from 0 to 1. The state variables must then change from 00 to 11, which defines a race condition. The transitions listed under each table show three possible ways that the state variables may change. Either they can change simultaneously from 00 to 11, or they may change in sequence from 00 to 01 and then to 11, or they may change in sequence from 00 to 10 and then to 11. In all cases, the final stable state is the same, so the race is noncritical. In (a), the final total state is $y_1y_2x = 111$, and in (b), it is 011.

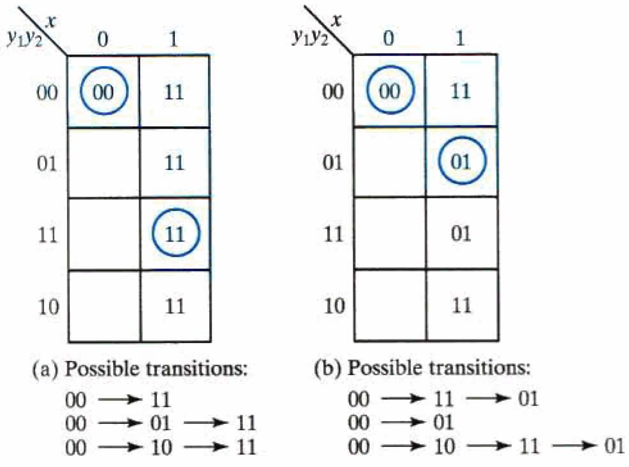
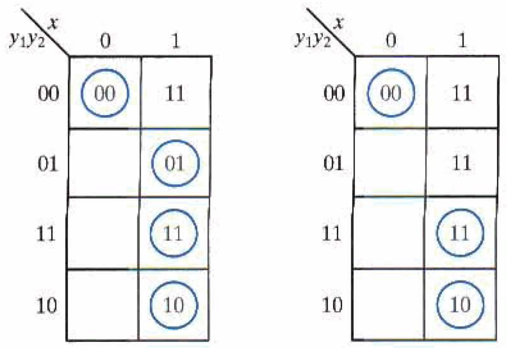


FIGURE 9.6
Examples of noncritical races



(a) Possible transitions:

$00 \rightarrow 11$
$00 \rightarrow 01$
$00 \rightarrow 10$

(b) Possible transitions:

$00 \rightarrow 11$
$00 \rightarrow 01 \rightarrow 11$
$00 \rightarrow 10$

FIGURE 9.7
Examples of critical races

The transition tables of Fig. 9.7 illustrate critical races. Here again, we start with the total stable state $y_1y_2x = 000$ and change the input from 0 to 1. The state variables must then change from 00 to 11. If they change simultaneously, the final total stable state is 111. In the transition table of part (a), if, because of unequal propagation delay, Y_2 changes to 1 before Y_1 does, then the circuit goes to the total stable state 011 and remains there. If, however, Y_1 changes first, the internal state becomes 10 and the circuit will remain in the stable total state 101. Hence, the race is critical because the circuit goes to different stable states, depending on the order in which the state variables change. The transition table of Fig. 9.7(b) illustrates another critical race, in which two possible transitions result in one final total state, but the third possible transition goes to a different total state.

Races may be avoided by making a proper binary assignment to the state variables. The state variables must be assigned binary numbers in such a way that only one state variable can change at any one time when a state transition occurs in the flow table. The subject of race-free state assignment is discussed in Section 9.6.

Races can be avoided by directing the circuit through intermediate unstable states with a unique state-variable change. When a circuit goes through a unique sequence of unstable states, it is said to have a *cycle*. Fig. 9.8 illustrates the occurrence of cycles. Again, we start with $y_1y_2 = 00$ and change the input from 0 to 1. The transition table of part (a) gives a *unique* sequence that terminates in a total stable state 101. The table in (b) shows that even though the state variables change from 00 to 11, the cycle provides a unique transition from 00 to 01 and then to 11. Care must be taken when using a cycle that terminates with a stable state. If a cycle does not terminate with a stable state, the circuit will keep going from one unstable state to another, making the entire circuit unstable. This phenomenon is demonstrated in Fig. 9.8(c) and also in the next example.

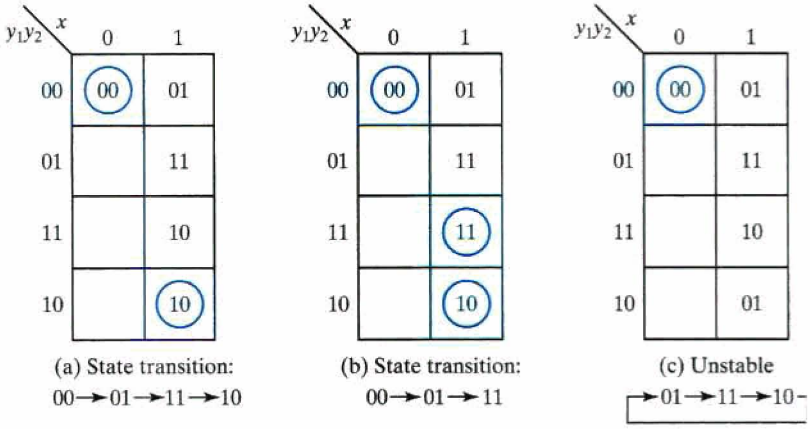

y_1y_2
x
0
1
00
01
11
10
(a) State transition: 00→01→11→10
(b) State transition: 00→01→11
(c) Unstable
→01→11→10

FIGURE 9.8
Examples of cycles

Stability Considerations

Because of the feedback connection that exists in asynchronous sequential circuits, care must be taken to ensure that the circuit does not become unstable. An unstable condition will cause the circuit to oscillate between unstable states. The transition-table method of analysis can be useful in detecting the occurrence of instability.

Consider, for example, the circuit of Fig. 9.9(a). The excitation function is

$$Y = (x_1y)'x_2 = (x_1' + y')x_2 = x_1'x_2 + x_2y'$$

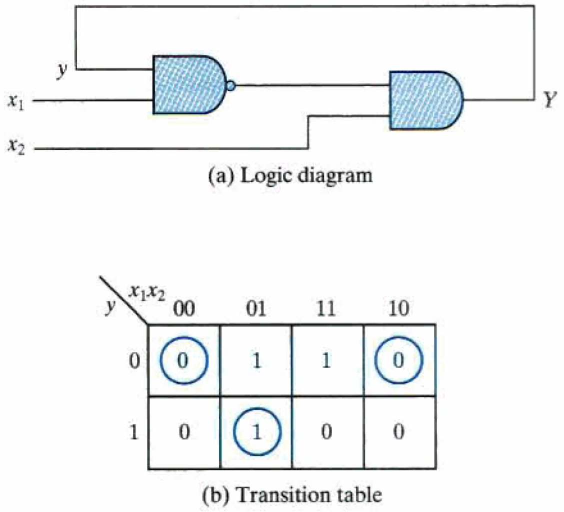


FIGURE 9.9
Example of an unstable circuit

The transition table for the circuit is shown in Fig. 9.9(b). Those values of Y which are equal to y are circled and represent stable states. Uncircled entries indicate unstable conditions. Note that column 11 has no stable states. This means that with input x_1x_2 fixed at 11, the values of Y and y are never the same. If $y = 0$, then $Y = 1$, which causes a transition to the second row of the table, with $y = 1$ and $Y = 0$. This in turn causes a transition back to the first row, with the result that the state variable alternates between 0 and 1 indefinitely, as long as the input is 11.

The instability condition can be detected directly from the logic diagram. Let $x_1 = 1$, $x_2 = 1$, and $y = 1$. Then the output of the NAND gate is equal to 0, and the output of the AND gate is equal to 0, making Y equal to 0, with the result that $Y \neq y$. Now if $y = 0$, the output of the NAND gate is 1 and the output of the AND gate is 1, making Y equal to 1, with the result that $Y \neq y$. If it is assumed that each gate has a propagation delay of 5 ns (including transmission over the wires), we will find that Y will be 0 for 10 ns and 1 for the next 10 ns. This will result in a square-wave waveform with a period of 20 ns. The frequency of oscillation is the reciprocal of the period and is equal to 50 MHz. Unless one is designing a square-wave generator, the instability that may occur in asynchronous sequential circuits is undesirable and must be avoided.

9.3 CIRCUITS WITH LATCHES

Historically, asynchronous circuits were known and used before synchronous circuits were developed. The first practical digital circuits were constructed with relays, which are more adaptable to asynchronous operations. For this reason, the traditional method of asynchronous circuit configuration has been with components that are connected to form one or more feedback loops. When digital circuits are constructed with electronic components, it is convenient to employ the *SR* latch (introduced in Section 5.3) as a memory element. The use of *SR* latches in asynchronous sequential circuits produces an orderly pattern in the logic diagrams, with the memory elements clearly visible. In this section, we analyze the operation of the *SR* latch, using the technique introduced in the previous section. We then show a procedure for implementing asynchronous sequential circuits using *SR* latches.

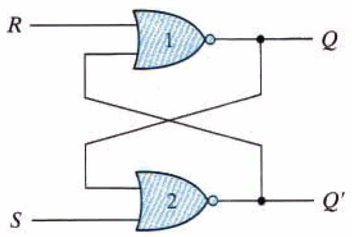
SR Latch

The *SR* latch is a digital circuit with two inputs S and R and two cross-coupled NOR gates or two cross-coupled NAND gates. The cross-coupled NOR gate circuit is shown in Fig. 9.10. This circuit and its truth table are taken from Fig. 5.3. In order to analyze the circuit by the transition-table method, it is first redrawn in Fig. 9.10(c) to see the feedback path from the output of gate 1 to the input of gate 2. The output Q is equivalent to the excitation variable Y and the secondary variable y. The Boolean function for the output is

$$Y = [(S + y)' + R]' = (S + y)R' = SR' + R'y$$

Plotting Y as in Fig. 9.10(d), we obtain the transition table for the circuit.

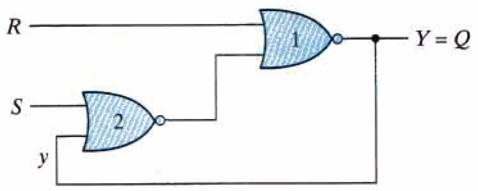
We can now investigate the behavior of the *SR* latch from the transition table. The state with $SR = 10$ is a stable state because $Y = y = 1$; likewise, the state with $SR = 01$ is a stable state, because $Y = y = 0$. With $SR = 10$, the output $Q = Y = 1$ and the latch is said



(a) Cross-coupled circuit

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(After SR = 10)
0	1	0	1	
0	0	0	1	(After SR = 01)
1	1	0	0	

(b) Truth table



(c) Circuit showing feedback

y \ SR	00	01	11	10
0	⓪	⓪	⓪	1
1	①	0	0	①

$Y = SR' + R'y$
$Y = S + R'y$ when $SR = 0$

(d) Transition table

FIGURE 9.10
***SR* latch with NOR gates**

to be set. Changing S to 0 leaves the circuit in the set state. With $SR = 01$, the output $Q = Y = 0$ and the latch is said to be reset. A change of R back to 0 leaves the circuit in the reset state. These conditions are also listed in the truth table. The circuit exhibits some difficulty when both S and R are equal to 1. From the truth table, we see that both Q and Q' are equal to 0, a condition that violates the requirement that these two outputs be the complement of each other. Moreover, from the transition table, we note that going from $SR = 11$ to $SR = 00$ produces an unpredictable result. If S goes to 0 first, the output remains at 0, but if R goes to 0 first, the output goes to 1. In normal operation, we must make sure that 1's are not applied to both the S and R inputs simultaneously. This condition can be expressed by the Boolean function $SR = 0$, which states that the ANDing of S and R must always result in a 0.

Coming back to the excitation function, we note that when we OR the Boolean expression SR' with SR, the result is the single variable S:

$$SR' + SR = S(R' + R) = S$$

From this, we infer that $SR' = S$ when $SR = 0$. Therefore, the excitation function derived previously, namely,

$$Y = SR' + R'y$$

can be expressed in Fig. 9.10(d) as the *reduced excitation function*

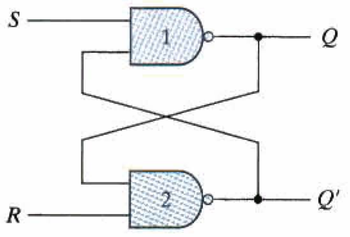
$$Y = S + R'y \quad \text{when } SR = 0$$

To analyze a circuit with an *SR* latch, we must first check that the Boolean condition $SR = 0$ holds at all times. We then use the reduced excitation function to analyze the circuit. However, if it is found that both *S* and *R* can be equal to 1 at the same time, then it is necessary to use the original excitation function.

The analysis of the *SR* latch with NAND gates is carried out in Fig. 9.11. The NAND latch operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0 to *R* causes the output *Q* to go to 0, thus putting the latch in the reset state. After the *R* input returns to 1, a change of *S* to 0 causes a change to the set state. The condition to be avoided here is that both *S* and *R* not be 0 simultaneously. This condition is satisfied when $S'R' = 0$. The excitation function for the circuit in Fig. 9.11(c) is

$$Y = [S(Ry)']' = S' + Ry$$

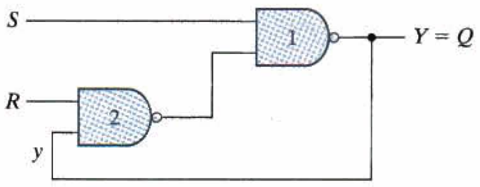
Comparing this with the excitation function of the NOR latch, we note that *S* has been replaced with S' and R' with *R*. Hence, the input variables for the NAND latch require the complemented values of those used in the NOR latch. For this reason, the NAND latch is sometimes referred to as an $S'R'$ latch (or $\overline{S}$–$\overline{R}$ latch).



(a) Cross-coupled circuit

S	*R*	*Q*	*Q'*	
1	0	0	1	
1	1	0	1	(After *SR* = 10)
0	1	1	0	
1	1	1	0	(After *SR* = 01)
0	0	1	1	

(b) Truth table



(c) Circuit showing feedback

y \ *SR*	00	01	11	10
0	1	1	$\textcircled{0}$	$\textcircled{0}$
1	$\textcircled{1}$	$\textcircled{1}$	$\textcircled{1}$	0

(d) Transition table

FIGURE 9.11
***SR* latch with NAND gates**

Analysis Example

Asynchronous sequential circuits can be constructed with the use of *SR* latches with or without external feedback paths. Of course, there is always a feedback loop within the latch itself. The analysis of a circuit with latches will be demonstrated by means of a specific example from which it will be possible to generalize the procedural steps necessary to analyze other, similar circuits.

