7

Sequential Logic

This chapter will analyze and synthesize both synchronous and asynchronous sequential machines. Analysis is achieved by applying certain techniques to existing machines to observe their behavior. Synchronization for synchronous machines is accomplished by *clock* pulses, where a clock is a device that produces pulses at regular intervals.

Before proceeding with the analysis of synchronous sequential machines, the concept of a state will be defined. A *state* is a set of values that is measured at different locations within the machine. The values correspond to the set or reset condition of storage elements. The storage elements may be clocked flip-flops or *SR* latches. A storage element can store one bit of information, and therefore, has two stable states: 0 and 1. A machine is stable in a particular state when no input signals are changing; that is, the input variables and the clock, if applicable, are stable.

If there are p storage elements, then the state of the machine is the p-tuple of the storage element states and consists of 0s and 1s, where each bit is the value of a particular storage element cell. Thus, there are 2^p possible states, some of which may be unused. For example, if there are four storage elements, then possible states (also called state codes) are: 0101, 1001, 1100, etc.

A sequential logic circuit consists of combinational logic and storage elements. The circuit is called sequential because operations are performed in sequence. A sequential circuit can assume a finite number of internal states and can, therefore, be regarded as a *finite-state machine* (or simply a state machine) with a finite number of

inputs and a finite number of outputs. In general, there are two types of sequential circuits: synchronous sequential machines (clocked) and asynchronous sequential machines (not clocked).

7.1 Analysis of Synchronous Sequential Machines

A *synchronous sequential machine* is a machine whose present outputs are a function of the present state only or the present state and present inputs. The present states are determined by the sequence of previous inputs (the input history) and the previous states. The sequence of previous inputs is the order in which the inputs occurred.

A requirement of a synchronous sequential machine is that state changes occur only when the machine is clocked, either on the positive or negative transition of the clock. Thus, input changes do not affect the present state of the machine until the occurrence of the next active clock transition. Clock pulses are synchronization signals provided by a clock, usually an astable multivibrator, which generates a periodic series of pulses. The storage elements are affected only at the positive or negative transition of the clock pulses.

7.1.1 Machine Alphabets

A sequential machine is a mathematical model of a sequential logic circuit and consists of the following three alphabets:

Input alphabet There is a set of external inputs consisting of n binary variables

$$\{x_1, x_2, \cdots, x_n\}$$

where $x_i = 0$ or 1 for $1 \leq i \leq n$. The inputs generate 2^n input combinations of ordered n-tuples. Each combination of the input variables is referred to as a vector (or symbol) of the input alphabet. Thus, the input alphabet X is the set of 2^n input symbols as shown in Equation 7.1.

$$X = \{X_0, X_1, X_2, \cdots, X_{2^n-2}, X_{2^n-1}\} \tag{7.1}$$

For example, if a machine has three input variables, x_1, x_2, and x_3, then the input alphabet consists of eight symbols X_0 through X_7, as shown below.

$$X = \{000, 001, 010, \cdots, 110, 111\}$$

where

$$\begin{aligned} X_0 &= x_1x_2x_3 = 000 \\ X_1 &= x_1x_2x_3 = 001 \\ X_2 &= x_1x_2x_3 = 010 \\ &\cdots \\ X_7 &= x_1x_2x_3 = 111 \end{aligned}$$

Some vectors of the input alphabet may not be used.

State alphabet There is a set of present internal state variables consisting of p synchronous storage elements

$$\{y_1, y_2, \cdots, y_p\}$$

where $y_i = 0$ or 1 for $1 \le i \le p$. The storage elements generate 2^p possible states, although the machine may not use all states.

Each combination of the storage element values is referred to as a state (or *state code*) of the state alphabet. Thus, the state alphabet Y is the set of 2^p states, as shown in Equation 7.2.

$$Y = \{Y_0, Y_1, Y_2, \cdots, Y_{2^p-2}, Y_{2^p-1}\} \tag{7.2}$$

For example, if a machine has four storage elements y_1, y_2, y_3, and y_4, then the state alphabet consists of 16 states Y_0 through Y_{15}:

$$Y = \{0000, 0001, 0010, \cdots, 1110, 1111\}$$

where

$$\begin{aligned} Y_0 &= y_1y_2y_3y_4 = 0000 \\ Y_1 &= y_1y_2y_3y_4 = 0001 \\ Y_2 &= y_1y_2y_3y_4 = 0010 \\ &\cdots \\ Y_{15} &= y_1y_2y_3y_4 = 1111 \end{aligned}$$

A 0 state indicates a reset storage element and a 1 indicates a set storage element. The state of the machine, therefore, is the p-tuple of the storage element states.

Output alphabet There is a set of outputs consisting of m binary variables

$$\{z_1, z_2, \cdots, z_m\}$$

where $z_i = 0$ or 1 for $1 \le i \le m$. The outputs generate 2^m output combinations of ordered m-tuples. Each combination of the output variables is referred to as a vector (or symbol) of the output alphabet. Thus, the output alphabet Z is the set of 2^m output vectors as shown in Equation 7.3.

$$Z = \{Z_0, Z_1, Z_2, \cdots, Z_{2^m-2}, Z_{2^m-1}\} \tag{7.3}$$

For example, if a machine has two outputs z_1 and z_2, then the output alphabet consists of four symbols Z_0 through Z_3:

$$Z = \{00, 01, 10, 11\}$$

where

$$Z_0 = z_1 z_2 = 00$$
$$Z_1 = z_1 z_2 = 01$$
$$Z_2 = z_1 z_2 = 10$$
$$Z_3 = z_1 z_2 = 11$$

Some symbols of the output alphabet may not be used.

The present external inputs $x_1, x_2, \cdots, x_n$ and the present values of the state variables $y_1, y_2, \cdots, y_p$, which were obtained at clock (t), combine to produce the present outputs $z_1, z_2, \cdots, z_m$ and also the next state of the machine at the occurrence of the next clock ($t + 1$).

The following notation will be used to represent the specified vectors and states:

$X_{i(t)}$ is the present input vector, where $X_{i(t)} \in X$.
$Y_{j(t)}$ is the present state, where $Y_{j(t)} \in Y$.
$Y_{k(t+1)}$ is the next state, where $Y_{k(t+1)} \in Y$.
$Z_{r(t)}$ is the present output vector, where $Z_{r(t)} \in Z$.

The *Cartesian product* of two sets is defined as follows: For any two sets S and T, the Cartesian product of S and T is written as $S \times T$ and is the set of all *ordered pairs* of S and T, where the first member of the ordered pair is an element of S and the second member is an element of T. Thus, the general classification of a synchronous sequential machine M can be defined as the 5-tuple shown in Equation 7.4.

$$M = (X, Y, Z, \delta, \lambda) \tag{7.4}$$

where

1. X is a nonempty finite set of inputs.
2. Y is a nonempty finite set of states.
3. Z is a nonempty finite set of outputs.
4. δ is the next-state function which maps the Cartesian product of $X \times Y$ into Y.
5. λ is the output function which maps the Cartesian product of $X \times Y$ into Z.

A synchronous sequential machine is deterministic and can now be defined in terms of the machine alphabets and the next-state function δ. The next state $Y_{k(t+1)}$ is uniquely determined by the present inputs $X_{i(t)}$ and the present state $Y_{j(t)}$. Thus, the next state can be expressed as shown in Equation 7.5 as a function of the δ next-state logic.

$$Y_{k(t+1)} = \delta(X_{i(t)}, Y_{j(t)}) \tag{7.5}$$

7.1.2 Storage Elements

This section will review the operating characteristics of the *SR* latch, *D* flip-flop, *JK* flip-flop, and *T* flip-flop. A latch is a level-sensitive storage element in which a change to an input signal affects the output directly without recourse to a clock input. The set (*s*) and reset (*r*) inputs may be active high or active low. The *D*, *JK*, and *T* flip-flops, however, are triggered on the application of a clock signal and are positive- or negative-edge-triggered devices.

***SR* latch** The *SR* latch is usually implemented using either NAND gates or NOR gates, as shown in Figure 7.1(a) and Figure 7.1(b), respectively. When a negative pulse (or level) is applied to the *–set* input of the NAND gate latch, the output $+y_1$ becomes active at a high voltage level. This high level is also connected to the input of NAND gate 2. Since the set and reset inputs cannot both be active simultaneously, the reset input is at a high level, providing a low voltage level on the output of gate 2 which is fed back to the input of gate 1. The negative feedback, therefore, provides a second set input to the latch. The original set pulse can now be removed and the latch will remain set. Concurrent set and reset inputs represent an invalid condition, since both outputs will be at the same voltage level; that is, outputs $+y_1$ and $-y_1$ will both be at the more positive voltage level — an invalid state for a bistable device with complementary outputs.

If the NAND gate latch is set, then a low voltage level on the *–reset* input will cause the output of gate 2 to change to a high level which is fed back to gate 1. Since both inputs to gate 1 are now at a high level, the $+y_1$ and $-y_1$ outputs will change to a low and high level, respectively, which is the reset state for the latch. The *characteristic table* for an *SR* latch is shown in Table 7.1, where $Y_{j(t)}$ and $Y_{k(t+1)}$ are the present state and next state of the latch, respectively. The excitation equation is shown in Equation 7.6.

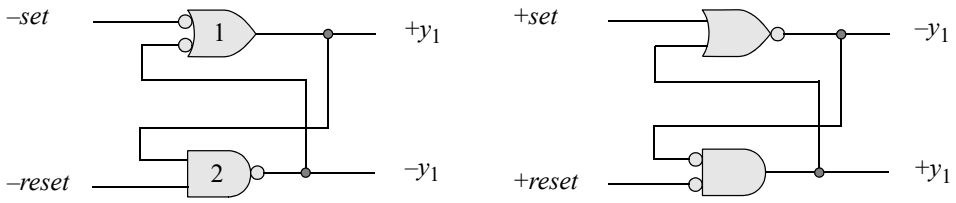


Figure 7.1 *SR* latches: (a) using NAND gates and (b) using NOR gates.

Table 7.1 *SR* Latch Characteristic Table

Data Inputs *S R*	Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$
0 0	0	0
0 0	1	1
0 1	0	0
0 1	1	0
1 0	0	1
1 0	1	1
1 1	0	Invalid
1 1	1	Invalid

$$Y_{k(t+1)} = S + R' Y_{j(t)} \tag{7.6}$$

***D* flip-flop** A *D* flip-flop is an edge-triggered device with one data input and one clock input. Figure 7.2 illustrates a positive-edge-triggered *D* flip-flop. The $+y_1$ output will assume the state of the *D* input at the next positive clock transition. After the occurrence of the clock's positive edge, any change to the *D* input will not affect the output until the next active clock transition. The characteristic table for a *D* flip-flop is shown in Table 7.2 and the corresponding excitation equation in Equation 7.7.

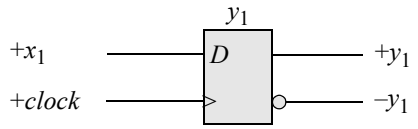


Figure 7.2 A positive-edge-triggered *D* flip-flop.