The circuit shown in Fig. 9.12 has two *SR* latches with outputs Y_1 and Y_2. There are two inputs, x_1 and x_2, and two external feedback loops giving rise to the secondary variables, y_1 and y_2. Note that this circuit resembles a conventional sequential circuit with latches behaving like flip-flops without clock pulses. The analysis of the circuit requires that we first obtain the Boolean functions for the *S* and *R* inputs in each latch:

$$\begin{aligned} S_1 &= x_1y_2 \qquad & S_2 &= x_1x_2 \\ R_1 &= x_1'x_2' \qquad & R_2 &= x_2'y_1 \end{aligned}$$

We then check whether the condition $SR = 0$ is satisfied to ensure proper operation of the circuit:

$$\begin{aligned} S_1R_1 &= x_1y_2x_1'x_2' = 0 \\ S_2R_2 &= x_1x_2x_2'y_1 = 0 \end{aligned}$$

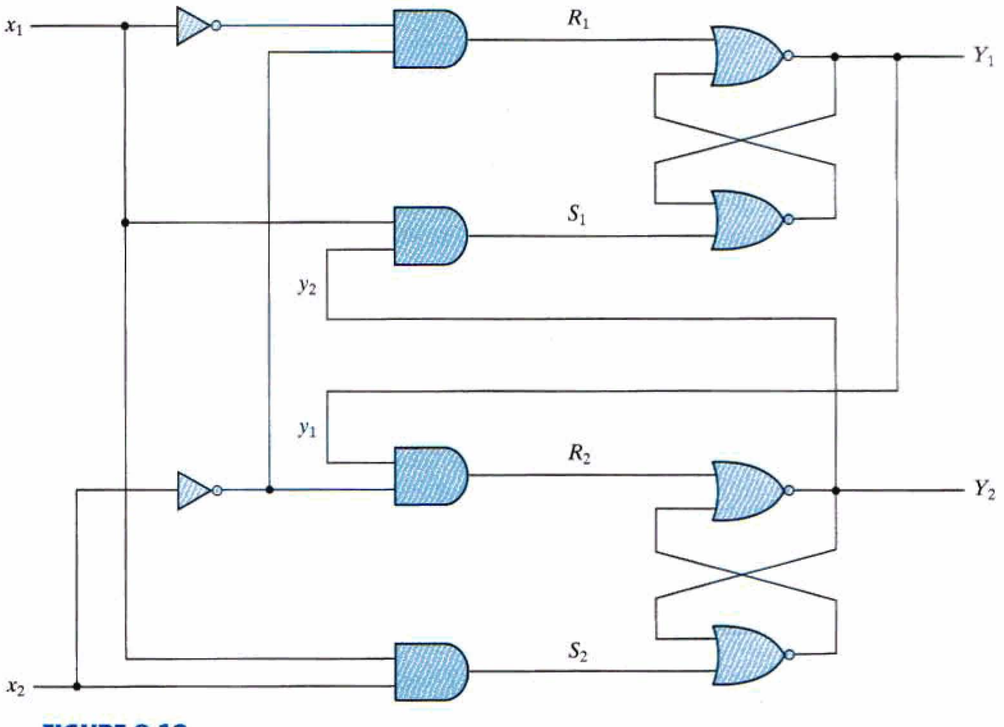


FIGURE 9.12
Example of a circuit with *SR* latches

$y_1y_2 \backslash x_1x_2$	00	01	11	10
00	$\textcircled{00}$	$\textcircled{00}$	01	$\textcircled{00}$
01	$\textcircled{01}$	$\textcircled{01}$	11	11
11	00	$\textcircled{11}$	$\textcircled{11}$	10
10	00	$\textcircled{10}$	11	$\textcircled{10}$

FIGURE 9.13
Transition table for the circuit of Fig. 9.12

The result is 0 because $x_1x_1' = x_2x_2' = 0$.

The next step is to derive the transition table of the circuit. Remember that the transition table specifies the value of Y as a function of y and x. The excitation functions are derived from the relation $Y = S + R'y$ (see Figure 9.11(d)) and are

$$Y_1 = S_1 + R_1'y_1 = x_1y_2 + (x_1 + x_2)y_1 = x_1y_2 + x_1y_1 + x_2y_1$$
$$Y_2 = S_2 + R_2'y_2 = x_1x_2 + (x_2 + y_1')y_2 = x_1x_2 + x_2y_2 + y_1'y_2$$

We now develop a composite map for $Y = Y_1Y_2$. The y variables are assigned to the rows in the map, and the x variables are assigned to the columns, as shown in Fig. 9.13. The Boolean functions Y_1 and Y_2, as just expressed, are used to plot the composite map for Y. The entries of Y in each row that have the same value as that given to y are circled and represent stable states. Investigating the transition table, we deduce that the circuit is stable. There is a critical race condition when the circuit is initially in total state $y_1y_2x_1x_2 = 1101 (Y_1Y_2 = 11)$ and x_2 changes from 1 to 0 $(Y_1Y_2 = 00)$. If Y_1 changes to 0 before Y_2, the circuit goes to total state 0100 instead of 0000. However, with approximately equal delays in the gates and latches, this undesirable situation is not likely to occur.

The procedure for analyzing an asynchronous sequential circuit with *SR* latches can be summarized as follows:

1. Label each latch output with Y_i and its external feedback path (if any) with y_i for $i = 1, 2, \ldots, k$.
2. Derive the Boolean functions for the S_i and R_i inputs in each latch.
3. Check whether $SR = 0$ for each NOR latch or whether $S'R' = 0$ for each NAND latch. If either of these conditions is not satisfied, there is a possibility that the circuit may not operate properly.
4. Evaluate $Y = S + R'y$ for each NOR latch or $Y = S' + Ry$ for each NAND latch.
5. Construct a map, with the y's representing the rows and the x inputs representing the columns.
6. Plot the value of $Y = Y_1Y_2 \cdots Y_k$ in the map.
7. Circle all stable states such that $Y = y$. The resulting map is then the transition table.

Latch Excitation Table

The transition table of the *SR* latch is useful for analysis and for defining the operation of the latch. It specifies the excitation variable Y when the secondary variable y and the inputs S and R are known. During the implementation process, the transition table of the circuit is available and we wish to find the values of S and R. For this reason, we need a table that lists the required inputs S and R for each of the possible transitions from y to Y. Such a list is called an *excitation table*.

The excitation table of the *SR* latch is shown in Fig. 9.14(b). The first two columns list the four possible transitions from y to Y. The next two columns specify the required input values that will result in the specified transition. For example, in order to provide a transition from $y = 0$ to $Y = 1$, it is necessary to ensure that input $S = 1$ and input $R = 0$. This is shown in the second row of the transition table.

The required input conditions for each of the four transitions in the excitation table can be derived directly from the latch transition table of Fig. 9.10(d) after removing the unstable condition $SR = 11$. For example, the transition table shows that in order to change from $y = 0$ to $Y = 0$, *SR* can be either 00 or 01. This means that S must be 1 and R may be either 0 or 1. Therefore, the first row in the excitation table shows $S = 0$ and $R = \text{X}$, where X is a don't-care condition signifying either a 0 or a 1.

Implementation Example

A sequential circuit with *SR* latches is implemented through a procedure for obtaining the logic diagram from a given transition table. The procedure requires that we determine the Boolean functions for the S and R inputs of each latch. The logic diagram is then obtained by drawing the *SR* latches and the logic gates that implement the S and R functions. To demonstrate the procedure, we will repeat the implementation example of Fig. 9.5. The output circuit remains the same and will not be repeated again.

The transition table from Fig. 9.5(a) is duplicated in Fig. 9.14(a). From the information given in the transition table and from the latch excitation table conditions in Fig. 9.14(b), we can obtain the maps for the S and R inputs of the latch, as shown in Fig. 9.14(c) and (d). For example, the square in the second row and third column ($yx_1x_2 = 111$) in Fig. 9.14(a) requires a transition from $y = 1$ to $Y = 1$. The excitation table specifies $S = \text{X}$, $R = 0$ for this change. Therefore, the corresponding square in the S map is marked with an X and the one in the R map with a 0. All other squares are filled with values in a similar manner. The maps are then used to derive the simplified Boolean functions

$$S = x_1x_2' \qquad \text{and} \qquad R = x_1'$$

The logic diagram consists of an *SR* latch and the gates required to implement the S and R Boolean functions. The circuit is as shown in Fig. 9.14(e) when a NOR latch is used. With a NAND latch, we must use the complemented values for S and R:

$$S = (x_1x'_2)' \qquad \text{and} \qquad R = x_1$$

This circuit is shown in Fig. 9.14(f).

The general procedure for implementing a circuit with *SR* latches from a given transition table can now be summarized as follows:

$y \backslash x_1x_2$	00	01	11	10
0	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	1
1	0	0	$\textcircled{1}$	$\textcircled{1}$

(a) Transition table
$Y = x_1x'_2 + x_1y$

y	Y	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	1

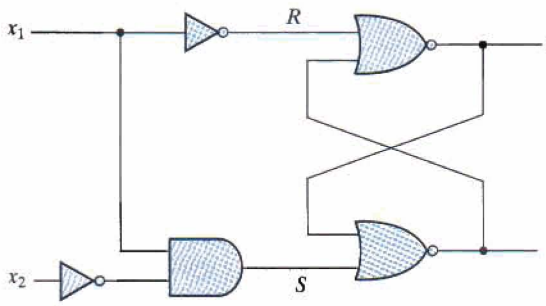
(b) Latch excitation table

$y \backslash x_1x_2$	00	01	11	10
0	0	0	0	1
1	0	0	X	X

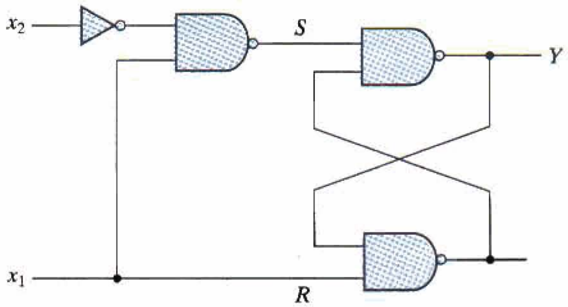
(c) Map for $S = x_1x'_2$

$y \backslash x_1x_2$	00	01	11	10
0	X	X	X	0
1	1	1	0	0

(d) Map for $R = x'_1$



(e) Circuit with NOR latch



(f) Circuit with NAND latch

FIGURE 9.14
Derivation of a latch circuit from a transition table

1. Given a transition table that specifies the excitation function $Y = Y_1Y_2 \cdots Y_k$, derive a pair of maps for S_i and R_i for each $i = 1, 2, \ldots, k$. This is done by using the conditions specified in the latch excitation table of Fig. 9.14(b).
2. Derive the simplified Boolean functions for each S_i and R_i. Care must be taken not to make S_i and R_i equal to 1 in the same minterm square.
3. Draw the logic diagram, using k latches together with the gates required to generate the S and R Boolean functions. For NOR latches, use the S and R Boolean functions obtained in step 2. For NAND latches, use the complemented values of those obtained in step 2.

Another useful example of latch implementation is found in Section 9.7.

Debounce Circuit

Input binary information in a digital system can be generated manually by means of mechanical switches. One position of the switch provides a voltage equivalent to logic 1, and the other position provides a second voltage equivalent to logic 0. Mechanical switches are also used to start, stop, or reset the digital system. In testing digital circuits in the laboratory, the input signals will normally come from switches. A common characteristic of a mechanical switch is that when the arm is thrown from one position to the other, the switch contact vibrates or bounces several times before coming to a final rest. In a typical switch, the contact bounce may take several milliseconds to die out, causing the signal to oscillate between 1 and 0 because the switch contact is vibrating.

A debounce circuit is a circuit which removes the series of pulses that result from a contact bounce and produces a single smooth transition of the binary signal from 0 to 1 or from 1 to 0. One such circuit consists of a single-pole, double-throw switch connected to an *SR* latch, as shown in Fig. 9.15. The center contact is connected to ground that provides a signal equivalent to logic 0. When one of the two contacts, *A* or *B*, is not connected to ground through the switch, it behaves like a logic-1 signal. A resistor is sometimes connected from each contact to a fixed voltage to provide a firm logic-1 signal. When the switch is thrown from position *A* to position *B* and back, the outputs of the latch produce a single pulse as shown, negative for Q and positive for Q'. The switch is usually a push button whose contact rests in position *A*. When the push button is depressed, it goes to position *B*, and when released, it returns to position *A*.

The operation of the debounce circuit is as follows: When the switch rests in position *A*, we have the condition $S = 0$, $R = 1$ and $Q = 1$, $Q' = 0$. (See Fig. 9.11(b).) When the switch is moved to position *B*, the ground connection causes *R* to go to 0, while *S* becomes a 1 because contact *A* is open. This condition in turn causes output Q to go to 0 and Q' to go to 1. After the switch makes an initial contact with *B*, it bounces several times, but for proper operation, we must assume that it does not bounce back far enough to reach point *A*. The output of the latch will be unaffected by the contact bounce because Q' remains 1 (and Q remains 0) whether *R* is equal to 0 (contact with ground) or equal to 1 (no contact with ground). When the switch returns to position *A*, *S* becomes 0 and Q returns to 1. The output again will exhibit a smooth transition, even if there is a contact bounce in position *A*.

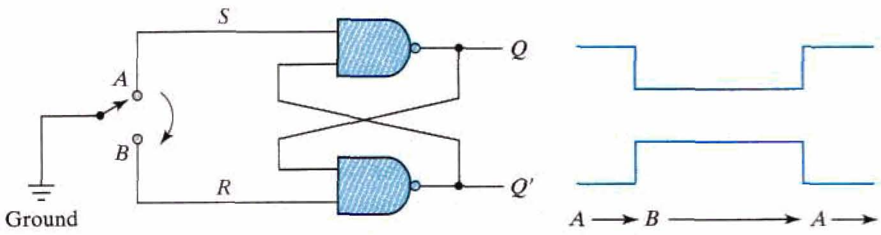


FIGURE 9.15
Debounce circuit

9.4 DESIGN PROCEDURE

The design of an asynchronous sequential circuit starts from the statement of the problem and culminates in a logic diagram. There are a number of design steps that must be carried out in order to minimize the complexity of the circuit and to produce a stable circuit without critical races. Briefly, the design steps are as follows: A primitive flow table is obtained from the design specifications. The flow table is then reduced to a minimum number of states. Next, the states are given a binary assignment from which we obtain the transition table. Finally, from the transition table, we derive the logic diagram as a combinational circuit with feedback or as a circuit with *SR* latches.

The design process will be demonstrated by going through a specific example. Once this example is mastered, it will be easier to understand the design steps that are enumerated at the end of this section. Some of the steps require the application of formal procedures, and these are discussed in greater detail in the sections that follow.

Design Example

It is necessary to design a gated latch circuit with two inputs G (gate) and D (data) and one output Q. Binary information present at the D input is transferred to the Q output when G is equal to 1. The Q output will follow the D input as long as $G = 1$. When G goes to 0, the information that was present at the D input at the time the transition occurred is retained at the Q output. The gated latch is a memory element that accepts the value of D when $G = 1$ and retains this value after G goes to 0. Once $G = 0$, a change in D does not change the value of the output Q.

Primitive Flow Table

As defined previously, a primitive flow table is a flow table with only one stable total state in each row. Remember that a total state consists of the internal state combined with the input. The derivation of the primitive flow table can be facilitated if we first form a table with all possible total states in the system. This is shown in Table 9.2 for the gated latch. Each row in the table specifies a total state, which consists of a letter designation for the internal state and a

Table 9.2
Gated-Latch Total States

	Inputs		Output	
State	***D***	***G***	***Q***	**Comments**
a	0	1	0	$D = Q$ because $G = 1$
b	1	1	1	$D = Q$ because $G = 1$
c	0	0	0	After state a or d
d	1	0	0	After state c
e	1	0	1	After state b or f
f	0	0	1	After state e

possible input combination for D and G. The output Q is also shown for each total state. We start with the two total states that have $G = 1$. From the design specifications, we know that $Q = 0$ if $DG = 01$ and $Q = 1$ if $DG = 11$, because D must be equal to Q when $G = 1$. We assign these conditions to states a and b. When G goes to 0, the output depends on the last value of D. Thus, if the transition of DG is from 01 to 00 to 10, then Q must remain 0 because D is 0 at the time of the transition from 1 to 0 in G. If the transition of DG is from 11 to 10 to 00, then Q must remain 1. This information results in six different total states, as shown in the table. Note that simultaneous transitions of two input variables, such as from 01 to 10 or from 11 to 00, are not allowed in fundamental-mode operation.

The primitive flow table for the gated latch is shown in Fig. 9.16. It has one row for each state and one column for each input combination. First, we fill in one square in each row belonging to the stable state in that row. These entries are determined from Table 9.2. For example, state a is stable and the output is 0 when the input is 01. This information is entered into the flow table in the first row and second column. Similarly, the other five stable states together with their output are entered into the corresponding input columns.

Next, we note that since both inputs are not allowed to change simultaneously, we can enter dash marks in each row that differs in two or more variables from the input variables associated with the stable state. For example, the first row in the flow table shows a stable state with an input of 01. Since only one input can change at any given time, it can change to 00 or 11, but not to 10. Therefore, we enter two dashes in the 10 column of row a. This will eventually result in a don't-care condition for the next state and output in this square. Following the same procedure, we fill in a second square in each row of the primitive flow table.

Next, it is necessary to find values for two more squares in each row. The comments listed in Table 9.2 may help in deriving the necessary information. For example, state c is associated with input 00 and is reached after a change in input from state a or d. Therefore, an unstable state

		Inputs DG			
		00	01	11	10
States	a	c, –	$\textcircled{a}$, 0	b, –	–, –
	b	–, –	a, –	$\textcircled{b}$, 1	e, –
	c	$\textcircled{c}$, 0	a, –	–, –	d, –
	d	c, –	–, –	b, –	$\textcircled{d}$, 0
	e	f, –	–, –	b, –	$\textcircled{e}$, 1
	f	$\textcircled{f}$, 1	a, –	–, –	e, –

FIGURE 9.16
Primitive flow table

c is shown in column 00 and rows *a* and *d* in the flow table. The output is marked with a dash to indicate a don't-care condition. The interpretation of this situation is that if the circuit is in stable state *a* and the input changes from 01 to 00, the circuit first goes to an unstable next state *c*, which changes the present-state value from *a* to *c*, causing a transition to the third row and first column of the table. The unstable state values for the other squares are determined in a similar manner. All outputs associated with unstable states are marked with a dash to indicate don't-care conditions. The assignment of actual values to the outputs is discussed further, after the design example is completed.

Reduction of the Primitive Flow Table

The primitive flow table has only one stable state in each row. The table can be reduced to a smaller number of rows if two or more stable states are placed in the same row. The grouping of stable states from separate rows into one common row is called *merging*. Merging a number of stable states in the same row means that the binary state variable ultimately assigned to the merged row will not change when the input variable changes. This is because, in a primitive flow table, the state variable changes every time the input changes, but in a reduced flow table, a change of input will not cause a change in the state variable if the next stable state is in the same row.

A formal procedure for reducing a flow table is given in Section 9.5. In order to complete the design example in the current section without going through the formal procedure, we will apply the merging process by using a simplified version of the merging rules. Two or more rows in the primitive flow table can be merged into one row if there are nonconflicting states and outputs in each of the columns. Whenever one state symbol and don't-care entries are encountered in the same column, the state is listed in the merged row. Moreover, if the state is circled in one of the rows, it is also circled in the merged row. The output value is included with each stable state in the merged row. Because the merged states have the same output, the state cannot be distinguished on the basis of the output.

We now apply these rules to the primitive flow table of Fig. 9.16. To see how this is done, the primitive flow table is separated into two parts of three rows each, as shown in Fig. 9.17(a). Each part shows three stable states that can be merged because there are no conflicting entries in each of the four columns. The first column shows state *c* in all the rows and 0 or a dash for the output. Since a dash represents a don't-care condition, it can be associated with any state or output. The two dashes in the first column can be taken to be 0 output to make all three rows identical to a stable state *c* with a 0 output. The second column shows that the dashes can be assigned to correspond to a stable state *a* with a 0 output. Note that if a state is circled in one of the rows, it is also circled in the merged row. Similarly, the third column can be merged into an unstable state *b* with a don't-care output, and the fourth column can be merged into stable state *d* and a 0 output. Thus, the three rows *a*, *c*, and *d* can be merged into one row with three stable states and one unstable state, as shown in the first row of Fig. 9.17(b). The second row of the reduced table results from the merging of rows *b*, *e*, and *f* of the primitive flow table. In this example, there are two ways that the reduced table can be drawn. First, the letter symbols for the states can be retained to show the relationship between the reduced and primitive flow tables. Alternatively, because the two tables have the same output, we can assign a common letter symbol to all of the stable states of the merged rows. Thus, states *c* and *d* are replaced by state *a*, and states *e* and *f* are replaced by state *b*. Both alternatives are shown in Fig. 9.17(b).

States \ DG	00	01	11	10
a	c, –	ⓐ, 0	b, –	–, –
c	ⓒ, 0	a, –	–, –	d, –
d	c, –	–, –	b, –	ⓓ, 0

States \ DG	00	01	11	10
b	–, –	a, –	ⓑ, 1	e, –
e	f, –	–, –	b, –	ⓔ, 1
f	ⓕ, 1	a, –	–, –	e, –

(a) States that are candidates for merging

States \ DG	00	01	11	10
a, c, d	ⓒ, 0	ⓐ, 0	b, –	ⓓ, 0
b, e, f	ⓕ, 1	a, –	ⓑ, 1	ⓔ, 1

States \ DG	00	01	11	10
a	ⓐ, 0	ⓐ, 0	b, –	ⓐ, 0
b	ⓑ, 1	a, –	ⓑ, 1	ⓑ, 1

(b) Reduced table (two alternatives)

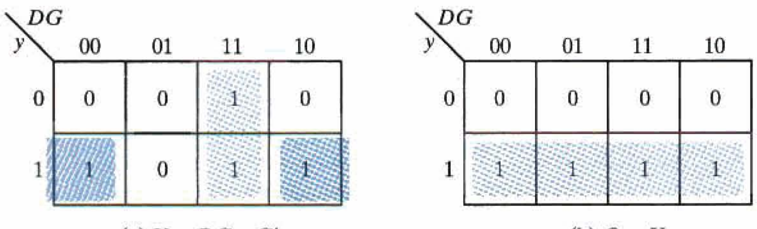
FIGURE 9.17
Reduction of the primitive flow table

Transition Table and Logic Diagram

In order to obtain the circuit described by the reduced flow table, it is necessary to assign a distinct binary value to each state. This assignment converts the flow table into a transition table. In the general case, a binary state assignment must be made to ensure that the circuit will be free of critical races. The state-assignment problem in asynchronous sequential circuits and ways to solve it are discussed in Section 9.6. Fortunately, there can be no critical races in a two-row flow table; therefore, we can finish the design of the gated latch prior to studying that section. Assigning 0 to state a and 1 to state b in the reduced flow table of Fig. 9.17(b), we obtain the transition table of Fig. 9.18(a). The transition table is, in effect, a map for the excitation variable Y. The simplified Boolean function for Y is then obtained from the map as

$$Y = DG + G'y$$

There are two don't-care outputs in the final reduced flow table. If we assign values to the output as shown in Fig. 9.18(b), it is possible to make output Q identical to the map of the excitation function Y. Alternatively, if we replace the don't-care by 1 when $y = 1$ and $DG = 01$, the map reduces to $Q = Y$. If we assign the other possible values to the don't-care outputs, we can make output Q equal to y. In either case, the logic diagram of the gated latch is as shown in Fig. 9.19.



$y \backslash DG$	00	01	11	10
0	0	0	1	0
1	1	0	1	1

(a) $Y = DG + G'y$

$y \backslash DG$	00	01	11	10
0	0	0	0	0
1	1	1	1	1

(b) $Q = Y$

FIGURE 9.18
Transition table and output map for gated latch

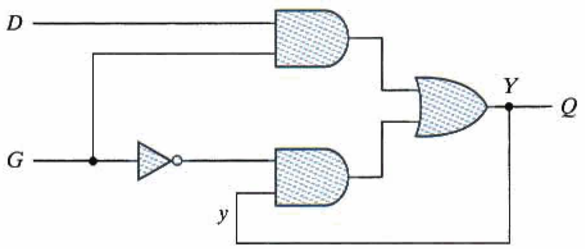


FIGURE 9.19
Gated-latch logic diagram

The diagram can also be implemented by an *SR* latch. Using the procedure outlined in Section 9.3, we first obtain the Boolean functions for *S* and *R*, as shown in Fig. 9.20(a). The logic diagram with NAND gates (see Fig. 5.4) is shown in Fig. 9.20(b). Note that the gated latch is a level-sensitive *D*-latch, introduced in Section 5.3 and Fig. 5.6.

Assigning Outputs to Unstable States

The stable states in a flow table have specific output values associated with them. The unstable states have unspecified output entries designated by a dash. The output values for the unstable states must be chosen so that no momentary false outputs occur when the circuit switches between stable states. This means that if an output variable is not supposed to change as the result of a transition, then an unstable state that is a transient state between two stable states must have the same output value as the stable states. Consider, for example, the flow table of Fig. 9.21(a). A transition from stable state *a* to stable state *b* goes through the unstable state *b*. If the output assigned to the unstable state *b* is a 1, then a momentary short pulse will appear on the output as the circuit shifts from an output of 0 in state *a* to an output of 1 for the unstable *b* and back to 0 when the circuit reaches stable state *b*. Thus, the output corresponding to unstable state *b* must be specified as 0 to avoid a momentary false output.

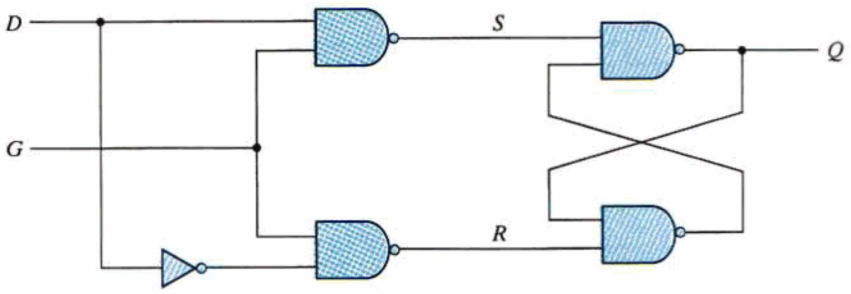
y \ DG	00	01	11	10
0	0	0	1	0
1	X	0	X	X

(a) $S = DG$

y \ DG	00	01	11	10
0	X	X	0	X
1	0	1	0	0

$R = D'G$

(a) Maps for S and R



(b) Logic diagram

FIGURE 9.20
Circuit with *SR* latch

a	a, 0	b , –
b	c , –	b, 0
c	c, 1	d , –
d	a, –	d, 1

(a) Flow table

0	0
X	0
1	1
X	1

(b) Output assignment

FIGURE 9.21
Assigning output values to unstable states

If an output variable is to change value as a result of a change in state, then this variable is assigned a don't-care condition. For example, the transition from stable state b to stable state c in Fig. 9.21(a) changes the output from 0 to 1. If a 0 is entered as the output value for the unstable state c, then the change in the output variable will not take place until the end of the transition. If a 1 is entered, the change will take place at the start of the transition. Since it makes no difference when the change in output occurs, we place a don't-care entry for the output associated with unstable state *c*. Fig. 9.21(b) shows the output assignment for the flow table, demonstrating the four possible combinations of changes in output that can occur. The procedure for making the assignment to outputs associated with unstable states can be summarized as follows:

1. Assign a 0 to an output variable associated with an unstable state which is a transient state between two stable states that have a 0 in the corresponding output variable.
2. Assign a 1 to an output variable associated with an unstable state which is a transient state between two stable states that have a 1 in the corresponding output variable.
3. Assign a don't-care condition to an output variable associated with an unstable state which is a transient state between two stable states that have different values (0 and 1, or 1 and 0) in the corresponding output variable.