Table 7.2 *D* Flip-Flop Characteristic Table

Data Input D	Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$
0	0	0
0	1	0
1	0	1
1	1	1

$$Y_{k(t+1)} = D \tag{7.7}$$

***JK* flip-flop** The *JK* flip-flop is also an edge-triggered storage device. The active clock transition can be either the positive or negative edge. Figure 7.3 illustrates a negative-edge-triggered *JK* flip-flop. The functional characteristics of the *JK* data inputs are defined in Table 7.3. The characteristic table of Table 7.4 lists the next state $Y_{k(t+1)}$ for each combination of *J*, *K*, and the present state $Y_{j(t)}$ based on the functional characteristics of *J* and *K*. Table 7.5 shows an excitation table in which a particular state transition predicates a set of values for *J* and *K*. This table is especially useful in the synthesis of synchronous sequential machines.

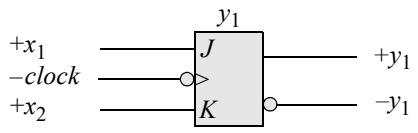


Figure 7.3 A negative-edge-triggered *JK* flip-flop.

Table 7.3 *JK* Functional Characteristic Table

J K	Function
0 0	No change
0 1	Reset
1 0	Set
1 1	Toggle

Table 7.4 *JK* Flip-Flop Characteristic Table

Data Inputs *J K*	Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$
0 0	0	0
0 0	1	1
0 1	0	0
0 1	1	0
1 0	0	1
1 0	1	1
1 1	0	1
1 1	1	0

Table 7.5 Excitation Table for a *JK* Flip-Flop

Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$	Data Inputs *J K*	
0	0	0 –	A dash (–) indicates a "don't care" condition
0	1	1 –	
1	0	– 1	
1	1	– 0	

The excitation equation for a *JK* flip-flop is derived from Table 7.5 and is shown in Equation 7.8.

$$Y_{k(t+1)} = Y_{j(t)}' J + Y_{j(t)} K' \tag{7.8}$$

***T* flip-flop** The toggle (*T*) flip-flop is shown in Figure 7.4 as a positive-edge-triggered device. When the *T* input is at a logic 1 level, the flip-flop will toggle (change state) at the next active clock transition. The characteristic table is shown in Table 7.6 and the excitation table in Table 7.7. The corresponding excitation equation is shown in Equation 7.9.

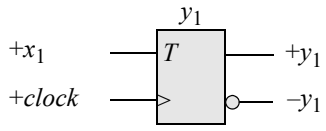


Figure 7.4 A *T* flip-flop.

Table 7.6 *T* Flip-Flop Characteristic Table

Data Input T	Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$
0	0	0
0	1	1
1	0	1
1	1	0

Table 7.7 *T* Flip-Flop Excitation Table

Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$	Data Input T
0	0	0
0	1	1
1	0	1
1	1	0

$$Y_{k(t+1)} = Y_{j(t)}' T + Y_{j(t)} T' \tag{7.9}$$

7.1.3 Classes of Sequential Machines

This section will present classes of sequential machines. With the exception of the last class of machines (asynchronous sequential machines), all other machines are synchronous, where changes to states and outputs occur only in synchronization with clock pulses. The storage elements in synchronous sequential machines are bistable multivibrators such as, *D* flip-flops and *JK* flip-flops. The storage elements in asynchronous sequential machines are *SR* latches. Some of the machines in the following sections contain no data inputs; however, all of the machines produce outputs. Each machine consists of at least one of the following three logical units:

1. Input combinational logic
2. Synchronous storage elements
3. Output combinational logic

If input combinational logic is used, then the δ next-state function maps the present state or the present state and the present inputs into the state alphabet. That is, the δ mapping generates the input equations for the storage elements. If output combinational logic is used, then the λ function maps the present state or the present state and the present inputs into the output alphabet. Thus, the λ mapping generates outputs from the machine.

Registers In their simplest form, registers contain only storage elements. They may, however, consist of input logic and output logic as shown in Figure 7.5. If input logic is specified in the design, then the next-state function δ is a function of the input alphabet *X*, and maps *X* into *Y*, as shown in Equation 7.10. Also, δ is a function of the present input vector $X_{i(t)}$, and maps $X_{i(t)}$ into the next state $Y_{k(t+1)}$, as shown in Equation 7.11. Equation 7.12 defines the next state.

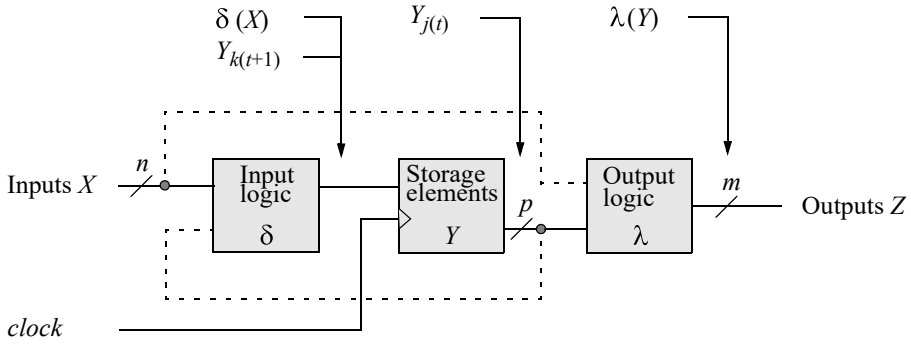


Figure 7.5 Register general block diagram.

$$\delta(X) : X \rightarrow Y \tag{7.10}$$

$$\delta(X_{i(t)}) : X_{i(t)} \rightarrow Y_{k(t+1)} \tag{7.11}$$

$$Y_{k(t+1)} = \delta(X_{i(t)}) \tag{7.12}$$

When the output function λ is implemented in a register, then λ is a function of the state alphabet *Y* and maps *Y* into the output alphabet *Z*, as shown in Equation 7.13. The present output vector $Z_{r(t)}$ is shown in Equation 7.14.

$$\lambda(Y) : Y \rightarrow Z \tag{7.13}$$

$$Z_{r(t)} = \lambda(Y_{j(t)}) \tag{7.14}$$

Registers can be Moore- or Mealy-type machines depending on the configuration of the output logic. Moore and Mealy machines are presented in Section 7.2.4 and Section 7.2.5, respectively. Registers are used to store binary information in a digital

system. The information can consist of a single bit or of several bits which define instructions or operands. An *n*-bit register consists of *n* storage elements and can store *n* bits of binary information, one bit in each storage element.

Data can be loaded into a register either in parallel or in serial format. Parallel loading is faster, because all storage elements receive new information during one clock pulse. During a serial load operation, each storage element receives new data from the element to its immediate left or right, depending on the direction of loading (shifting). The first storage element receives its data from an external source.

A typical 8-bit parallel-in, parallel-out register is shown in Figure 7.6, which consists of the next-state function δ, storage elements $Y[7:0] = y_7, y_6, y_5, \ldots, y_0$, and the output function λ. Both the δ and λ mappings use combinational logic consisting of eight AND gates for each of the two functions. The clock pulse sets the register to the contents of the input vector $X[7:0] = x_7, x_6, x_5, \cdots, x_0$.

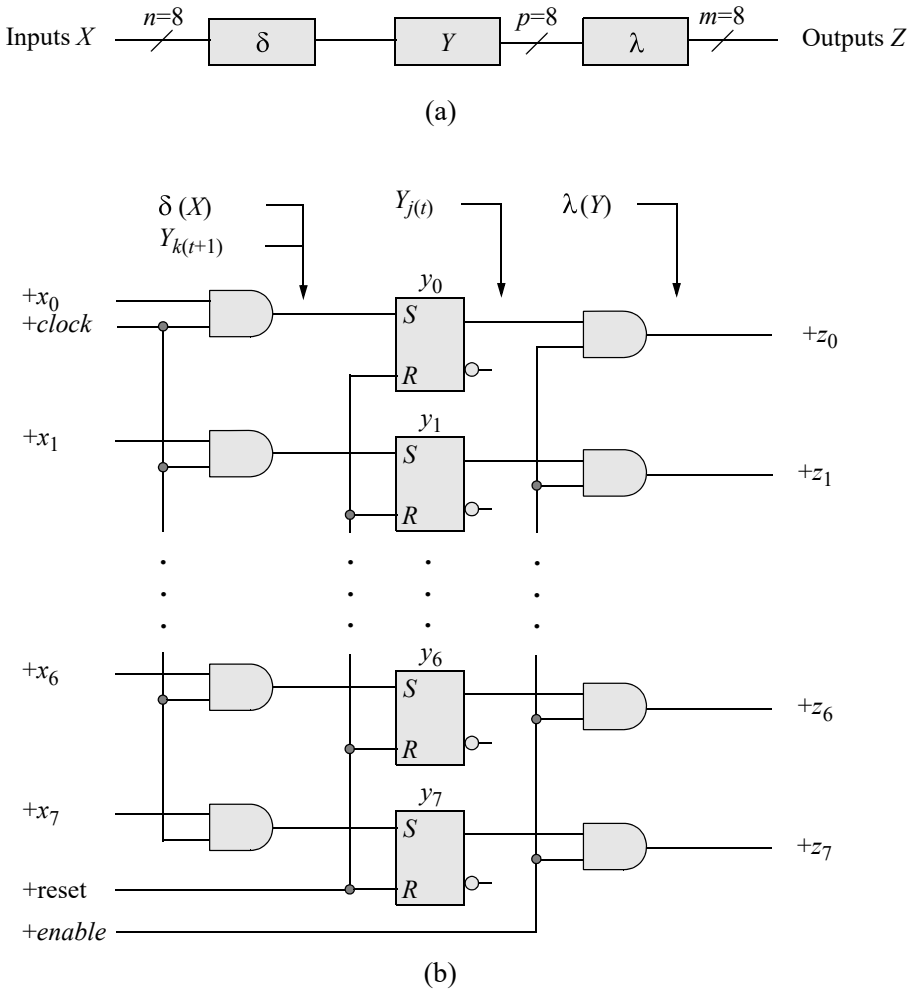


Figure 7.6 Register implemented in a parallel-in, parallel-out configuration: (a) block diagram and (b) logic diagram using latches for the storage elements.

Because the storage elements are latches, the input data must not change while the *clock* is positive. An *enable* signal is one of the inputs to the λ output function combinational logic and transforms the state of the register to the output vector $Z[7:0] = z_7, z_6, z_5, \cdots, z_0$.