Summary of Design Procedure

The design of asynchronous sequential circuits can be carried out by using the procedure illustrated in the previous example. Some of the design steps need further elaboration and are explained in upcoming sections. The procedural steps are as follows:

1. Obtain a primitive flow table from the given design specifications. This is the most difficult part of the design, because it is necessary to use intuition and experience to arrive at the correct interpretation of the problem specifications.
2. Reduce the flow table by merging rows in the primitive flow table. A formal procedure for merging rows in the flow table is given in Section 9.5.
3. Assign binary state variables to each row of the reduced flow table to obtain the transition table. The state-assignment procedure that eliminates any possible critical races is given in Section 9.6.
4. Assign output values to the dashes associated with the unstable states to obtain the output maps. This procedure was explained previously.
5. Simplify the Boolean functions of the excitation and output variables and draw the logic diagram, as shown in Section 9.2. The logic diagram can be drawn with *SR* latches, as shown in Section 9.3 and also at the end of Section 9.7.

9.5 REDUCTION OF STATE AND FLOW TABLES

The procedure for reducing the number of internal states in an asynchronous sequential circuit resembles the procedure that is used for synchronous circuits. An algorithm for the state reduction of a completely specified state table was given in Section 5.7. We will review this algorithm and apply it to a state-reduction method that uses an implication table. The algorithm and the implication table will then be modified to cover the state reduction of incompletely specified state tables. The modified algorithm will be used to explain the procedure for reducing the flow table of asynchronous sequential circuits.

Table 9.3
State Table to Demonstrate Equivalent States

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

Implication Table and Implied States

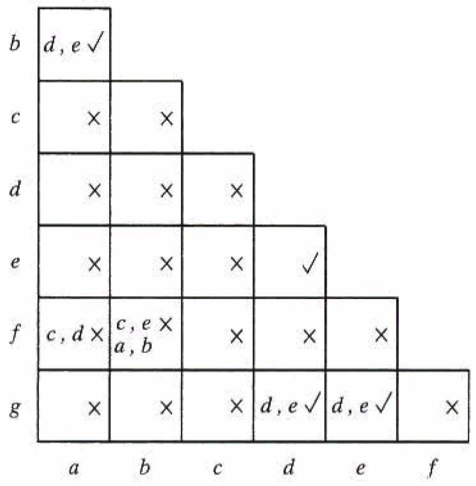
The state-reduction procedure for completely specified state tables is based on an algorithm that combines two states in a state table into one, as long as they can be shown to be equivalent. Two states are equivalent if, for each possible input, they give exactly the same output and go to the same next states or to equivalent next states. Table 6.6 shows an example of equivalent states that have the same next states and outputs for each combination of inputs. There are occasions when a pair of states do not have the same next states, but, nonetheless, go to equivalent next states. Consider, for example, the state table shown in Table 9.3. The present states a and b have the same output for the same input. Their next states are c and d for $x = 0$ and b and a for $x = 1$. If we can show that the pair of states (c, d) are equivalent, then the pair of states (a, b) will also be equivalent, because they will have the same or equivalent next states. When this relationship exists, we say that (a, b) *imply* (c, d) in the sense that if a and b are equivalent then c and d have to be equivalent. Similarly, from the last two rows of Table 9.3, we find that the pair of states (c, d) implies the pair of states (a, b). The characteristic of equivalent states is that if (a, b) imply (c, d) and (c, d) imply (a, b), then both pairs of states are equivalent; that is, a and b are equivalent, and so are c and d. As a consequence, the four rows of Table 9.3 can be reduced to two rows by combining a and b into one state and c and d into a second state.

The checking of each pair of states for possible equivalence in a table with a large number of states can be done systematically by means of an implication table, which is a chart that consists of squares, one for every possible pair of states, that provide spaces for listing any possible implied states. By judicious use of the table, it is possible to determine all pairs of equivalent states. The state table of Table 9.4 will be used to illustrate this procedure. The implication table is shown in Fig. 9.22. On the left side along the vertical are listed all the states defined in the state table except the first, and across the bottom horizontally are listed all the states except the last. The result is a display of all possible combinations of two states, with a square placed in the intersection of a row and a column where the two states can be tested for equivalence. Two states having different outputs for the same input are not equivalent.

Two states that are not equivalent are marked with a cross $(\times)$ in the corresponding square, whereas their equivalence is recorded with a check mark $(\surd)$. Some of the squares have entries of implied states that must be investigated further to determine whether they are equivalent. The step-by-step procedure of filling in the squares is as follows: First, we place a cross in any square corresponding to a pair of states whose outputs are not equal for every input. In this case,

Table 9.4
State Table to Be Reduced

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



b	d, e ✓					
c	×	×				
d	×	×	×			
e	×	×	×	✓		
f	c, d ×	c, e × a, b	×	×	×	
g	×	×	×	d, e ✓	d, e ✓	×
	a	b	c	d	e	f

FIGURE 9.22
Implication table

state c has a different output than any other state, so a cross is placed in the two squares of row c and the four squares of column c. There are nine other squares in this category in the implication table.

Next, we enter in the remaining squares the pairs of states that are implied by the pair of states representing the squares. We do that starting from the top square in the left column and going down and then proceeding with the next column to the right. From the state table, we see that pair (a, b) implies (d, e), so (d, e) is recorded in the square defined by column a and row b. We proceed in this manner until the entire table is completed. Note that states (d, e) are equivalent because they go to the same next state and have the same output. Therefore, a check mark is recorded in the square defined by column d and row e, indicating that the two states are equivalent and independent of any implied pair.

The next step is to make successive passes through the table to determine whether any additional squares should be marked with a cross. A square in the table is crossed out if it contains at least one implied pair that is not equivalent. For example, the square defined by a and f is marked with a cross next to c, d because the pair (c, d) defines a square that contains a cross. This procedure is repeated until no additional squares can be crossed out. Finally, all the squares that have no crosses are recorded with check marks. These squares define pairs of equivalent states. In this example, the equivalent states are

$$(a, b)\ (d, e)\ (d, g)\ (e, g)$$

We now combine pairs of states into larger groups of equivalent states. The last three pairs can be combined into a set of three equivalent states (d, e, g) because each one of the states in the group is equivalent to the other two. The final partition of the states consists of the equivalent states found from the implication table, together with all the remaining states in the state table that are not equivalent to any other state. This group consists of

$$(a, b)\ (c)\ (d, e, g)\ (f)$$

Thus, Table 9.4 can be reduced from seven states to four, one for each member of the preceding partition. The reduced state table is obtained by replacing state b by a and states e and g by d and is shown in Table 9.5.

Merging of the Flow Table

There are occasions when the state table for a sequential circuit is incompletely specified. This happens when certain combinations of inputs or input sequences never occur because of external or internal constraints. In such a case, the next states and outputs that should have occurred if all inputs were possible are never attained and are regarded as don't-care conditions. Although synchronous sequential circuits may sometimes be represented by incompletely specified state tables, our interest here is with asynchronous sequential circuits, for which the primitive flow table is always incompletely specified.

Incompletely specified states can be combined to reduce the number of states in the flow table. Such states cannot be called equivalent, because the formal definition of equivalence requires that all outputs and next states be specified for all inputs. Instead, two incompletely specified states that can be combined are said to be *compatible*. Two states are compatible if,

Table 9.5
Reduced State Table

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

	00	01	11	10
a	c, –	ⓐ 0	b, –	–, –
b	–, –	a, –	ⓑ 1	e, –
c	ⓒ 0	a, –	–, –	d, –
d	c, –	–, –	b, –	ⓓ 0
e	f, –	–, –	b, –	ⓔ 1
f	ⓕ 1	a, –	–, –	e, –

(a) Primitive flow table

b	✓				
c	✓	d, e ×			
d	✓	d, e ×	✓		
e	c, f ×	✓	d, e × c, f ×	×	
f	c, f ×	✓	×	d, e × c, f ×	✓
	a	b	c	d	e

(b) Implication table

FIGURE 9.23
Flow and implication tables

for each possible input, they have the same output whenever it is specified and their next states are compatible whenever they are specified. All don't-care conditions marked with dashes have no effect in the search for compatible states, as they represent unspecified conditions.

The process that must be applied in order to find a suitable group of compatibles for the purpose of merging a flow table can be divided into three steps:

1. Determine all compatible pairs by using the implication table.
2. Find the maximal compatibles with the use of a merger diagram.
3. Find a minimal collection of compatibles that covers all the states and is closed.

The minimal collection of compatibles is then used to merge the rows of the flow table. We will now proceed to show and explain the three procedural steps, using the primitive flow table from the design example in the previous section.

Compatible Pairs

The procedure for finding compatible pairs is illustrated in Fig. 9.23. The primitive flow table in (a) is the same as Fig. 9.16. The entries in each square represent the next state and output. The dashes represent the unspecified states or outputs. The implication table is used to find compatible states, just as it is used to find equivalent states in the completely specified case. The only difference is that, when comparing rows, we are at liberty to adjust the dashes to fit any desired condition.

Two states are compatible if, in every column of the corresponding rows in the flow table, there are identical or compatible states and if there is no conflict in the output values. For example, rows a and b in the flow table are found to be compatible, but rows a and f will be compatible only if c and f are compatible. However, rows c and f are not compatible, because they have different outputs in the first column. This information is recorded in the implication table. A check mark designates a square whose pair of states are compatible. Those states which are not compatible are marked with a cross. The remaining squares are recorded with the implied pairs that need further investigation.

Once the initial implication table has been filled, it is scanned again to cross out the squares whose implied states are not compatible. The remaining squares that contain check marks define the compatible pairs. In the example of Fig. 9.23, the compatible pairs are

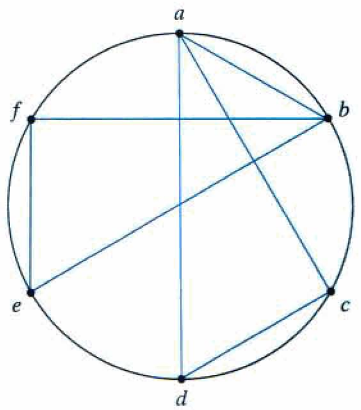
$$(a, b)\ (a, c)\ (a, d)\ (b, e)\ (b, f)\ (c, d)\ (e, f)$$

Maximal Compatibles

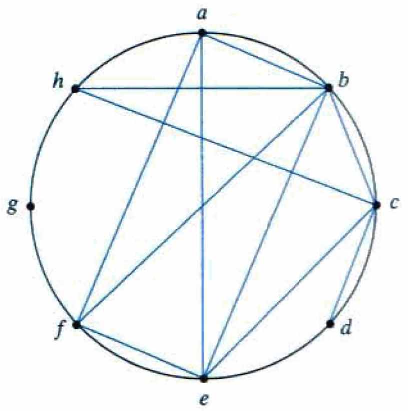
Having found all the compatible pairs, the next step is to find larger sets of states that are compatible. The *maximal compatible* is a group of compatibles that contains all the possible combinations of compatible states. The maximal compatible can be obtained from a merger diagram, as shown in Fig. 9.24. The merger diagram is a graph in which each state is represented by a dot placed along the circumference of a circle. Lines are drawn between any two corresponding dots that form a compatible pair. All possible compatibles can be obtained from the merger diagram by observing the geometrical patterns in which states are connected to each other. An isolated dot represents a state that is not compatible with any other state. A line represents a compatible pair. A triangle constitutes a compatible with three states. An n-state compatible is represented in the merger diagram by an n-sided polygon with all its diagonals connected.

The merger diagram of Fig. 9.24(a) is obtained from the list of compatible pairs derived from the implication table of Fig. 9.23. There are seven straight lines connecting the dots, one for each compatible pair. The lines form a geometrical pattern consisting of two triangles connecting (a, c, d) and (b, e, f) and a line (a, b). The maximal compatibles are

$$(a, b)\ (a, c, d)\ (b, e, f)$$



(a) Maximal compatible: $(a, b)\ (a, c, d)\ (b, e, f)$



(b) Maximal compatible: $(a, b, e, f)\ (b, c, h)\ (c, d)\ (g)$

FIGURE 9.24
Merger diagrams

Figure 9.24(b) shows the merger diagram of an eight-state flow table. The geometrical patterns are a rectangle with its two diagonals connected to form the four-state compatible (a, b, e, f), a triangle (b, c, h), a line (c, d), and a single state g that is not compatible with any other state. The maximal compatibles are

$$(a, b, e, f)\ (b, c, h)(c, d)\ (g)$$

The maximal compatible set can be used to merge the flow table by assigning one row in the reduced table to each member of the set. However, quite often the maximal compatibles do not necessarily constitute the set of compatibles that is minimal. In many cases, it is possible to find a smaller collection of compatibles that will satisfy the condition for merging rows.

Closed-Covering Condition

The condition that must be satisfied for merging rows is that the set of chosen compatibles must *cover* all the states and must be *closed.* The set will cover all the states if it includes all the states of the original state table. The closure condition is satisfied if there are no implied states or if the implied states are included within the set. A closed set of compatibles that covers all the states is called a *closed covering.* The closed-covering condition will be explained by means of two examples.

Consider the maximal compatibles from Fig. 9.24(a). If we remove (a, b), we are left with a set of two compatibles:

$$(a, c, d)\ (b, e, f)$$

All six states from the flow table in Fig. 9.23 are included in this set. Thus, the set satisfies the covering condition. There are no implied states for (a, c); (a, d); (c, d); (b, e); (b, f); and (e, f), as is seen from the implication table of Fig. 9.23(b), so the closure condition is also satisfied. Therefore, the primitive flow table can be merged into two rows, one for each of the compatibles. The detailed construction of the reduced table for this particular example was done in the previous section and is shown in Fig. 9.17(b).

The second example is from a primitive flow table (not shown) whose implication table is given in Fig. 9.25(a). The compatible pairs derived from the implication table are

$$(a, b)\ (a, d)\ (b, c)\ (c, d)\ (c, e)\ (d, e)$$

From the merger diagram of Fig. 9.25(b), we determine the maximal compatibles:

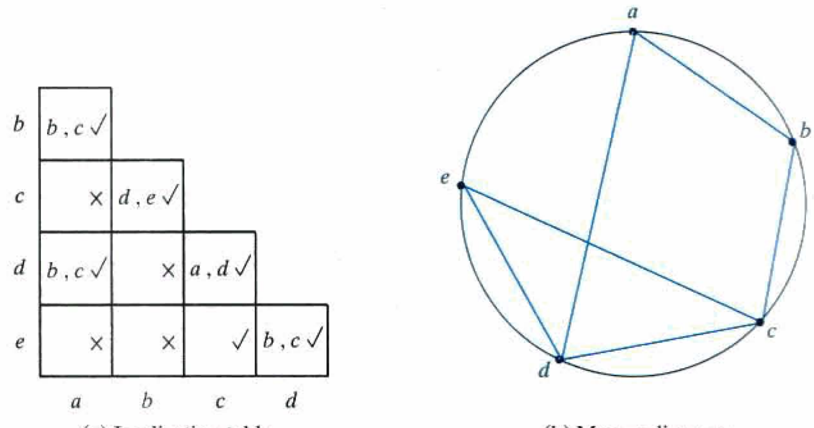
$$(a, b)\ (a, d)\ (b, c)\ (c, d, e)$$

If we choose the two compatibles

$$(a, b)\ (c, d, e)$$

then the set will cover all five states of the original table. The closure condition can be checked by means of a closure table, as shown in Fig. 9.25(c). The implied pairs listed for each compatible are taken directly from the implication table. The implied pair of states for (a, b) is (b, c). But (b, c) is not included in the chosen set of (a, b) (c, d, e), so this set of compatibles is not closed. A set of compatibles that will satisfy the closed-covering condition is

$$(a, d)\ (b, c)\ (c, d, e)$$



b	b , c ✓			
c	×	d , e ✓		
d	b , c ✓	×	a , d ✓	
e	×	×	✓	b , c ✓
	a	b	c	d

(a) Implication table

(b) Merger diagram

Compatibles	(a, b)	(a, d)	(b, c)	(c, d, e)
Implied states	(b, c)	(b, c)	(d, e)	(a, d) (b, c)

(c) Closure table

FIGURE 9.25
Choosing a set of compatibles

The set is covered because it contains all five states. Note that the same state can be repeated more than once. The closure condition is satisfied because the implied states are (b, c) (d, e) and (a, d), which are included in the set. The original flow table (not shown here) can be reduced from five rows to three rows by merging rows a and d, b and c, and c, d, and e. Note also that an alternative satisfactory choice of closed-covered compatibles would be (a, b) (b, c) (d, e). In general, there may be more than one possible way of merging rows when reducing a primitive flow table.

9.6 RACE-FREE STATE ASSIGNMENT

Once a reduced flow table has been derived for an asynchronous sequential circuit, the next step in the design is to assign binary variables to each stable state. This assignment results in the transformation of the flow table into its equivalent transition table. The primary objective in choosing a proper binary state assignment is the prevention of critical races. The problem of critical races was discussed in Section 9.2 in conjunction with Fig. 9.7.

Critical races can be avoided by making a binary state assignment in such a way that only one variable changes at any given time when a state transition occurs in the flow table. To accomplish this objective, it is necessary that states between which transitions occur be given adjacent assignments. Two binary values are said to be *adjacent* if they differ in only one variable. For example, 010 and 011 are adjacent because they differ only in the third bit.

In order to ensure that a transition table has no critical races, it is necessary to test each possible transition between two stable states and verify that the binary state variables change one at a time. This is a tedious process, especially when there are many rows and columns in the table. To simplify matters, we will explain the procedure of binary state assignment by going through examples with only three and four rows in the flow table. These examples will demonstrate the general procedure that must be followed to ensure a race-free state assignment. The procedure can then be applied to flow tables with any number of rows and columns.

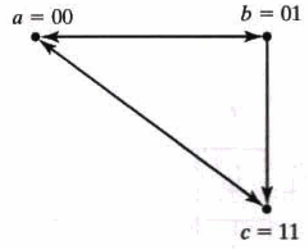
Three-Row Flow-Table Example

The assignment of a single binary variable to a flow table with two rows does not impose critical race problems. A flow table with three rows requires an assignment of two binary variables. The assignment of binary values to the stable states may cause critical races if it is not done properly. Consider, for example, the reduced flow table of Fig. 9.26(a). The outputs have been omitted from the table for simplicity. Inspection of row *a* reveals that there is a transition from state *a* to state *b* in column 01 and from state *a* to state *c* in column 11. This information is transferred into a *transition diagram*, as shown in Fig. 9.26(b). The directed lines from *a* to *b* and from *a* to *c* represent the two transitions just mentioned. Similarly, the transitions from the other two rows are represented by directed lines in the diagram, which is a pictorial representation of all required transitions between rows.

To avoid critical races, we must find a binary state assignment such that only one binary variable changes during each state transition. An attempt to find such an assignment is shown in the transition diagram. State *a* is assigned binary 00, and state *c* is assigned binary 11. This assignment will cause a critical race during the transition from *a* to *c* because there are two changes in the binary state variables and the transition from *a* to *c* may occur directly or pass through *b*. Note that the transition from *c* to *a* also causes a race condition, but it is noncritical because the transition does not pass through other states.

	x_1x_2 00	01	11	10
a	ⓐ	*b*	*c*	ⓐ
b	*a*	ⓑ	ⓑ	*c*
c	*a*	ⓒ	ⓒ	ⓒ

(a) Flow table

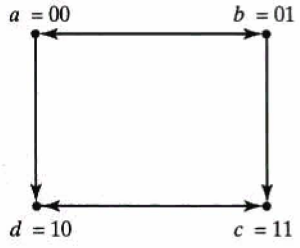


(b) Transition diagram

FIGURE 9.26
Three-row flow-table example

	x_1x_2 00	01	11	10
a	$\textcircled{a}$	b	d	$\textcircled{a}$
b	a	$\textcircled{b}$	$\textcircled{b}$	c
c	d	$\textcircled{c}$	$\textcircled{c}$	$\textcircled{c}$
d	a	-	c	-

(a) Flow table



(b) Transition diagram

FIGURE 9.27
Flow table with an extra row

A race-free assignment can be obtained if we add an extra row to the flow table. The use of a fourth row does not increase the number of binary state variables, but it allows the formation of cycles between two stable states. Consider the modified flow table in Fig. 9.27. The first three rows represent the same conditions as the original three-row table. The fourth row, labeled d, is assigned the binary value 10, which is adjacent to both a and c. The transition from a to c must now go through d, with the result that the binary variables change from $a = 00$, to $d = 10$, to $c = 11$, thus avoiding a critical race. This is accomplished by changing row a, column 11, to d and row d, column 11, to c. Similarly, the transition from c to a is shown to go through unstable state d even though column 00 represents a noncritical race.

The transition table corresponding to the flow table with the indicated binary state assignment is shown in Fig. 9.28. The two dashes in row d represent unspecified states that can be considered don't-care conditions. However, care must be taken not to assign 10 to these squares, in order to avoid the possibility of an unwanted stable state being established in the fourth row.

	x_1x_2 00	01	11	10
$a = 00$	$\textcircled{00}$	01	10	$\textcircled{00}$
$b = 01$	00	$\textcircled{01}$	$\textcircled{01}$	11
$c = 11$	10	$\textcircled{11}$	$\textcircled{11}$	$\textcircled{11}$
$d = 10$	00	-	11	-

FIGURE 9.28
Transition table

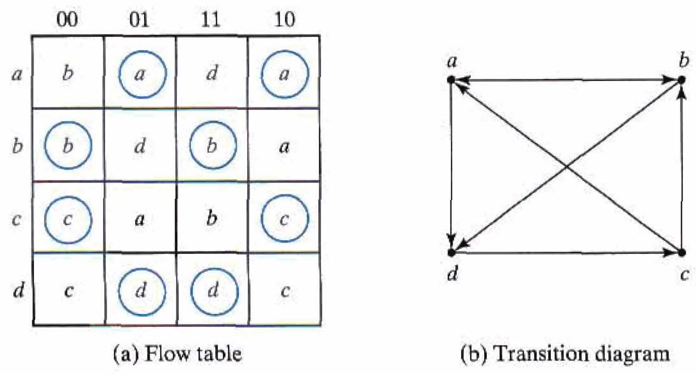


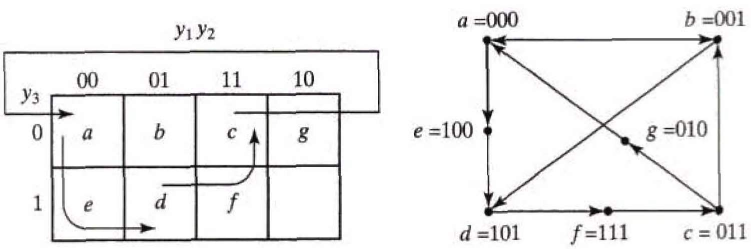
FIGURE 9.29
Four-row flow-table example

This example demonstrates the use of an extra row in the flow table for the purpose of achieving a race-free assignment. The extra row is not assigned to any specific stable state, but instead is used to convert a critical race into a cycle that goes through adjacent transitions between two stable states. Sometimes, just one extra row may not be sufficient to prevent critical races, and it may be necessary to add two or more extra rows in the flow table. This possibility is demonstrated in the next example.

Four-Row Flow-Table Example

A flow table with four rows requires a minimum of two state variables. Although a race-free assignment is sometimes possible with only two binary state variables, in many cases the requirement of extra rows to avoid critical races will dictate the use of three binary state variables. Consider, for example, the flow table and its corresponding transition diagram shown in Fig. 9.29. If there were no transitions in the diagonal direction (from b to d or from c to a), it would be possible to find an adjacent assignment for the remaining four transitions. With one or two diagonal transitions, there is no way of assigning two binary variables that satisfy the adjacency requirement. Therefore, at least three binary state variables are needed.

Figure 9.30 shows a state assignment map that is suitable for any four-row flow table. States a, b, c, and d are the original states, and $e, f,$ and g are extra states. States placed in adjacent squares in the map will have adjacent assignments. State b is assigned binary 001 and is adjacent to the other three original states. The transition from a to d must be directed through the extra state e to produce a cycle so that only one binary variable changes at a time. Similarly, the transition from c to a is directed through g, and the transition from d to c goes through f. By using the assignment given by the map, the four-row table can be expanded to a seven-row table that is free of critical races, as shown in Fig. 9.31. Note that although the flow table has seven rows, there are only four stable states. The uncircled states in the three extra rows are there merely to provide a race-free transition between the stable states.



(a) Binary assignment (b) Transition diagram

FIGURE 9.30
Choosing extra rows for the flow table

	00	01	11	10
000 = *a*	*b*	ⓐ	*e*	ⓐ
001 = *b*	ⓑ	*d*	ⓑ	*a*
011 = *c*	ⓒ	*g*	*b*	ⓒ
010 = *g*	–	*a*	–	–
110 –	–	–	–	–
111 = *f*	*c*	–	–	*c*
101 = *d*	*f*	ⓓ	ⓓ	*f*
100 = *e*	–	–	*d*	–

FIGURE 9.31
State assignment to modified flow table

This example demonstrates a possible way of selecting extra rows in a flow table in order to achieve a race-free assignment. A state-assignment map similar to the one used in Fig. 9.30(a) can be helpful in most cases. Sometimes we can take advantage of unspecified entries in the flow table. Instead of adding rows to the table, we may be able to eliminate critical races by directing some of the state transitions through the don't-care entries. The actual assignment is done by trial and error, until a satisfactory assignment is found that resolves all critical races.

Multiple-Row Method

The method for making race-free state assignments by adding extra rows in the flow table, as demonstrated in the previous two examples, is sometimes referred to as the *shared-row* method. A second method, called the *multiple-row* method, is not as efficient, but is easier to apply. In multiple-row assignment, each state in the original flow table is replaced by two or more combinations of state variables. The state-assignment map of Fig. 9.32(a) shows a multiple-row assignment that can be used with any four-row flow table. There are two binary state variables for each stable state, each variable being the logical complement of the other. For example, the original state a is replaced with two equivalent states $a_1 = 000$ and $a_2 = 111$. The output values, not shown here, must be the same in a_1 and a_2. Note that a_1 is adjacent to b_1, c_2, and d_1, and a_2 is adjacent to c_1, b_2, and d_2, and, similarly, each state is adjacent to three states with different letter designations. The behavior of the circuit is the same whether the internal state is a_1 or a_2, and so on for the other states.

Figure 9.32(b) shows the multiple-row assignment for the original flow table of Fig. 9.29(a). The expanded table is formed by replacing each row of the original table with two rows. For example, row b is replaced by rows b_1 and b_2, and stable state b is entered in columns 00 and 11 in both rows b_1 and b_2. After all the stable states have been entered, the unstable states are filled in by reference to the assignment specified in the map of part (a). In choosing the next state for a given present state, a state that is adjacent to the present state is selected from the map. In the original table, the next states of b are a and d for inputs 10 and 01, respectively. In the expanded table, the next states of b_1 are a_1 and d_2, because these are the states adjacent to b_1. Similarly, the next states of b_2 are a_2 and d_1, because they are adjacent to b_2.

y_1 \ y_2y_3	00	01	11	10
0	a_1	b_1	c_1	d_1
1	c_2	d_2	a_2	b_2

(a) Binary assignment

	00	01	11	10
$000 = a_1$	b_1	a_1	d_1	a_1
$111 = a_2$	b_2	a_2	d_2	a_2
$001 = b_1$	b_1	d_2	b_1	a_1
$110 = b_2$	b_2	d_1	b_2	a_2
$011 = c_1$	c_1	a_2	b_1	c_1
$100 = c_2$	c_2	a_1	b_2	c_2
$010 = d_1$	c_1	d_1	d_1	c_1
$101 = d_2$	c_2	d_2	d_2	c_2

(b) Flow table

FIGURE 9.32
Multiple-row assignment

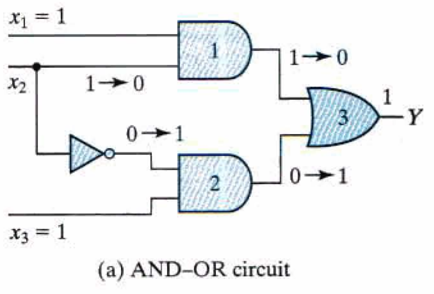
In the multiple-row assignment, the change from one stable state to another will always cause a change of only one binary state variable. Each stable state has two binary assignments with exactly the same output. At any given time, only one of the assignments is in use. For example, if we start with state a_1 and input 01 and then change the input to 11, 01, 00, and back to 01, the sequence of internal states will be a_1, d_1, c_1, and a_2. Although the circuit starts in state a_1 and terminates in state a_2, as far as the input–output relationship is concerned, the two states a_1 and a_2 are equivalent to state a of the original flow table.