Figure 7.7 illustrates an 8-bit serial-in, serial-out shift register implemented with *D* flip-flops. There is no logic shown for either the δ next-state function or the λ output function, although the input logic is implied by the *D* input of the flip-flop. The shift register is reset initially, then information is shifted into stage_0 one bit per clock pulse. The positive transition of the clock signal causes the following shift sequence to occur:

$x_0 \rightarrow y_0$
$y_j \rightarrow y_{j+1}$
y_7 is lost

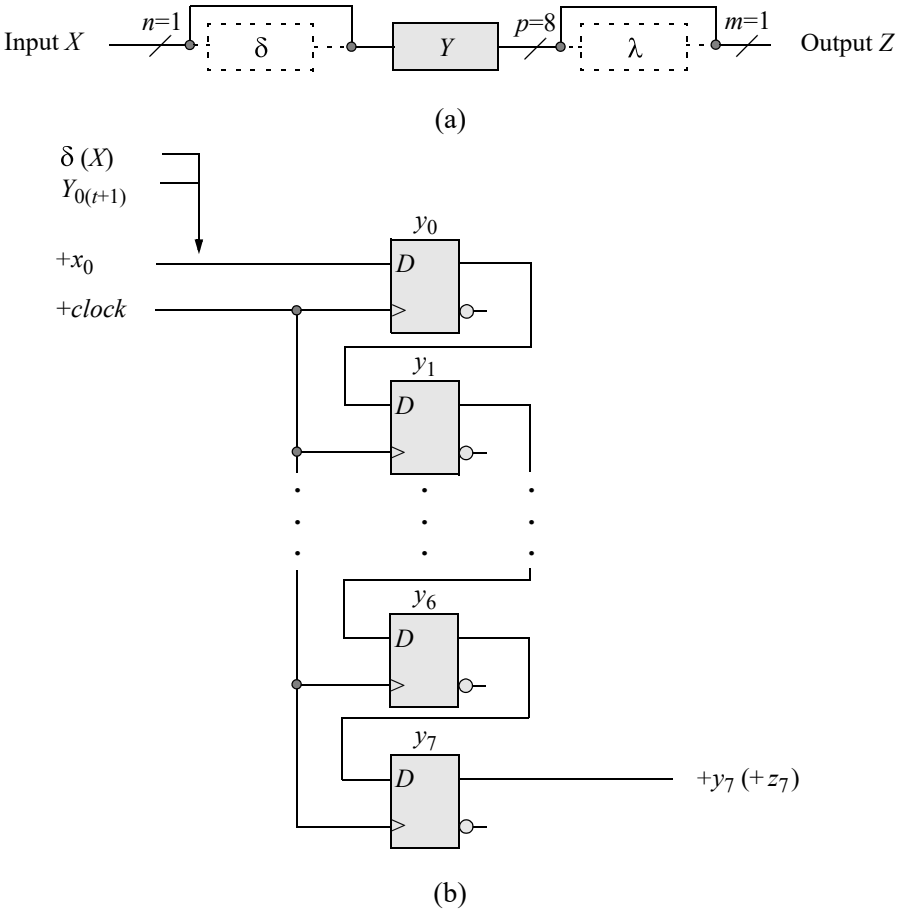


Figure 7.7 Register implemented in a serial-in, serial-out configuration: (a) block diagram and (b) logic diagram using *D* flip-flops for the storage elements.

Counters This section will present some general comments regarding counters. The details of synthesis will be covered in Section 7.2.3. Counters are one of the simplest types of sequential machines, requiring only one input in most cases. The single input is a clock pulse. Although most counters can be categorized as a type of Moore machine, counters are of sufficient importance to warrant a separate classification. A *counter* is constructed from one or more flip-flops that change state in a prescribed sequence upon the application of a series of clock pulses. The sequence of states in a counter may generate a binary count, a binary-coded decimal (BCD) count, or any other counting sequence. The counting sequence does not have to be sequential.

Counters are used for counting the number of occurrences of an event and for general timing sequences. A block diagram of a synchronous counter is shown in Figure 7.8. The diagram depicts a typical counter consisting of combinational input logic for the δ next-state function, storage elements, and combinational output logic for the λ output function. Input logic is required when an initial count must be loaded into the counter. The input logic then differentiates between a clock pulse that is used for loading and a clock pulse that is used for counting. Not all counters are implemented with input and output logic, however. Some counters contain only storage elements that are connected in cascade.

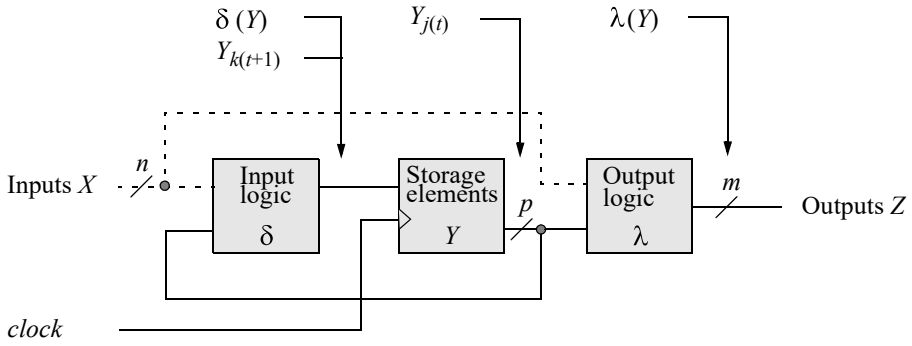
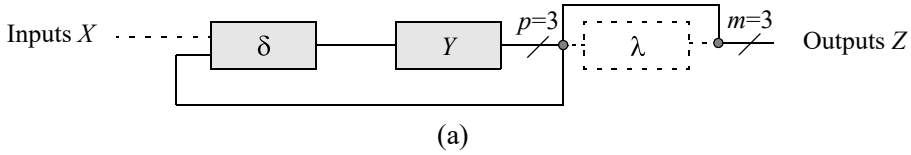


Figure 7.8 Counter block diagram.

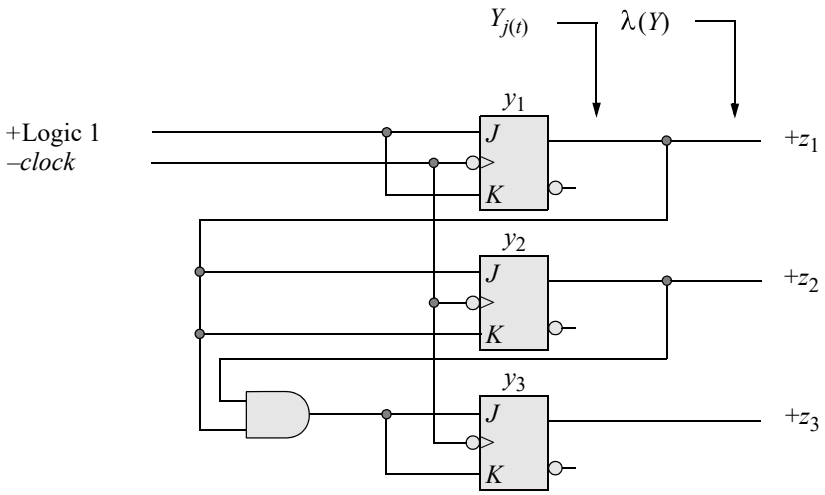
Counters can be designed as count-up counters, in which the counting sequence increases numerically, or as count-down counters, in which the counting sequence decreases numerically. A counter may also be designed as both a count-up counter and a count-down counter, the mode of operation being controlled by a separate input.

Figure 7.9 illustrates a synchronous modulo-8 count-up binary counter. Synchronous counters are faster than asynchronous counters because the clock pulse is transmitted to all stages simultaneously. The counter of Figure 7.9 is implemented with *JK* flip-flops as the storage elements, where each flip-flop is wired in toggle mode; that is, $JK = 11$. The clock inputs to the storage elements are negative edge-triggered inputs; thus, the counter is incremented on each negative transition of the clock pulse, as

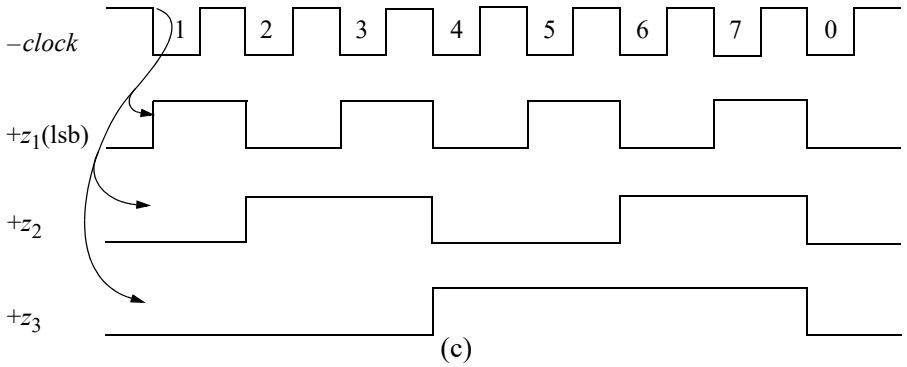
shown in Figure 7.9(c). The counter is designed as a modulo-8 counter; therefore, after being reset initially, the counting sequence is 000, 001, 010, . . . 110, 111, 000, ...



(a)



(b)



(c)

Figure 7.9 Counter implemented as a modulo-8 count-up binary counter using *JK* flip-flops: (a) block diagram; (b) logic diagram; and (c) timing diagram.

Moore machines Moore machines are synchronous sequential machines in which the output function λ produces an output vector Z_r which is determined by the present state only, and is not a function of the present inputs. The general configuration of a Moore machine is shown in Figure 7.10. The next-state function δ is an $(n + p)$-input, p-output switching function. The output function λ is a p-input, m-output switching function. If a Moore machine has no data input, then it is referred to as an *autonomous* machine. Autonomous circuits are independent of the inputs. The clock signal is not considered as a data input. An autonomous Moore machine is an important class of synchronous sequential machines, the most common application being a counter, as discussed previously. A Moore machine may be synchronous or asynchronous; however, this section pertains to synchronous organizations only.

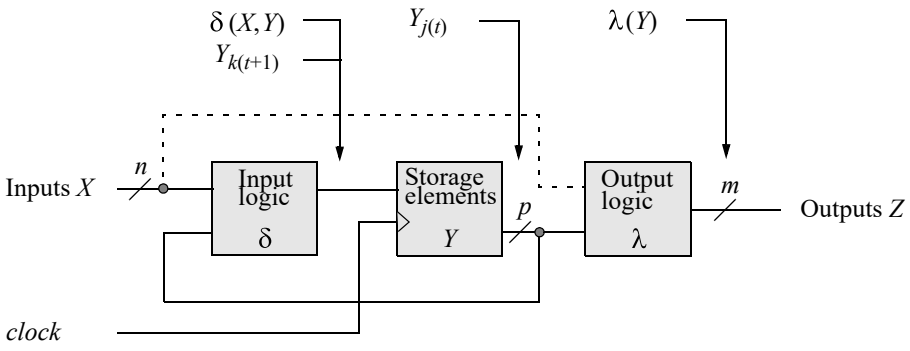


Figure 7.10 Moore synchronous sequential machine in which the outputs are a function of the present state only.