9.7 HAZARDS

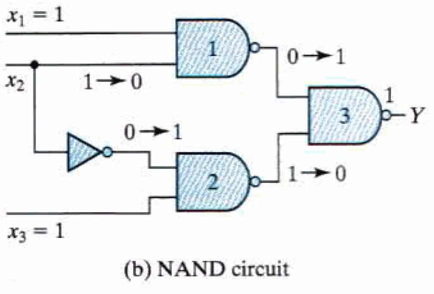
In designing asynchronous sequential circuits, care must be taken to conform with certain restrictions and precautions to ensure that the circuits operate properly. The circuit must be operated in fundamental mode with only one input changing at any time and must be free of critical races. In addition, there is one more phenomenon, called a *hazard*, that may cause the circuit to malfunction. Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays. Hazards occur in combinational circuits, where they may cause a temporary false output value. When they occur in asynchronous sequential circuits, hazards may result in a transition to a wrong stable state. It is therefore necessary to check for possible hazards and determine whether they can cause improper operations. If so, then steps must be taken to eliminate their effect.

Hazards in Combinational Circuits

A hazard is a condition in which a change in a single variable produces a momentary change in output when no change in output should occur. The circuit of Fig. 9.33(a) depicts the occurrence of a hazard. Assume that all three inputs are initially equal to 1. This causes the output of gate 1 to be 1, that of gate 2 to be 0, and that of the circuit to be 1. Now consider a change in x_2 from 1 to 0. Then the output of gate 1 changes to 0 and that of gate 2 changes to 1, leaving the output at 1. However, the output may momentarily go to 0 if the propagation delay through the inverter is taken into consideration. The delay in the inverter may cause the output of gate 1 to change to 0 before the output of gate 2 changes to 1. In that case, both inputs



(a) AND–OR circuit



(b) NAND circuit

FIGURE 9.33
Circuits with hazards

of gate 3 are momentarily equal to 0, causing the output to go to 0 for the short time during which the input signal from x_2 is delayed while it is propagating through the inverter circuit.

The circuit of Fig. 9.33(b) is a NAND implementation of the Boolean function in Fig. 9.33(b), and it has a hazard for the same reason. Because gates 1 and 2 are NAND gates, their outputs are the complement of the outputs of the corresponding AND gates. When x_2 changes from 1 to 0, both inputs of gate 3 may be equal to 1, causing the output to produce a momentary change to 0 when it should have stayed at 1.

The two circuits shown in Fig. 9.33 implement the Boolean function in sum-of-products form:

$$Y = x_1x_2 + x_2'x_3$$

This type of implementation may cause the output to go to 0 when it should remain a 1. If, however, the circuit is implemented instead in product-of-sums form (see Section 3.5), namely,

$$Y = (x_1 + x_2')(x_2 + x_3)$$

then the output may momentarily go to 1 when it should remain 0. The first case is referred to as *static 1-hazard* and the second case as *static 0-hazard.* A third type of hazard, known as *dynamic hazard,* causes the output to change three or more times when it should change from 1 to 0 or from 0 to 1. Figure 9.34 illustrates the three types of hazards. When a circuit is implemented in sum-of-products form with AND–OR gates or with NAND gates, the removal of static 1-hazard guarantees that no static 0-hazards or dynamic hazards will occur.

A hazard can be detected by inspection of the map of the particular circuit. To illustrate, consider the map in Fig. 9.35(a), which is a plot of the function implemented in Fig. 9.33. The change in x_2 from 1 to 0 moves the circuit from minterm 111 to minterm 101. The hazard exists because the change in input results in a different product term covering the two minterms.

FIGURE 9.34
Types of hazards

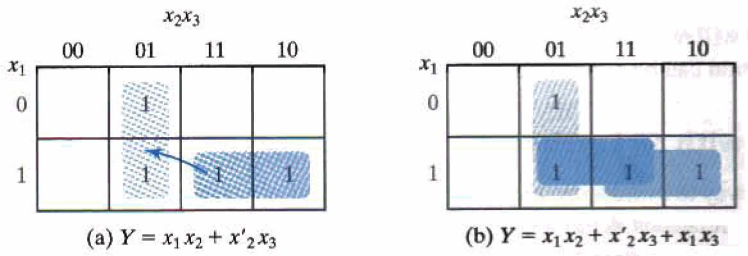

x_2x_3
x_1
00
01
11
10
0
1
1
1
1
1
(a) $Y = x_1x_2 + x'_2x_3$
x_2x_3
x_1
00
01
11
10
0
1
1
1
1
1
(b) $Y = x_1x_2 + x'_2x_3 + x_1x_3$

FIGURE 9.35
Maps illustrating a hazard and its removal

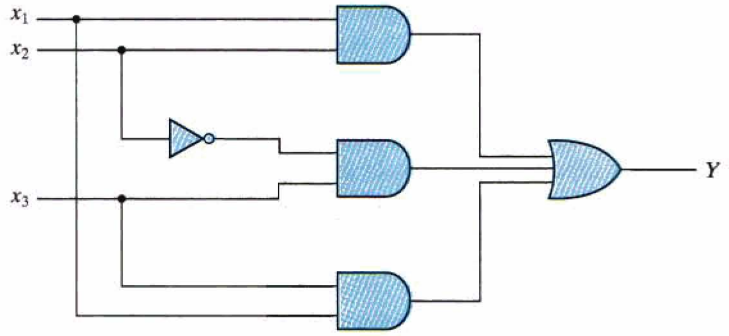


FIGURE 9.36
Hazard-free circuit

Minterm 111 is covered by the product term implemented in gate 1 of Fig. 9.33, and minterm 101 is covered by the product term implemented in gate 2. Whenever the circuit must move from one product term to another, there is a possibility of a momentary interval when neither term is equal to 1, giving rise to an undesirable 0 output.

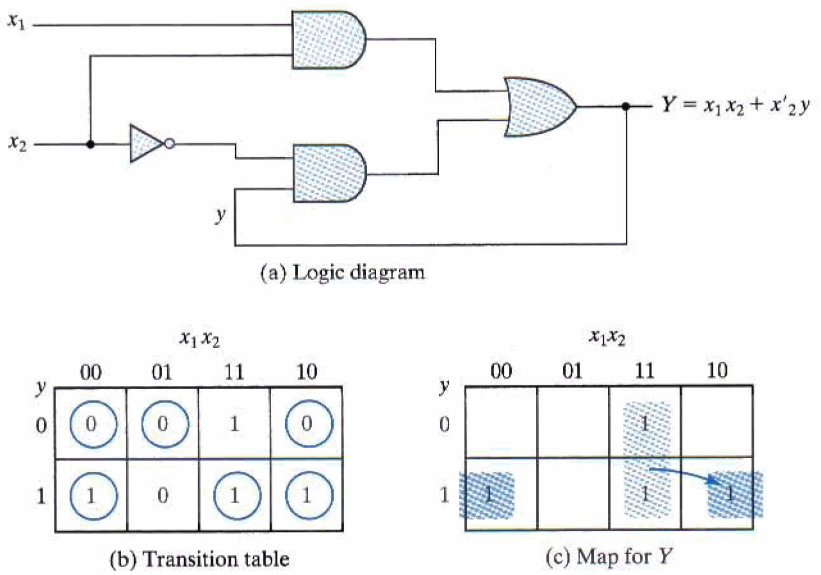
The remedy for eliminating a hazard is to enclose the two minterms in question with another product term that overlaps both groupings. This situation is shown in the map of Fig. 9.35(b), where the two minterms that cause the hazard are combined into one product term. The hazard-free circuit obtained by such a configuration is shown in Fig. 9.36. The extra gate in the circuit generates the product term x_1x_3. In general, hazards in combinational circuits can be removed by covering any two minterms that may produce a hazard with a product term common to both. The removal of hazards requires the addition of redundant gates to the circuit.

Hazards in Sequential Circuits

In normal combinational-circuit design associated with synchronous sequential circuits, hazards are of no concern, since momentary erroneous signals are not generally troublesome. However, if a momentary incorrect signal is fed back in an asynchronous sequential circuit, it may cause the circuit to go to the wrong stable state. This situation is illustrated in Fig. 9.37. If the circuit is in total stable state $yx_1x_2 = 111$ and input x_2 changes from 1 to 0, the next total stable state should be 110. However, because of the hazard, output Y may go to 0 momentarily. If this false signal feeds back into gate 2 before the output of the inverter goes to 1, the output of gate 2 will remain at 0 and the circuit will switch to the incorrect total stable state 010. This malfunction can be eliminated by adding an extra gate, as is done in Fig. 9.36.

Implementation with *SR* Latches

Another way to avoid static hazards in asynchronous sequential circuits is to implement the circuit with *SR* latches. A momentary 0 signal applied to the *S* or *R* inputs of a NOR latch will have no effect on the state of the circuit. Similarly, a momentary 1 signal applied to the *S* and *R* inputs of a NAND latch will have no effect on the state of the latch. In Fig. 9.33(b), we observed



(a) Logic diagram

(b) Transition table

(c) Map for Y

FIGURE 9.37
Hazard in an asynchronous sequential circuit

that a two-level sum-of-products expression implemented with NAND gates may have a static 1-hazard if both inputs of gate 3 go to 1, changing the output from 1 to 0 momentarily. But if gate 3 is part of a latch, the momentary 1 signal will have no effect on the output, because a third input to the gate will come from the complemented side of the latch that will be equal to 0 and thus maintain the output at 1. To clarify what was just said, consider a NAND *SR* latch with the following Boolean functions for *S* and *R*:

$$S = AB + CD$$
$$R = A'C$$

Since this is a NAND latch, we must apply the complemented values to the inputs:

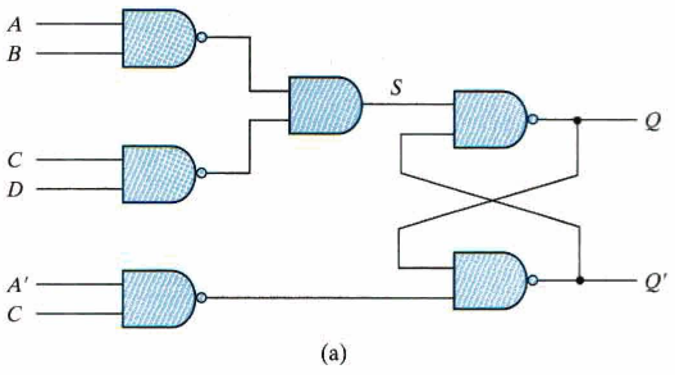
$$S = (AB + CD)' = (AB)'(CD)'$$
$$R = (A'C)'$$

This implementation is shown in Fig. 9.38(a). *S* is generated with two NAND gates and one AND gate. The Boolean function for output *Q* is

$$Q = (Q'S)' = [Q'(AB)'(CD)']'$$

This function is generated in Fig. 9.38(b) with two levels of NAND gates. If output *Q* is equal to 1, then *Q'* is equal to 0. If two of the three inputs go momentarily to 1, the NAND gate associated with output *Q* will remain at 1 because *Q'* is maintained at 0.

Figure 9.38(b) shows a typical circuit that can be used to construct asynchronous sequential circuits. The two NAND gates forming the latch normally have two inputs. However, if the



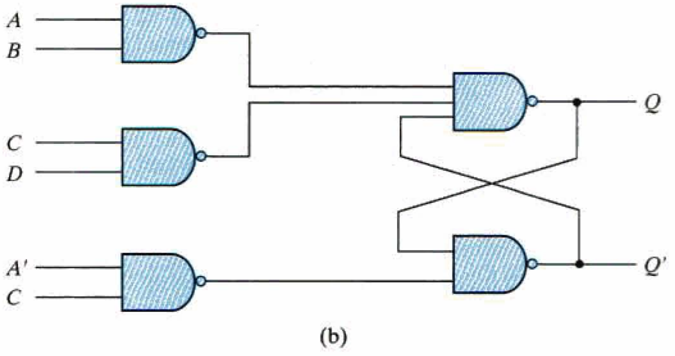


FIGURE 9.38
Latch implementation

S or *R* functions contain two or more product terms when expressed as a sum of products, then the corresponding NAND gate of the *SR* latch will have three or more inputs. Thus, the two terms in the original sum-of-products expression for *S* are *AB* and *CD*, and each is implemented with a NAND gate whose output is applied to the input of the NAND latch. In this way, each state variable requires a two-level circuit of NAND gates. The first level consists of NAND gates that implement each product term in the original Boolean expression of *S* and *R*. The second level forms the cross-coupled connection of the *SR* latch with inputs that come from the outputs of each NAND gate in the first level.

Essential Hazards

Thus far, we have considered what are known as static and dynamic hazards. Another type of hazard that may occur in asynchronous sequential circuits is called an *essential hazard.* This type of hazard is caused by unequal delays along two or more paths that originate from the same input. An excessive delay through an inverter circuit in comparison to the delay associated with the feedback path may cause such a hazard. Essential hazards cannot be corrected by adding redundant gates as in static hazards. The problem that they impose can be corrected by adjusting the amount of delay in the affected path. To avoid essential hazards, each feedback loop must be handled with individual care to ensure that the delay in the feedback path is long enough compared with delays of other signals that originate from the input terminals. This problem tends to be specialized, as it depends on the particular circuit used and the size of the delays that are encountered in its various paths.