A Moore machine is a 5-tuple and can be defined as shown in Equation 7.15,

$$M = (X, Y, Z, \delta, \lambda) \tag{7.15}$$

where

1. X is a nonempty finite set of inputs such that,
   $X = \{X_0, X_1, X_2, \cdots, X_2{}^n{}_{-2}, X_2{}^n{}_{-1}\}$
2. Y is a nonempty finite set of states such that,
   $Y = \{Y_0, Y_1, Y_2, \cdots, Y_2{}^p{}_{-2}, Y_2{}^p{}_{-1}\}$
3. Z is a nonempty finite set of outputs such that,
   $Z = \{Z_0, Z_1, Z_2, \cdots, Z_2{}^m{}_{-2}, Z_2{}^m{}_{-1}\}$
4. $\delta(X, Y) : X \times Y \rightarrow Y$
5. $\lambda(Y) : Y \rightarrow Z$

A simple Moore machine is shown in Figure 7.11 in which the input alphabet is $X = \{X_0, X_1\}$, the state alphabet is $Y = \{Y_0, Y_1, Y_2, Y_3\}$, and the output alphabet is $Z = \{Z_0, Z_1\}$.

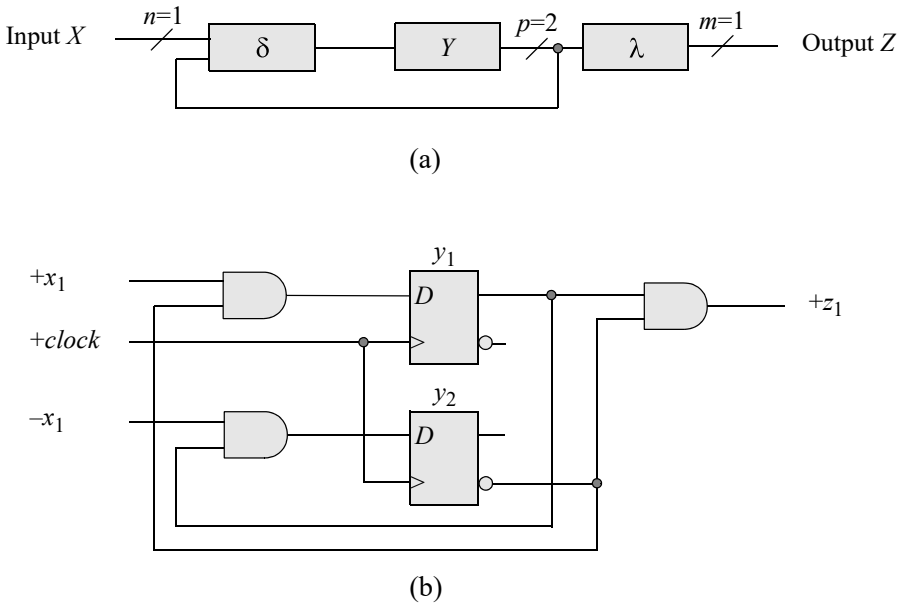


Figure 7.11 Moore machine: (a) block diagram and (b) logic diagram.

Mealy machines Mealy machines are synchronous sequential machines in which the output function λ produces an output vector $Z_{r(t)}$ which is determined by both the present input vector $X_{i(t)}$ and the present state of the machine $Y_{j(t)}$. The general configuration of a Mealy machine is shown in Figure 7.12.

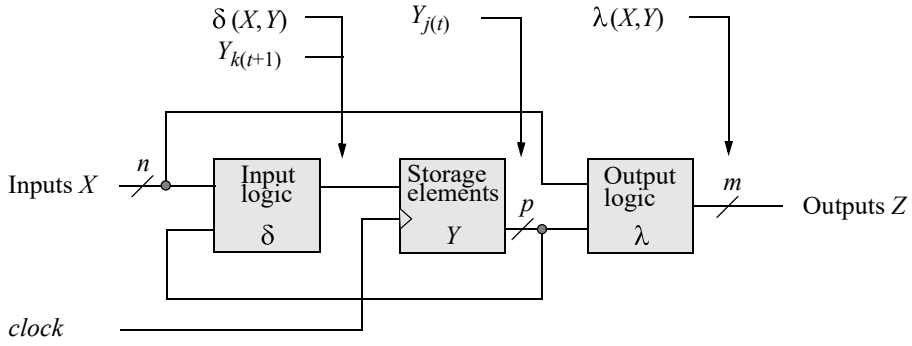


Figure 7.12 Mealy machine in which the outputs are a function of both the present state and the present inputs.

The next-state function δ is an $(n + p)$-input, p-output switching function. The output function λ is an $(n + p)$-input, m-output switching function. A Mealy machine is not an autonomous machine because the outputs are a function of the present state and the input signals. A Mealy machine may be synchronous or asynchronous; however, this section pertains to synchronous organizations only.

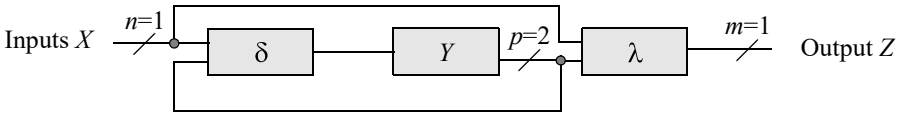
A Mealy machine is a 5-tuple and can be formally defined as shown in Equation 7.16,

$$M = (X, Y, Z, \delta, \lambda) \tag{7.16}$$

where

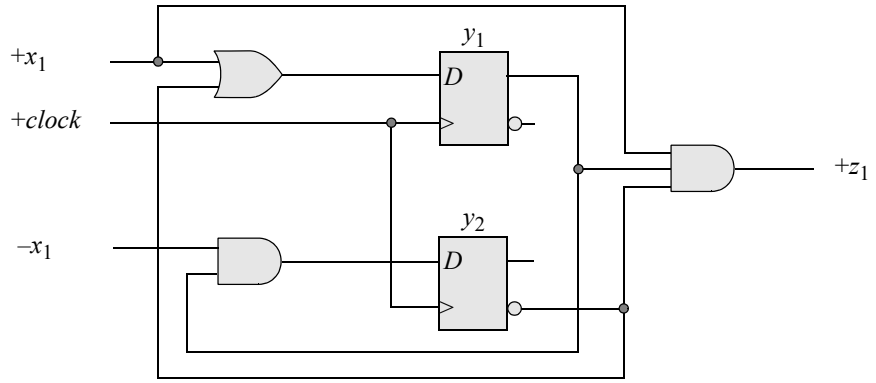
1. X is a nonempty finite set of inputs such that,
   $X = \{X_0, X_1, X_2, \cdots, X_2{}^n{}_{-2}, X_2{}^n{}_{-1}\}$
2. Y is a nonempty finite set of states such that,
   $Y = \{Y_0, Y_1, Y_2, \cdots, Y_2{}^p{}_{-2}, Y_2{}^p{}_{-1}\}$
3. Z is a nonempty finite set of outputs such that,
   $Z = \{Z_0, Z_1, Z_2, \cdots, Z_2{}^m{}_{-2}, Z_2{}^m{}_{-1}\}$
4. $\delta(X, Y) : X \times Y \rightarrow Y$
5. $\lambda(X, Y) : Y \rightarrow Z$

A simple Mealy machine is shown in Figure 7.13, in which the input alphabet is $X = \{X_0, X_1\}$, the state aphabet is $Y = \{Y_0, Y_1, Y_2, Y_3\}$, and the output alphabet is $Z = \{Z_0, Z_1\}$.



(a)

Figure 7.13 Mealy machine: (a) block diagram and (b) logic diagram.



(b)

Figure 7.13 (Continued)

7.1.4 Methods of Analysis

Analysis is the methodical investigation of a problem and the decomposition of the problem into smaller related units for further detailed study. The problem, in this case, is a synchronous sequential machine which will be studied using various analytical techniques. These techniques include a next-state table, a present-state map, next-state maps, input maps and their associated input equations, output maps and equations, a timing diagram, and a state diagram.

Understanding a synchronous sequential machine through analysis is ideal preparation for later synthesizing (or designing) sequential machines. Different machines will be presented in this section and, as each machine is analyzed, the various units of the machine will be identified with the corresponding equations. First, the techniques, or methods, that are used in the analysis procedure will be defined.

Next-state table The next-state table is a convenient method of describing the operation of a machine in tabular form. The table lists all possible present states and input values, together with the next state and present output. Table 7.8 shows a typical next-state table for a Moore machine with two *D* flip-flops. All combinations of two variables are listed under the present-state heading. The machine is assumed to be reset initially, which is represented by the first two rows, where $y_1y_2 = 00$. In this example, each pair of rows in the table corresponds to a state in the machine. For example, the first and second rows represent state $y_1y_2 = 00$. The state can also be expressed as a state name, such as "*a*."

Table 7.8 Typical Next-State Table for a Moore Synchronous Sequential Machine Using *D* Flip-Flops

State Name	Present State y_1y_2	Input x_1	Flip-Flop Inputs Dy_1 Dy_2	Next State y_1y_2	Output z_1
a	0 0	0	0 0	0 0	0
	0 0	1	0 1	0 1	0
b	0 1	0	0 1	0 1	0
	0 1	1	1 0	1 0	0
c	1 0	0	1 0	1 0	1
	1 0	1	1 1	1 1	1
d	1 1	0	1 1	1 1	0
	1 1	1	0 0	0 0	0

The entries in Table 7.8 denote the state transitions and output that correspond to a given sequence of inputs. In the first row, the present state is *a* ($y_1y_2 = 00$). If $x_1 = 0$, the machine remains in state *a* and the present output $z_1 = 0$. If, however, $x_1 = 1$ in state *a*, then the machine moves to state *b* ($y_1y_2 = 01$) at the next assertion of the clock and state *b* becomes the new present state. No indication is given in the next-state table as to the active assertion of the clock; the machine may be clocked on either the positive or negative clock transition.

Since *D* flip-flops are used in the synchronous sequential machine of Table 7.8, the next state is identical to the values of Dy_1 and Dy_2 after the active clock transition. Output z_1 is active when the present state is $y_1y_2 = 10$, regardless of the value of x_1. Since the output is a function of the present state only, the next-state table represents a Moore machine.

Next-state map The next-state map is simply the next-state table represented in Karnaugh map form as shown in Figure 7.14 for the machine of Table 7.8. The information that is specified in the next-state map can also be obtained from the logic diagram, if one is given. Since there are two flip-flops in the implementation of this machine, there are two next-state maps — one for each flip-flop. The map contains eight squares to accommodate the two flip-flops and input x_1.

For a *D* flip-flop, the next state corresponds to the present value of the *D* input. Referring to Table 7.8 and Figure 7.14, if the present state is $y_1y_2 = 00$ and $x_1 = 0$, then $Dy_1 = 0$ and the next state for y_1 will be 0. For a present state of $y_1y_2 = 00$ and $x_1 = 1$, then $Dy_1 = 0$ and the next state for $y_1 = 0$. Similarly, in state *c*, where $y_1y_2 = 10$, $Dy_1 = 1$ regardless of the value of x_1; therefore, the next state for y_1 is 1. Using

this procedure, the information contained in the next-state table is transferred to the next-state maps.

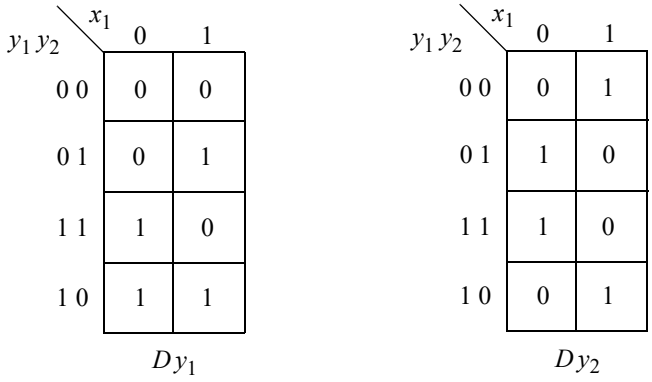


Figure 7.14 Next-state maps for the Moore machine of Table 7.8. These are the same as the input maps when *D* flip-flops are used.

Input map The input map represents the δ next-state function from which equations are generated for the data input logic of the flip-flop. Because *D* flip-flops are used in the implementation of the machine shown in Table 7.8, the next-state maps also specify the input maps; thus, the two types of maps are identical. In Figure 7.14, the input maps for y_1 and y_2 yield the input equations as shown in Equation 7.17, from which the δ next-state logic can be implemented directly.

$$\begin{aligned} Dy_1 &= x_1'y_1 + y_1y_2' + x_1y_1'y_2 \\ Dy_2 &= x_1'y_2 + x_1y_2' \end{aligned} \tag{7.17}$$

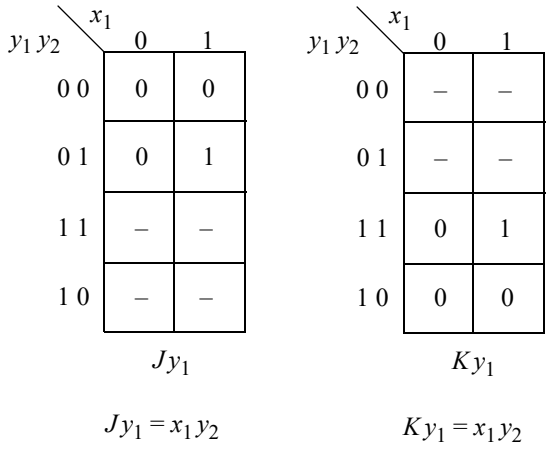
Using the same next-state table as Table 7.8, but replacing the *D* flip-flops with *JK* flip-flops yields the next-state table of Table 7.9. The next-state maps are identical to those shown in Figure 7.14. However, the input maps change due to the characteristics of a *JK* flip-flop, which are reproduced below.

State Transition			Values of	
From		To	*J*	*K*
0	→	0	0	–
0	→	1	1	–
1	→	0	–	1
1	→	1	–	0

Table 7.9 Typical Next-State Table for a Moore Machine Using *JK* Flip-Flops

State Name	Present State y_1y_2	Input x_1	Flip-Flop Inputs Jy_1	Ky_1	Jy_2	Ky_2	Next State y_1y_2	Output z_1
a	0 0	0	0	–	0	–	0 0	0
	0 0	1	0	–	1	–	0 1	0
b	0 1	0	0	–	–	0	0 1	0
	0 1	1	1	–	–	1	1 0	0
c	1 0	0	–	0	0	–	1 0	1
	1 0	1	–	0	1	–	1 1	1
d	1 1	0	–	0	–	0	1 1	0
	1 1	1	–	1	–	1	0 0	0

The input maps for the machine of Table 7.9 are shown in Figure 7.15. There are two maps for each flip-flop, one for the *J* input and one for the *K* input. The maps are constructed directly from Table 7.9. Figure 7.15 also specifies the *JK* input equations, which define the logic for the δ next-state function.



(Continued on next page)

Figure 7.15 Input maps for the Moore machine of Table 7.9.

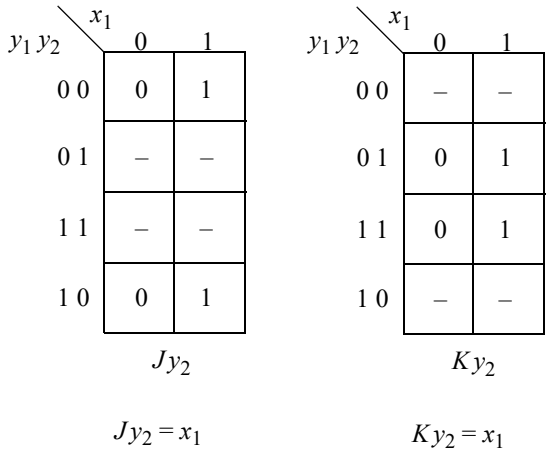


Figure 7.15 (Continued)

Output map The output map represents the λ output function from which equations are generated for the output logic of the machine. The output map for the machines of Table 7.8 and Table 7.9 is shown in Figure 7.16. Output z_1 is asserted in state y_1y_2 = 10 regardless of the value of x_1, which corresponds to the definition of a Moore machine.

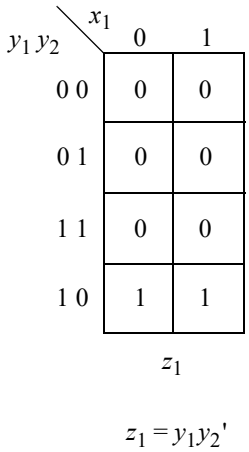


Figure 7.16 Output map for the Moore machine of Table 7.8 and Table 7.9.

Timing diagram Another useful tool for analyzing synchronous sequential machines is a timing diagram (or waveforms) which illustrates the voltage levels of the inputs, storage elements, and outputs as the machine progresses through a sequence of states. Using the Moore machine of Table 7.8 and assuming an initial reset state of $y_1y_2 = 00$, the timing diagram for this machine is shown in Figure 7.17 for an arbitrary input sequence of $x_1 = 1101$.

The *D* flip-flops are clocked on the positive clock transition. To assure that the flip-flops do not become *metastable*, any changes to input x_1 will occur on the negative clock transition. This guarantees that the *D* inputs will be stable before the next positive clock transition, thus meeting the setup requirements for the flip-flop. Metastability is a condition of instability on the output of a flip-flop caused by a change to the data input at or near the active clock transition.

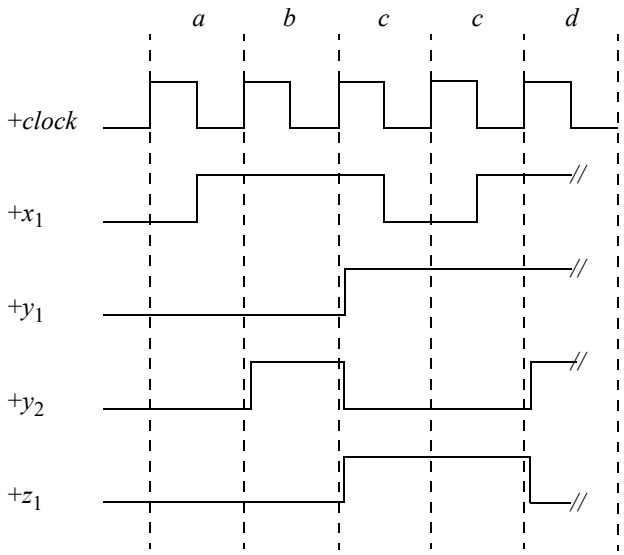


Figure 7.17 Timing diagram for the Moore machine of Table 7.8 for an input sequence of $x_1 = 1101$.

The machine begins in state *a*, where $y_1y_2 = 00$. At the positive clock transition at the end of state *a*, both flip-flops are clocked and the following events occur:

$y_1 = 0$, because all the terms in Equation 7.17 for Dy_1 were zero
$y_2 = 1$, because the term $x_1y_2' = 1$ in Equation 7.17 for Dy_2

Notice the small delay in the assertion of y_2. This is the propagation delay of the internal logic of flip-flop associated with the assertion of y_2. At the time when the

positive transition of *clk* occurred at the end of state *a*, flip-flop y_2 was deasserted and x_1 was asserted; therefore, the term x_1y_2' was true. The machine then enters state *b*, where $y_1y_2 = 01$.

In state *b*, x_1 remains asserted and the following events occur at the positive clock transition at the end of state *b*:

$y_1 = 1$, because the term $x_1y_1'y_2 = 1$ in Equation 7.17 for Dy_1
$y_2 = 0$, because all the terms in Equation 7.17 for Dy_2 were zero

The machine then enters state *c* the first time where $y_1y_2 = 10$. Output z_1 is asserted in state *c* regardless of the value of x_1. As before, the flip-flops are clocked on the positive clock transition and the flip-flop outputs become stable after the appropriate propagation delay of the devices. Input x_1 becomes deasserted at the negative clock transition of the first state *c*; flip-flop y_2 is already deasserted. Therefore, when the positive clock transition occurs at the end of the first state *c*, the following conditions exist:

$y_1 = 1$, because the term $x_1'y_1 = 1$ in Equation 7.17 for Dy_1
$y_2 = 0$, because all the terms in Equation 7.17 for Dy_2 were zero

The machine then enters state *c* the second time and x_1 becomes asserted. At the positive clock transition at the end of the second state *c*, the machine enters state *d*, where $y_1y_2 = 11$, because the following events occur:

$y_1 = 1$, because the term $y_1y_2' = 1$ in Equation 7.17 for Dy_1
$y_2 = 1$, because the term $x_1y_2' = 1$ in Equation 7.17 for Dy_2

From Table 7.8, it can be seen that the next state will be either state *d* if $x_1 = 0$ or state *a* if $x_1 = 1$. When analyzing a synchronous sequential machine, a timing diagram shows more detail than any other analytical method — the state times are precisely defined and the propagation delays are clearly illustrated. Since the clock is an astable multivibrator, the clock has no stable level. The clock signal in Figure 7.17 is specified as $+clk$, where the plus sign indicates that the positive clock transition is used to clock the flip-flops which are positive-edge-triggered devices. All other signals are active high (+).

State diagram A *state diagram* is a directed graph which is used in conjunction with the state table. The state diagram portrays the same information as the state table, but presents a graphical representation in which the state transitions are more easily followed. The state diagrams that are used in this book are similar to flow chart diagrams in which the transition sequences and thus, the operational characteristics of the machine, are clearly delineated. Two symbols are used: a state symbol and an output symbol.

The *state symbol* is designated by a circle as shown in Figure 7.18. These nodes (or vertices) correspond to the state of the machine; the state name, such as state *a*, is

placed inside the circle. The connecting directed lines between states correspond to the allowable state transitions. There are one or more entry paths and one or more exit paths as indicated by the arrows, unless the vertex is a *terminal state*, in which case there is no exit.