9.8 DESIGN EXAMPLE

We are now in a position to examine a complete design example of an asynchronous sequential circuit. This example may serve as a reference for the design of other, similar circuits. We will demonstrate the method of design by following the recommended procedural steps listed at the end of Section 9.4 and repeated next. After stating the design specifications,

1. Derive a primitive flow table.
2. Reduce the flow table by merging the rows.
3. Make a race-free binary state assignment.
4. Obtain the transition table and output map.
5. Obtain the logic diagram, using *SR* latches.

Design Specifications

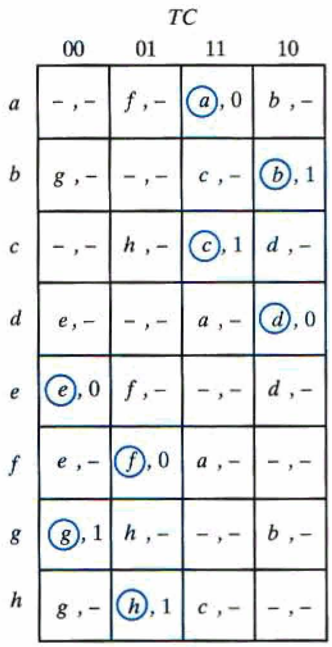
It is necessary to design a negative-edge-triggered *T* flip-flop. The circuit has two inputs, *T* (toggle) and *C* (clock), and one output, *Q*. The output state is complemented if $T = 1$ and the clock *C* changes from 1 to 0 (negative-edge triggering). Otherwise, under any other input condition, the output *Q* remains unchanged. Although this circuit can be used as a flip-flop in clocked sequential circuits, the internal design of the flip-flop (as is the case with all other flip-flops) is an asynchronous problem.

Primitive Flow Table

The derivation of the primitive flow table can be facilitated if we first derive a table that lists all possible total states in the circuit. This table is shown in Table 9.6. We start with the input condition $TC = 11$ and assign to it state *a*. The circuit goes to state *b* and the output *Q* is complemented from 0 to 1 when *C* changes from 1 to 0 while *T* remains a 1. Another change in the output occurs when the circuit goes from state *c* to state *d*. In this case, $T = 1$, *C* changes from 1 to 0, and the output *Q* is complemented from 1 to 0. The other four states in the table do not change the output, because *T* is equal to 0. If *Q* is initially 0, it stays at 0, and if it is initially at 1, it stays at 1, even though the clock input changes. This analysis identifies six total states. Note that simultaneous transitions of two input variables, such as that from 01 to 10, are not included, as they violate the condition for fundamental-mode operation.

Table 9.6
Specification of Total States

	Inputs		Output	
State	***T***	***C***	***Q***	**Comments**
a	1	1	0	Initial output is 0
b	1	0	1	After state *a*
c	1	1	1	Initial output is 1
d	1	0	0	After state *c*
e	0	0	0	After state *d* or *f*
f	0	1	0	After state *e* or *a*
g	0	0	1	After state *b* or *h*
h	0	1	1	After state *g* or *c*



	TC			
	00	01	11	10
a	–, –	*f*, –	$\textcircled{a}$, 0	*b*, –
b	*g*, –	–, –	*c*, –	$\textcircled{b}$, 1
c	–, –	*h*, –	$\textcircled{c}$, 1	*d*, –
d	*e*, –	–, –	*a*, –	$\textcircled{d}$, 0
e	$\textcircled{e}$, 0	*f*, –	–, –	*d*, –
f	*e*, –	$\textcircled{f}$, 0	*a*, –	–, –
g	$\textcircled{g}$, 1	*h*, –	–, –	*b*, –
h	*g*, –	$\textcircled{h}$, 1	*c*, –	–, –

FIGURE 9.39
Primitive flow table

The primitive flow table is shown in Fig. 9.39. The information for the table can be obtained directly from the conditions listed in Table 9.6. First, in each row, we fill in one square belonging to the stable state in that row, as listed in the table. Then we enter dashes in those squares whose input differs by two variables from the input corresponding to the stable state. Finally, we identify the unstable conditions by utilizing the information listed under the comments in Table 9.6.

b	a, c ×						
c	×	b, d ×					
d	b, d ×	×	a, c ×				
e	b, d ×	e, g × b, d ×	f, h ×	✓			
f	✓	e, g × a, c ×	f, h × a, c ×	✓	✓		
g	f, h ×	✓	b, d ×	e, g × b, d ×	×	e, g × f, h ×	
h	f, h × a, c ×	✓	✓	d, e × c, f ×	e, g × f, h ×	×	✓
	a	b	c	d	e	f	g

FIGURE 9.40
Implication table

Merging of the Flow Table

The rows in the primitive flow table are merged by first obtaining all compatible pairs of states. This is done by means of the implication table shown in Fig. 9.40. The squares that contain check marks define the compatible pairs:

$$(a, f)\ (b, g)\ (b, h)\ (c, h)\ (d, e)\ (d, f)\ (e, f)\ (g, h)$$

The maximal compatibles are obtained from the merger diagram shown in Fig. 9.41. The geometrical patterns that are recognized in the diagram consist of two triangles and two straight lines. The maximal compatible set is

$$(a, f)\ (b, g, h)\ (c, h)\ (d, e, f)$$

In this particular example, the minimal collection of compatibles is also the maximal compatible set. Note that the closed condition is satisfied because the set includes all the original eight states listed in the primitive flow table, although states h and f are repeated. The covering condition is also satisfied, because all the compatible pairs have no implied states, as can be seen from the implication table.

The reduced flow table is shown in Fig. 9.42. The table shown in part (a) of the figure retains the original state symbols, but merges the corresponding rows. For example, states a and f are compatible and are merged into one row that retains the original letter symbols of the states. Similarly, the other three compatible sets of states are used to merge the flow table into four rows, retaining the eight original letter symbols. The other alternative for drawing the merged flow table is shown in part (b) of the figure. Here, we assign a common letter symbol

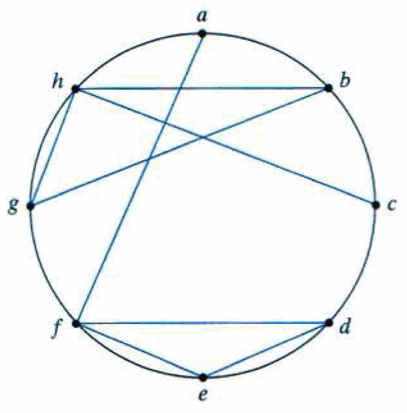


FIGURE 9.41
Merger diagram

	TC			
	00	01	11	10
a, f	*e*, –	ⓕ, 0	ⓐ, 0	*b*, –
b, g, h	ⓖ, 1	ⓗ, 1	*c*, –	ⓑ, 1
c, h	*g*, 1	ⓗ, 1	ⓒ, 1	*d*, –
d, e, f	ⓔ, 0	ⓕ, 0	*a*, –	ⓓ, 0

(a)

	TC			
	00	01	11	10
a	*d*, –	ⓐ, 0	ⓐ, 0	ⓑ, –
b	ⓑ, 1	ⓑ, 1	*c*, –	ⓑ, 1
c	*b*, –	ⓒ, 1	ⓒ, 1	*d*, –
d	ⓓ, 0	ⓓ, 0	*a*, –	ⓓ, 0

(b)

FIGURE 9.42
Reduced flow table

to all the stable states in each merged row. Thus, the symbol *f* is replaced by *a*, *g* and *h* are replaced by *b*, and similarly for the other two rows. The second alternative shows clearly a four-state flow table with only four letter symbols for the states.

State Assignment and Transition Table

The next step in the design is to find a race-free binary assignment for the four stable states in the reduced flow table. In order to find a suitable adjacent assignment, we draw the transition diagram, as shown in Fig. 9.43. For this example, it is possible to obtain a suitable adjacent assignment without the need of extra states, because there are no diagonal lines in the transition diagram.

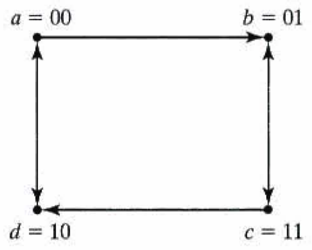


FIGURE 9.43
Transition diagram

y_1y_2 \ TC	00	01	11	10
$a = 00$	10	00	00	01
$b = 01$	01	01	11	01
$c = 11$	01	11	11	10
$d = 10$	10	10	00	10

(a) Transition table

y_1y_2 \ TC	00	01	11	10
00	0	0	0	X
01	1	1	1	1
11	1	1	1	X
10	0	0	0	0

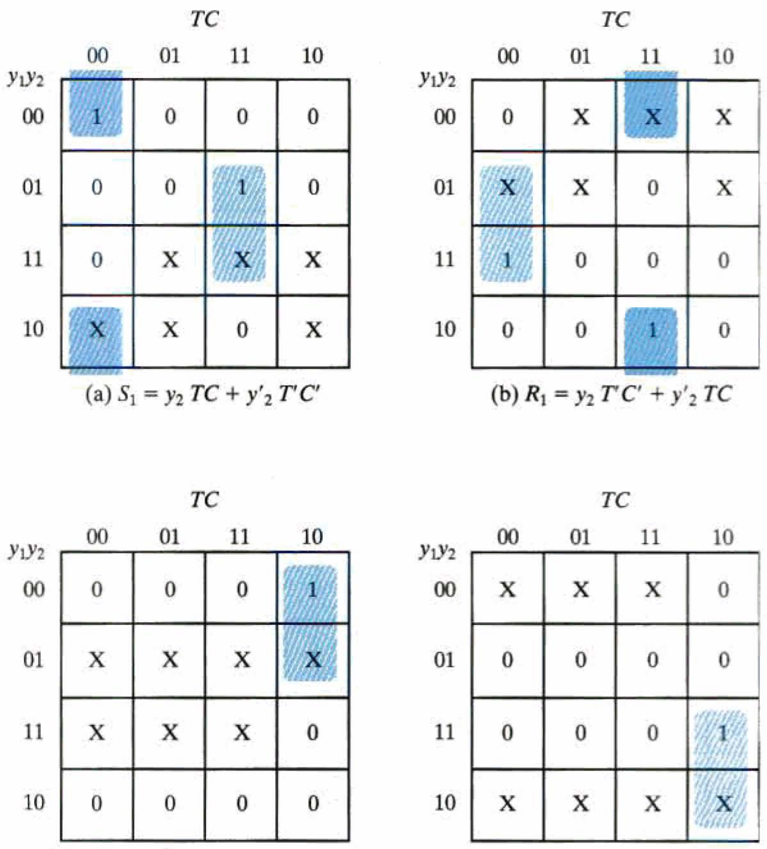
(b) Output map $Q = y_2$

FIGURE 9.44
Transition table and output map

Substituting the binary assignment indicated in the transition diagram into the reduced flow table, we obtain the transition table shown in Fig. 9.44. The output map is obtained from the reduced flow table. The dashes in the output section are assigned values according to the rules established in Section 9.4.

Logic Diagram

The circuit to be designed has two state variables, Y_1 and Y_2, and one output, Q. The output map in Fig. 9.44 shows that Q is equal to the state variable y_2. The implementation of the circuit requires two SR latches, one for each state variable. The maps for inputs S and R of the two latches are shown in Fig. 9.45. The maps are obtained from the information given in the transition table by using the conditions specified in the latch excitation table shown in Fig. 9.14(b). The simplified Boolean functions are listed under each map.



y_1y_2 \ TC	00	01	11	10
00	1	0	0	0
01	0	0	1	0
11	0	X	X	X
10	X	X	0	X

(a) $S_1 = y_2\,TC + y'_2\,T'C'$

y_1y_2 \ TC	00	01	11	10
00	0	X	X	X
01	X	X	0	X
11	1	0	0	0
10	0	0	1	0

(b) $R_1 = y_2\,T'C' + y'_2\,TC$

y_1y_2 \ TC	00	01	11	10
00	0	0	0	1
01	X	X	X	X
11	X	X	X	0
10	0	0	0	0

(c) $S_2 = y'_1\,TC'$

y_1y_2 \ TC	00	01	11	10
00	X	X	X	0
01	0	0	0	0
11	0	0	0	1
10	X	X	X	X

(d) $R_2 = y_1\,TC'$

FIGURE 9.45
Maps for latch inputs

The logic diagram of the circuit is shown in Fig. 9.46. Here we use two NAND latches with two or three inputs in each gate. This implementation is according to the pattern established in Section 9.7 in conjunction with Fig. 9.38(b). The S and R input functions require six NAND gates for their implementation.

The example just presented illustrates the complexity involved in designing asynchronous sequential circuits. It was necessary to go through 10 diagrams in order to obtain the final circuit diagram. Although most digital circuits are synchronous, there are occasions when one has to deal with asynchronous behavior. The basic properties presented in this chapter are essential to a full understanding of the internal behavior of digital circuits.

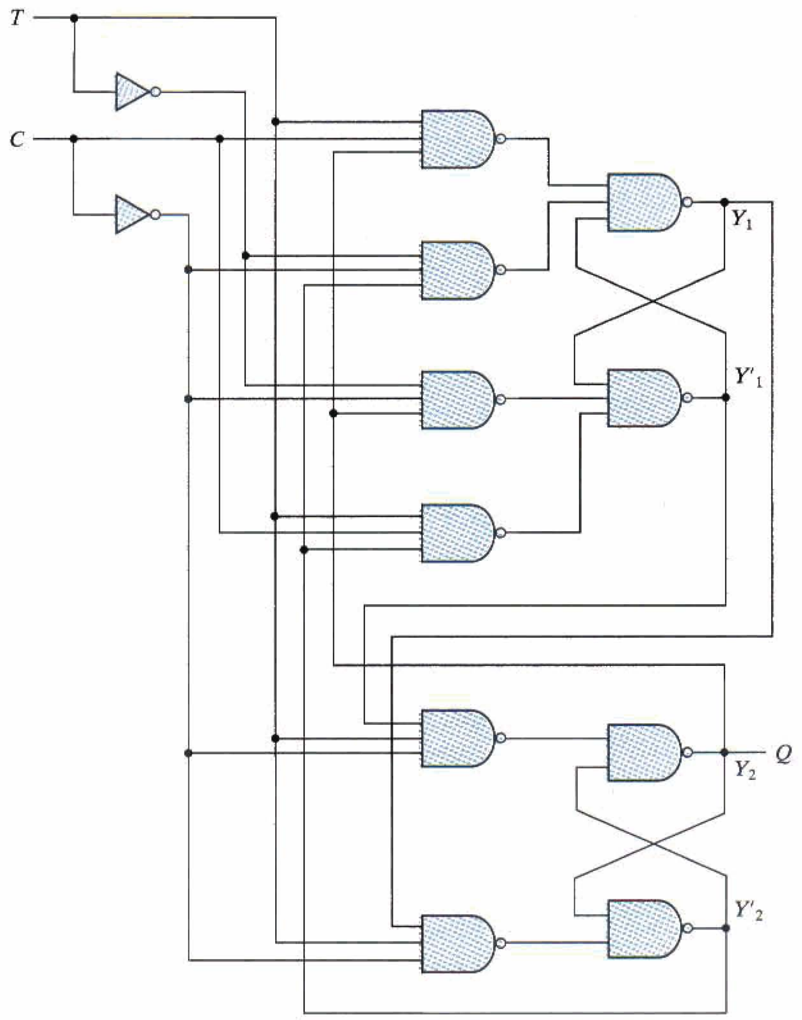


FIGURE 9.46
Logic diagram of negative-edge-triggered T flip-flop

PROBLEMS

Answers to problems marked with * appear at the end of the book.