The flip-flop names are positioned alongside the state symbol. In Figure 7.18, the machine is designed using three flip-flops which are designated as $y_1y_2y_3$, where y_3 is the low-order flip-flop. Directly beneath the flip-flop names, the *state code* is specified. The state code represents the state of the individual flip-flops. In Figure 7.18, the state code is 101, which corresponds to $y_1y_2'y_3$. If an input causes a transition from state *a* to another state, this is indicated by placing the name of the input variable adjacent to the exit arrow as shown in Figure 7.17 for x_1 and x_1'.

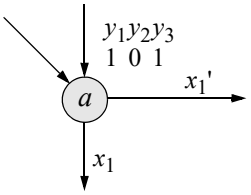


Figure 7.18 State diagram state symbol indicating state *a*.

The *output symbol* is represented by a rectangle and is placed immediately following the state symbol, as shown in Figure 7.19(a) for a Moore machine, or placed immediately after an input variable that causes the output to become active, as shown in Figure 7.19(b) for a Mealy machine. Figure 7.19(a) specifies a Moore machine in which output z_1 is a function of the present state only; that is, state *b*, where $y_1y_2 = 01$. Figure 7.19(b) indicates a Mealy machine in which output z_1 is a function of both the present state *b* ($y_1y_2 = 01$) and input x_1.

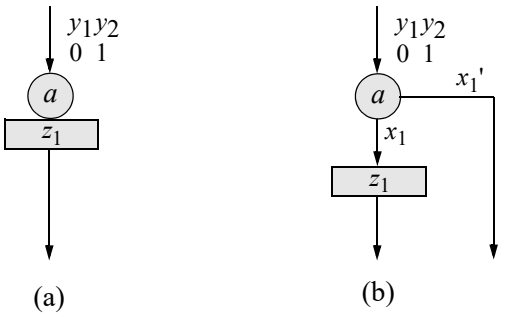


Figure 7.19 State diagram output symbol indicating output z_1: (a) Moore machine and (b) Mealy machine.

For a Moore machine, the outputs can be asserted for segments of the clock period rather than for the entire clock period only. This is illustrated in Figure 7.20 where the positive clock transitions define the clock cycles, and hence, the state times. Two clock cycles are shown, one for the present state $Y_{j(t)}$ and one for the next state $Y_{k(t+1)}$.

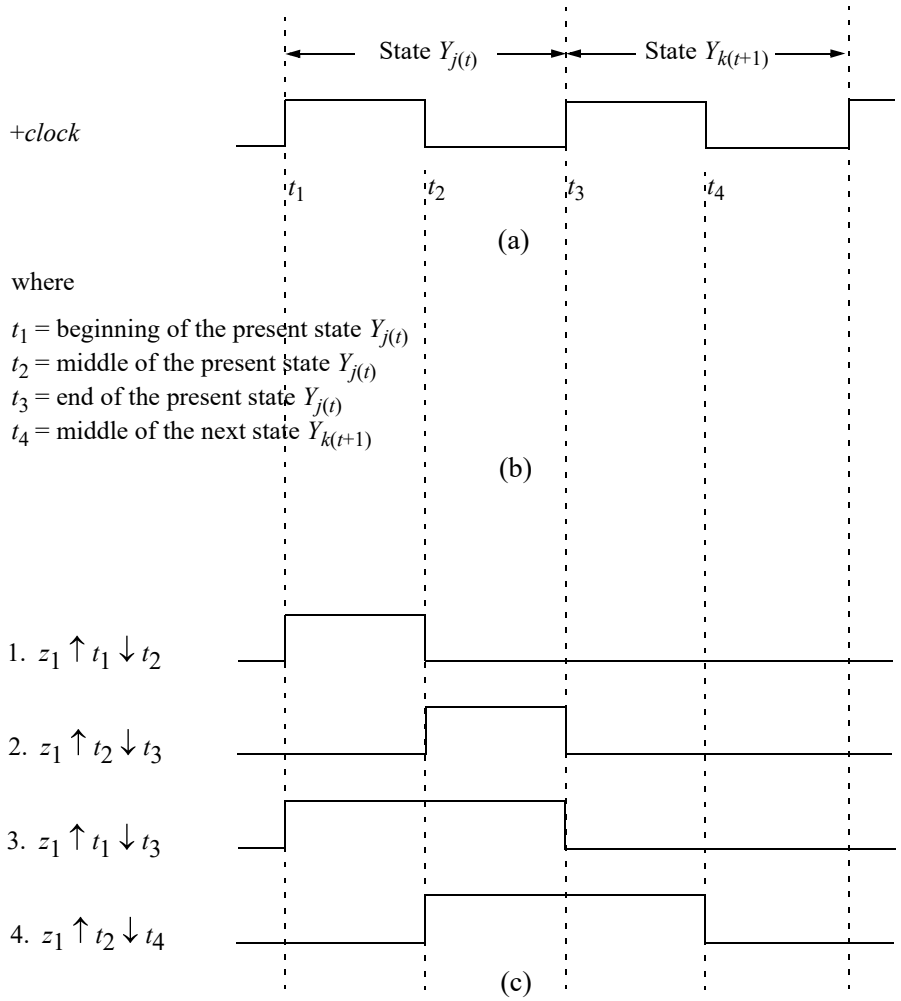


Figure 7.20 Output assertion/deassertion times for Moore machines: (a) clock pulses; (b) definition of assertion/deassertion times; and (c) assertion/deassertion statements with corresponding asserted outputs.

The leading edge of the clock pulse, which defines the beginning of the present state, is labeled t_1. The leading edge may be a positive or negative clock transition and

is used for clocking positive- or negative-edge-triggered devices, respectively. All assertion/deassertion times are referenced to the present state $Y_{j(t)}$. Time t_2 occurs at the middle of the present state; time t_3 occurs at the end of the present state; and time t_4 occurs at the midpoint of the next state $Y_{k(t+1)}$. The assertion of an output is indicated by an up-arrow (↑); deassertion is indicated by a down-arrow (↓). The output assertion/deassertion times for a Mealy machine cannot be uniquely specified as for a Moore machine, because the outputs are contingent not only upon a specific state but also upon the input variables, whose assertion times may not be known.

Asserting the output signals at various times and for different durations, as shown in Figure 7.20(c), provides more flexibility in the λ output logic. Waveforms 2 and 4 are especially useful in avoiding glitches, because the assertion of the outputs is delayed from the active clock transition where flip-flops change state — not all flip-flops may change state simultaneously.

7.1.5 Analysis Examples

Five synchronous sequential machines will be analyzed in this section. The first is a Moore implementation with *D* flip-flops; the second is a Moore machine implemented with *JK* flip-flops; the third is a Moore machine implemented with *D* flip-flops and linear-select multiplexers; the fourth is a Mealy machine implemented with one *JK* flip-flop; the fifth is a Mealy-Moore machine implemented with two *JK* flip-flops.

Example 7.1 The first synchronous sequential machine to be analyzed is the Moore machine of Figure 7.21 consisting of a single input x_1, two *D* flip-flops y_1 and y_2, and a single output z_1. Since there is only one input, the input alphabet is quite simple, containing only one scalar input: $x_1 = 0$ or $x_1 = 1$. Two storage elements y_1 and y_2 assume four unique values to specify four states $y_1y_2 = 00$, 01, 10, and 11.

State names are assigned to the states as shown in the next-state table of Table 7.10 where state *a* corresponds to $y_1y_2 = 00$. The state names do not necessarily have to be in ascending sequence when tabulated; they are, however, arranged in an ascending systematic sequence when entered in the state diagram, as will be shown later in this example.

The output alphabet consists of one scalar output: $z_1 = 0$ or $z_1 = 1$. Output z_1 is asserted during the last half of the clock pulse when *–clock* becomes a high logic level and $y_1y_2 = 11$. This indicates a Moore machine since the output is a function of the present state only and is not a function of the input. The machine will be analyzed by obtaining the next-state table, the input maps, the output map, the timing diagram, and the state diagram.

The next state of the Moore machine in this example is defined as

$$Y_{k(t+1)} = \begin{cases} \delta y_1(x_1, y_2) = x_1y_2 \\ \delta y_2(x_1, y_1) = x_1' + y_1' \end{cases}$$

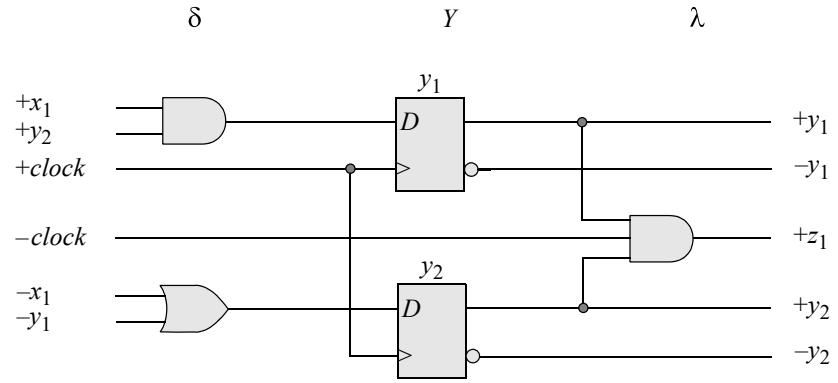


Figure 7.21 Moore machine for Example 7.1.

Table 7.10 Next-State Table for the Moore Machine of Figure 7.21

State Name	Present State y_1y_2	Input x_1	Flip-Flop Inputs Dy_1 Dy_2	Next State y_1y_2	Output z_1
a	0 0	0	0 1	0 1	0
	0 0	1	0 1	0 1	0
b	0 1	0	0 1	0 1	0
	0 1	1	1 1	1 1	0
d	1 0	0	0 1	0 1	0
	1 0	1	0 0	0 0	0
c	1 1	0	0 1	0 1	1
	1 1	1	1 0	1 0	1

The input maps are shown in Figure 7.22 and can be derived from either the next-state table or from the logic diagram. For example, in Table 7.10, states *b* and *c* specify that Dy_1 is asserted only when both x_1 and y_2 are asserted. Therefore, using x_1 as a map-entered variable, the input map for Dy_1 contains the entry x_1 in column y_2. In the same manner, the input map for y_2 and the output map for z_1 are derived.

The equation for Dy_2 can be derived by one of two methods:

1. Read directly from the map without changing the minterm entries. This yields $Dy_2 = y_1' + y_1x_1'$. Then using the absorption law, $Dy_2 = x_1' + y_1'$, or
2. Change the map entries as follows:

y_1 \ y_2	0	1
0	$x_1 + x_1'$	$x_1 + x_1'$
1	x_1'	x_1'

Dy_2

This does not change the minterm values. Since every square contains x_1', therefore, $Dy_2 = x_1'$. Now reassign $x_1 + x_1'$ as a value of 1, then combine the two 1s in row $y_1 = 0$ as $Dy_2 = y_1'$. Thus, $Dy_2 = x_1' + y_1'$.

$y_1\,y_2$ \ x_1	0	1
0 0	0	0
0 1	0	1
1 1	0	1
1 0	0	0

$Dy_1 = y_2x_1$

$y_1\,y_2$ \ x_1	0	1
0 0	1	1
0 1	1	1
1 1	1	0
1 0	1	0

$Dy_2 = x_1' + y_1'$

Figure 7.22 Input maps for the Moore machine of Figure 7.21.

The output map is shown in Figure 7.23 and can also be derived from either the next-state table or from the logic diagram. The output map does not show the time when z_1 is asserted, only the logic that causes the assertion.

$y_1\,y_2$ \ x_1	0	1
0 0	0	0
0 1	0	0
1 1	1	1
1 0	0	0

$z_1 = y_1y_2$

Figure 7.23 Output map for the Moore machine of Figure 7.21.

The timing diagram is shown in Figure 7.24 using an arbitrary input sequence of $x_1 = 1011$. The machine is reset initially to state *a*. Using the timing diagram in conjunction with the logic diagram of Figure 7.21 or the next-state table of Table 7.10, the machine proceeds to state *b* at the next positive clock transition, regardless of the value of x_1. At the positive clock transition at the end of state *b*, the following conditions exist: $x_1 = 0$, $y_1 = 0$, and $y_2 = 1$, and the machine remains in state *b* ($y_1y_2 = 01$), because

$$Dy_1 = x_1y_2 = 01 = 0$$
$$Dy_2 = x_1' + y_1' = 1 + 1 = 1$$

Using a similar procedure for the remaining states, it can be determined that the state transition sequence for $x_1 = 1011$ is *abbcd* · · · . Output z_1 is asserted when y_1y_2 *clock* = 110; that is, y_1y_2 *clock*' causes z_1 to be active during the last half of the clock cycle in which $y_1y_2 = 11$.

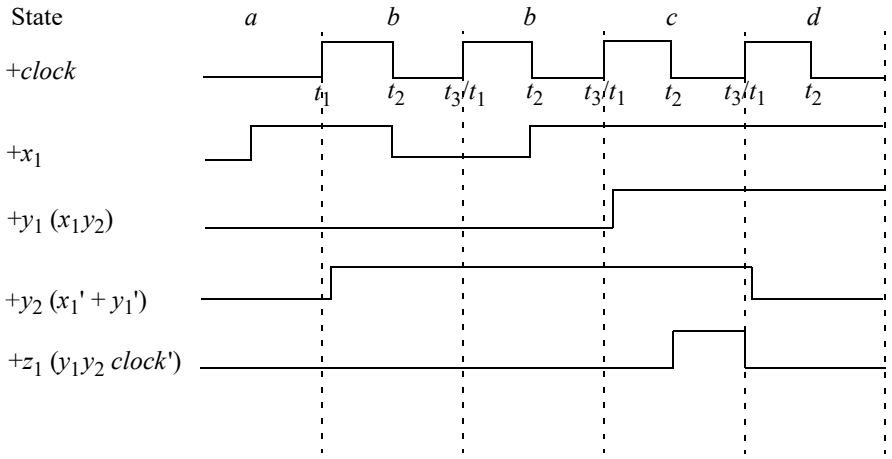


Figure 7.24 Timing diagram for the Moore machine of Figure 7.21 using an arbitrary sequence of $x_1 = 1011$.

The final analysis technique is the state diagram, which is a graphical representation of the functions δ and λ. The state diagram, illustrated in Figure 7.25, is derived directly from the next-state table of Table 7.10. Notice the assertion and deassertion of z_1 in state *c*. Output z_1 is asserted in state *c* at the midpoint (t_2) of the cycle and is deasserted at the end (t_3) of the cycle. This prevents a possible erroneous output (or glitch) from occurring when the state transition is from state *d* to state *b*.

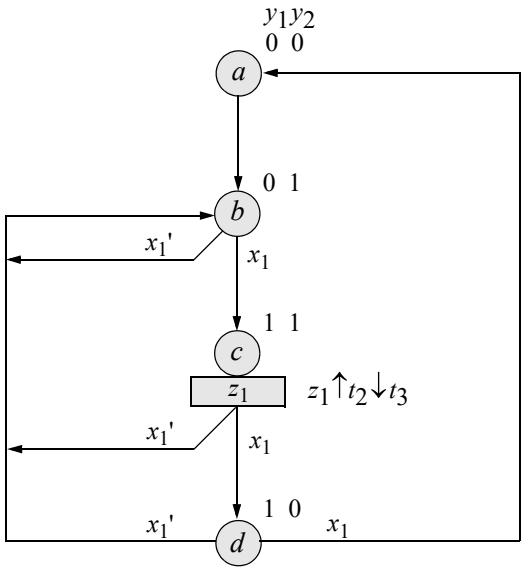


Figure 7.25 State diagram for the Moore machine of Figure 7.21.

Output glitches can occur when two or more flip-flops change state and the output assertion is at time t_1. When the active clock transition triggers the machine to initiate a state transition from state *d* to state *b*, both flip-flops change state. The machine then enters a period of instability until the machine stabilizes in state *b* ($y_1y_2 = 01$). If y_2 is faster at setting than y_1 is at resetting, then the machine will momentarily enter state *c* ($y_1y_2 = 11$). Because state *c* contains a Moore-type output, whenever the machine enters state *c*, output z_1 will be asserted. Thus, an erroneous output will be generated on z_1 as the machine passes through transient state *c* for a state transition sequence of $d \rightarrow b$.

The state diagram presents a much clearer state transition sequence than either the logic diagram or the timing diagram. The state diagram is hardware-independent; that is, it shows the complete operation of the machine at a glance for all possible input sequences, but does not indicate the type of storage elements or gates that are used in the implementation. The logic diagram shows the physical realization of the machine; however, the sequence of state transitions is obtained only after a tedious examination of inputs and present states. The timing diagram is an illustration of actual waveforms that would be observed on an oscilloscope for a given input sequence.

Example 7.2 Given the Moore machine of Figure 7.26, the machine will be analyzed by deriving next-state table and the state diagram. The next-state table is shown in Table 7.11. The machine is reset to state *a* ($y_1y_2 = 00$). In state *a*, the value of $Jy_1Ky_1 = 10$ regardless of the value of x_1. This represents a set condition for a *JK* flip-

flop; therefore, the next state for flip-flop $y_1 = 1$. Similarly, the value of $Jy_2Ky_2 = 01$, regardless of the value of x_1. This represents a reset condition for a *JK* flip-flop; therefore, the next state for flip-flop $y_2 = 0$. Thus, at the next negative clock transition the next state is state b ($y_1y_2 = 10$) and output z_1 is asserted, as shown in the state diagram of Figure 7.27.

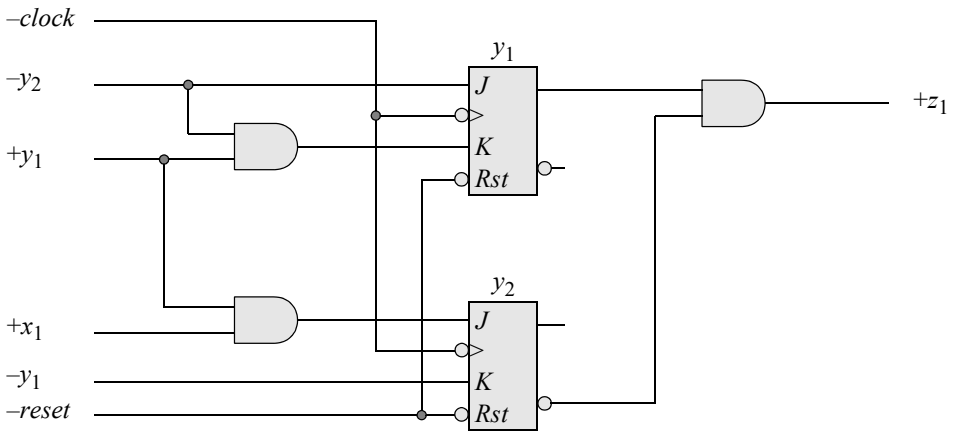


Figure 7.26 Moore machine to be analyzed for Example 7.2.

Table 7.11 Next-State Table for the Moore Machine of Figure 7.26

State Name	Present State y_1y_2	Input x_1	Flip-Flop Inputs Jy_1Ky_1	Flip-Flop Inputs Jy_2Ky_2	Next State y_1y_2	Output z_1
a	0 0	0	1 0	0 1	1 0	0
	0 0	1	1 0	0 1	1 0	0
b	1 0	0	1 1	0 0	0 0	1
	1 0	1	1 1	1 0	0 1	1
c	0 1	0	0 0	0 1	0 0	0
	0 1	1	0 0	0 1	0 0	0

In state b, if $x_1 = 0$, then $Jy_1Ky_1 = 11$, which represents a toggle condition for a *JK* flip-flop and y_1 toggles from 1 to 0 and $Jy_2Ky_2 = 00$, which represents a no change condition for a *JK* flip-flop; therefore, the next state is a ($y_1y_2 = 00$). In state b, if $x_1 = 1$, then $Jy_1Ky_1 = 11$, which toggles flip-flop y_1 from 1 to 0. In state b, if $x_1 = 1$,

then $Jy_2Ky_2 = 10$, which represents a set condition for a *JK* flip-flop and $y_2 = 1$; therefore, the next state is $y_1y_2 = 01$. In a similar manner, state *c* is obtained. The is no state *d* ($y_1y_2 = 11$) for this machine.

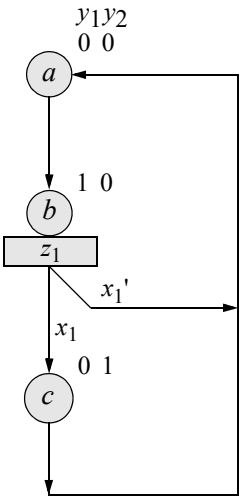


Figure 7.27 State diagram for the Moore machine of Figure 7.26.

Example 7.3 The logic diagram for the Moore machine shown in Figure 7.28 will be analyzed by obtaining the state diagram, the input maps, and the output maps.

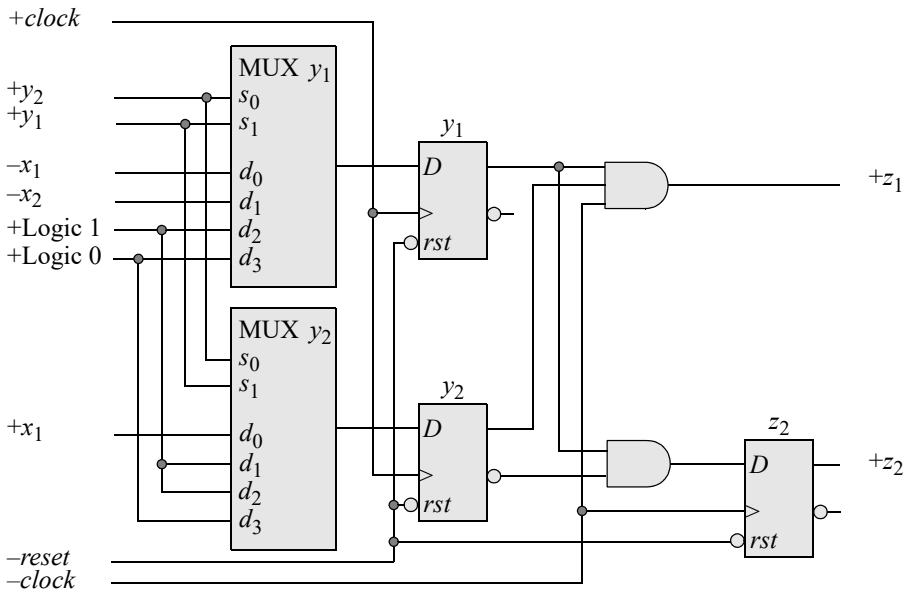


Figure 7.28 Logic diagram for the Moore machine of Example 7.3.