9.1 (a) Explain the difference between asynchronous and synchronous sequential circuits.
(b) Define fundamental-mode operation.
(c) Explain the difference between stable and unstable states.
(d) What is the difference between an internal state and a total state?

9.2* Derive the transition table for the asynchronous sequential circuit shown in Fig. P9.2. Determine the sequence of internal states Y_1Y_2 for the following sequence of inputs x_1x_2: 00, 10, 11, 01, 11, 10, 00.

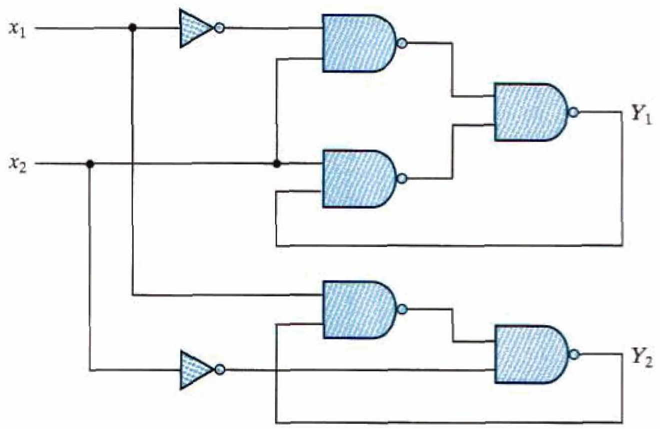


FIGURE P9.2

9.3 An asynchronous sequential circuit is described by the excitation function

$$Y = x_1x_2' + (x_1 + x_2')y$$

and the output function

$$z = y$$

(a) Draw the logic diagram of the circuit.
(b) Derive the transition table and output map.
(c) Obtain a two-state flow table.
(d)* Describe in words the behavior of the circuit.

9.4 An asynchronous sequential circuit has two internal states and one output. The two excitation functions and one output function describing the circuit are, respectively,

$$Y_1 = x_1x_2 + x_1y_2' + x_2'y_1$$
$$Y_2 = x_2 + x_1y_1'y_2 + x_1'y_1$$
$$z = x_2 + y_1$$

(a) Draw the logic diagram of the circuit.
(b) Derive the transition table and output map.
(c)* Obtain a flow table for the circuit.

9.5 Convert the flow table of Fig. P9.5 into a transition table by assigning the following binary values to the states: $a = 00$, $b = 11$, and $c = 01$.
(a) Assign values to the extra fourth state to avoid critical races.
(b) Assign outputs to the don't-care states to avoid momentary false outputs.
(c)* Derive the logic diagram of the circuit.

9.6 Investigate the transition table of Fig. P9.6, and determine all race conditions and whether they are critical or noncritical. Determine also whether there are any cycles.

9.7 Analyze the *SR* latch with control shown in Fig. 5.5. Obtain the transition table, and show that the circuit is unstable when all three inputs are equal to 1.

9.8 Modify the diagram of Fig. 5.5(a) to convert it into a *JK* type of latch by inserting two feedback connections from the outputs to the inputs. Show that the circuit is unstable when $J = K = 1$ while the control input C remains in the 1 state.

	x_1x_2 00	01	11	10
a	$\textcircled{a}$, 0	b , –	c , –	$\textcircled{a}$, 1
b	a , –	$\textcircled{b}$, 0	$\textcircled{b}$, 0	c , –
c	a , –	b , –	$\textcircled{c}$, 1	$\textcircled{c}$, 0

FIGURE P9.5

y_1y_2	x_1x_2 00	01	11	10
00	10	$\textcircled{00}$	11	10
01	$\textcircled{01}$	00	10	10
11	01	00	$\textcircled{11}$	$\textcircled{11}$
10	11	00	$\textcircled{10}$	$\textcircled{10}$

FIGURE P9.6

9.9 For the asynchronous sequential circuit shown in Fig. P9.9,

(a) derive the Boolean functions for the outputs of the two *SR* latches Y_1 and Y_2. Note that the S input of the second latch is $x_1'y_1'$.

(b) derive the transition table and output map of the circuit.

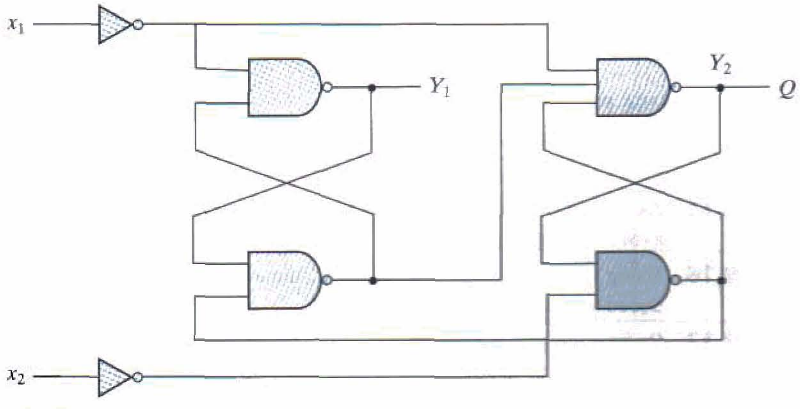


FIGURE P9.9

9.10* Implement the circuit defined in Problem 9.3 with a NOR *SR* latch. Repeat with a NAND *SR* latch.

9.11 Implement the circuit defined in Problem 9.4 with NAND *SR* latches.

9.12 Obtain a primitive flow table for a circuit with two inputs, x_1 and x_2, and two outputs, z_1 and z_2, that satisfy the following four conditions:

(a) When $x_1x_2 = 00$, the output is $z_1z_2 = 00$.
(b) When $x_1 = 1$ and x_2 changes from 0 to 1, the output is $z_1z_2 = 01$.
(c) When $x_2 = 1$ and x_1 changes from 0 to 1, the output is $z_1z_2 = 10$.
(d) Otherwise, the output does not change.

9.13* A traffic light is installed at a junction of a railroad and a road. The light is controlled by two switches in the rails placed 1 mile apart on either side of the junction. A switch is turned on when the train is over it and is turned off otherwise. The traffic light changes from green (logic 0) to red (logic 1) when the beginning of the train is 1 mile from the junction. The light changes back to green when the end of the train is 1 mile away from the junction. Assume that the length of the train is less than 2 miles.

(a) Obtain a primitive flow table for the circuit.
(b) Show that the flow table can be reduced to four rows.

9.14 It is necessary to design an asynchronous sequential circuit with two inputs, x_1 and x_2, and one output, z. Initially, both inputs and output are equal to 0. When x_1 or x_2 becomes 1, z becomes 1. When the second input also becomes 1, the output changes to 0. The output stays at 0 until the circuit goes back to the initial state.

(a) Obtain a primitive flow table for the circuit, and show that it can be reduced to the flow table shown in Fig. P9.14.
(b) Complete the design of the circuit.

	00	01	11	10
a	ⓐ, 0	ⓐ, 1	*b* , –	ⓐ, 1
b	*a* , –	ⓑ, 0	ⓑ, 0	ⓑ, 0

FIGURE P9.14

9.15 Assign output values to the don't-care states in the flow tables of Fig. P9.15 in such a way as to avoid transient output pulses.

9.16 Using the implication-table method, show that the state table listed in Table 5.7 cannot be reduced any further.

9.17 Reduce the number of states in the state table listed in Problem 5.12. Use an implication table.

9.18* Merge each of the primitive flow tables shown in Fig. P9.18. Proceed as follows:

	00	01	11	10
a	ⓐ, 0	*b* , –	– , –	*d* , –
b	*a* , –	ⓑ, 1	ⓑ, 1	*c* , –
c	*b* , –	– , –	*b* , –	ⓒ, 0
d	*c* , –	ⓓ, 1	*c* , –	ⓓ, 1

(a)

	00	01	11	10
a	ⓐ, 0	*b* , –	*b* , –	ⓐ, 0
b	*a* , –	ⓑ, 0	ⓑ, 1	*c* , –
c	*b* , –	*d* , –	ⓒ, 1	ⓒ, 1
d	ⓓ, 0	ⓓ, 1	*c* , –	*a* , –

(b)

FIGURE P9.15

	00	01	11	10
a	ⓐ, 0	*b* , –	– , –	*e* , –
b	*a* , –	ⓑ, 0	*c* , –	– , –
c	– , –	*d* , –	ⓒ, 0	*h* , –
d	*a* , –	ⓓ, 1	– , –	– , –
e	*a* , –	– , –	*f* , –	ⓔ, 0
f	– , –	*g* , –	ⓕ, 0	*h* , –
g	*a* , –	ⓖ, 0	– , –	– , –
h	*a* , –	– , –	– , –	ⓗ, 0

(a)

	00	01	11	10
a	ⓐ, 1	*f* , –	– , –	*e* , –
b	*c* , –	– , –	*j* , –	ⓑ, 0
c	ⓒ, 0	*d* , –	– , –	*b* , –
d	*c* , –	ⓓ, 0	*g* , –	– , –
e	*a* , –	– , –	*g* , –	ⓔ, 1
f	*a* , –	ⓕ, 1	*g* , –	– , –
g	– , –	*d* , –	ⓖ, 0	*k* , –
h	ⓗ, 0	*d* , –	– , –	*k* , –
j	– , –	*f* , –	ⓙ, 1	*b* , –
k	*a* , –	– , –	*j* , 1	ⓚ, 0

(b)

FIGURE P9.18

(a) Find all compatible pairs by means of an implication table.
(b) Find the maximal compatibles by means of a merger diagram.
(c) Find a minimal set of compatibles that covers all the states and is closed.

9.19 (a) Obtain a binary state assignment for the reduced flow table shown in Fig. P9.19. Avoid critical race conditions.
(b) Obtain the logic diagram of the circuit, using NAND latches and gates.

9.20* Find a critical race-free state assignment for the reduced flow table shown in Fig. P9.20.

	x_1x_2 00	01	11	10
a	$\textcircled{a}$, 0	$\textcircled{a}$, 1	b, –	d, –
b	a, –	$\textcircled{b}$, 0	$\textcircled{b}$, 0	c, –
c	a, –	–, –	d, –	$\textcircled{c}$, 0
d	a, –	a, –	$\textcircled{d}$, 1	$\textcircled{d}$, 1

FIGURE P9.19

	00	01	11	10
a	$\textcircled{a}$	d	$\textcircled{a}$	c
b	a	$\textcircled{b}$	$\textcircled{b}$	d
c	d	$\textcircled{c}$	b	$\textcircled{c}$
d	$\textcircled{d}$	$\textcircled{d}$	e	$\textcircled{d}$
e	f	c	$\textcircled{e}$	c
f	$\textcircled{f}$	b	a	$\textcircled{f}$

FIGURE P9.20

9.21 Consider the reduced flow table shown in Fig. P9.21.

(a) Obtain the transition diagram, and show that three state variables are needed for a race-free binary state assignment.

(b) Obtain the expanded flow table, using the multiple-row method of assignment as specified in Fig. 9.32(a).

	00	01	11	10
a	$\textcircled{a}$	c	$\textcircled{a}$	d
b	a	$\textcircled{b}$	c	$\textcircled{b}$
c	$\textcircled{c}$	$\textcircled{c}$	$\textcircled{c}$	d
d	$\textcircled{d}$	b	a	$\textcircled{d}$

FIGURE P9.21

9.22* Find a circuit that has no static hazards and implements the Boolean function

$$F(A, B, C, D) = \Sigma(0, 2, 6, 7, 8, 10, 12)$$

9.23* Draw the logic diagram of the product-of-sums expression

$$Y = (x_1 + x_2')(x_2 + x_3)$$

Show that there is a static 0-hazard when x_1 and x_3 are equal to 0 and x_2 goes from 0 to 1. Find a way to remove the hazard by adding one more OR gate.

9.24 The Boolean functions for the inputs of an *SR* latch are

$$S = x_1'x_2'x_3 + x_1x_2x_3$$
$$R = x_1x_2' + x_2x_3'$$

Obtain the circuit diagram, using a minimum number of NAND gates.

9.25 Complete the design of the circuit specified in Problem 9.13.

REFERENCES

1. Breeding, K. J. 1989. *Digital Design Fundamentals.* Englewood Cliffs, NJ: Prentice-Hall.
2. Friedman, A. D. 1986. *Fundamentals of Logic Design and Switching Theory.* Rockville, MD: Computer Science Press.

3. HILL, F. J., and G. R. PETERSON. 1981. *Introduction to Switching Theory and Logical Design*, 3d ed. New York: John Wiley.

4. KOHAVI, Z. 1978. *Switching and Automata Theory*, 2d ed. New York: McGraw-Hill.

5. MCCLUSKEY, E. J. 1986. *Logic Design Principles.* Englewood Cliffs, NJ: Prentice-Hall.

6. NELSON, V. P., H. T. NAGLE, J. D. IRWIN, and B. D. CARROLL. 1995. *Digital Logic Circuits Analysis and Design.* Upper Saddle River, NJ: Prentice Hall.

7. UNGER, S. H. 1969. *Asynchronous Sequential Switching Circuits.* New York: John Wiley.