Output z_1 has the following assertion/deassertion times: $z_1\uparrow t_2\downarrow t_3$; that is, z_1 is asserted during the last half of the clock cycle. Output z_2 has the following assertion/deassertion times: $z_2\uparrow t_2\downarrow t_4$; that is, z_2 is asserted during the last half of the present clock cycle and during the first half of the following clock cycle.

The multiplexers are linear-select multiplexers with select inputs of $s_0s_1 = y_1y_2$, where y_2 is the low-order select input. Flip-flops y_1 and y_2 are clocked on the positive transition of the clock. Note that output z_1 is asserted by the term y_1y_2clk', which is the last half of the present clock cycle. In order to have output z_2 asserted from t_2 to t_4, a D flip-flop is used, which is clocked on the negative transition of the clock.

The logic diagram will now be analyzed. The machine is reset to state a (y_1y_2 = 00). In state a, if $x_1 = 0$, then the D input for flip-flop $y_1 = 1$ and the D input for flip-flop $y_2 = 0$. Therefore, the next state is d ($y_1y_2 = 10$) in which the flip-flop for z_2 is clocked on the negative transition of the clock. In state a, if $x_1 = 1$, then the D input for flip-flop $y_1 = 0$ and the D input for flip-flop $y_2 = 1$. Therefore, the next state is b (y_1y_2 = 01).

In state b, if $x_2 = 1$, then flip-flop $y_1 = 0$ at the next positive clock transition and flip-flop $y_2 = 1$. Therefore, the machine remains in state b. In state b, if $x_2 = 0$, then the next state is c ($y_1y_2 = 11$), in which output z_1 is asserted during the last half of the clock cycle. In a similar manner, the remaining states are derived, producing the state diagram of Figure 7.29.

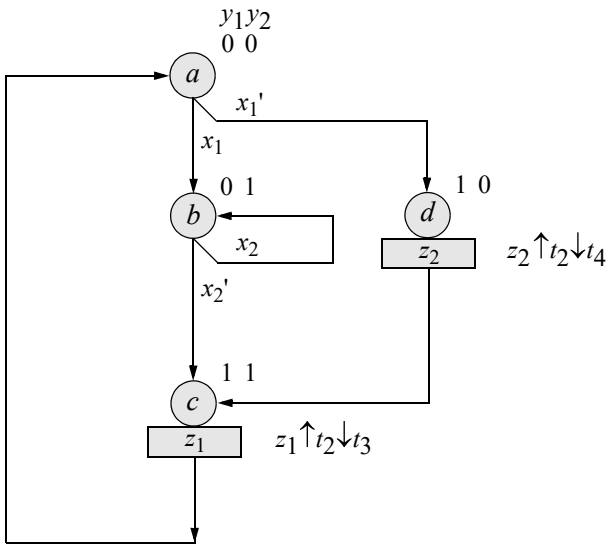


Figure 7.29 State diagram of the Moore machine of Example 7.3.

The input maps for flip-flops y_1 and y_2 are shown in Figure 7.30 using x_1 and x_2 as map-entered variables. The input maps are derived from the logic diagram or from the next-state table. In state a ($y_1y_2 = 00$), flip-flop $y_1 = 1$ only if x_1 is deasserted and

flip-flop $y_2 = 1$ only if x_1 is asserted. Therefore, minterm location 0 in the map for Dy_1 contains the variable x_1' and minterm location 0 in the map for Dy_2 contains the variable x_1. In state b ($y_1y_2 = 01$), flip-flop $y_1 = 1$ only if x_2 is deasserted and flip-flop $y_2 = 1$ regardless of value of x_2. In a similar manner, the remaining minterm locations in both maps are derived. The input equations for Dy_1 and Dy_2 are not necessary, because multiplexers are used for the input logic.

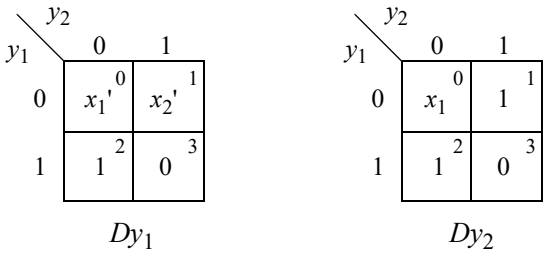


Figure 7.30 Input maps for the Moore machine of Figure 7.28.

The output maps for z_1 and z_2 are shown in Figure 7.31 and are derived directly from the state diagram or from the logic diagram. The equations for z_1 and z_2 are shown in Equation 7.18.

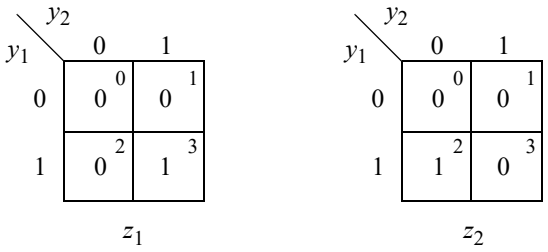


Figure 7.31 Output maps for the Moore machine of Figure 7.28.

$$z_1 = y_1y_2$$
$$z_2 = y_1y_2' \tag{7.18}$$

Example 7.4 The Mealy machine shown in Figure 7.32 will be analyzed by obtaining the next-state table and the state diagram. By applying all combinations of x_1x_2 in both states of the flip-flop, the next-state table shown in Table 7.12 is obtained. Either the next-state table or the logic diagram can be used to obtain the state diagram of Figure 7.33.

The machine is reset initially to state a ($y_1 = 0$). Since the K input of the JK flip-flop is connected to a logic 1, the flip-flop will either reset or toggle upon the application of a positive clock transition. In state a ($y_1 = 0$), the machine will sequence to state b (1) for the following conditions:

$$x_1 + x_1'x_2 = x_1 + x_2.$$

Output z_1 will be asserted in state b ($y_1 = 1$) if input x_1 is asserted.

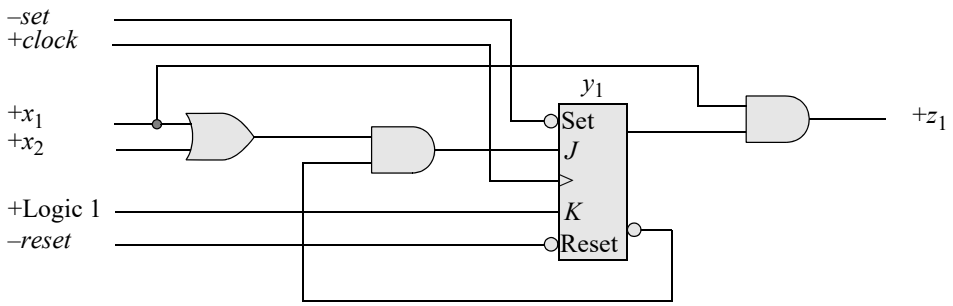


Figure 7.32 Logic diagram for the Mealy machine of Example 7.4.

Table 7.12 Next-State Table for the Mealy Machine of Example 7.4

State Name	Present State y_1	Inputs x_1x_2	Flip-Flop Inputs Jy_1Ky_1	Next State y_1	Output z_1
a	0	0 0	0 1	0	0
	0	0 1	1 1	1	0
	0	1 0	1 1	1	0
	0	1 1	1 1	1	0
b	1	0 0	0 1	0	0
	1	0 1	0 1	0	0
	1	1 0	0 1	0	1
	1	1 1	0 1	0	1

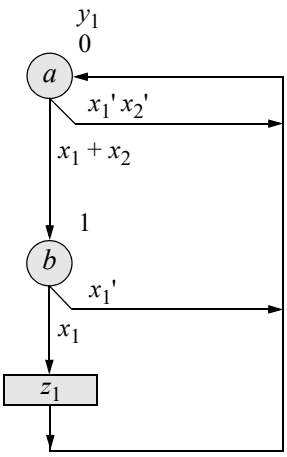


Figure 7.33 State diagram for the Mealy machine of Example 7.4.

Example 7.5 The synchronous sequential machine shown in Figure 7.34 is implemented with *JK* flip-flops and contains both Moore- and Mealy-type outputs. The machine will be analyzed by obtaining the next-state table for all combinations of the input variables, the state diagram, the input maps for Jy_1Ky_1 and Jy_2Ky_2, and the output maps for z_1 and z_2. All Karnaugh maps will use x_1 and x_2 as map-entered variables. The machine is reset to state a ($y_1y_2 = 00$).

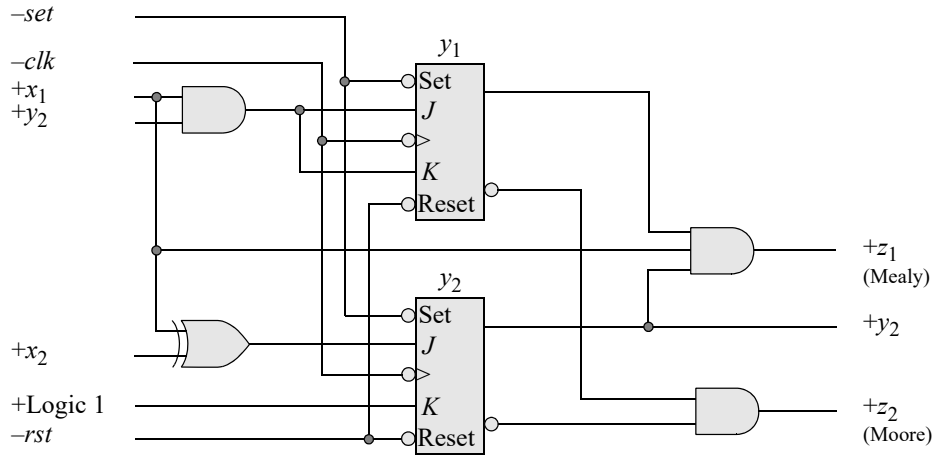


Figure 7.34 Logic diagram for the synchronous sequential machine of Example 7.5.

The δ next-state logic consists of one AND gate and one exclusive-OR function. There are four possible input vectors: $x_1x_2 = 00$, 01, 10, and 11. There are four possible states: $y_1y_2 = 00$, 01, 10, and 11. The λ output logic consists of two AND gates: one for z_1 and one for z_2. By applying all combinations of the inputs beginning in state a ($y_1y_2 = 00$), the next-state table of Table 7.13 is obtained.

For example, in state a, $y_1 = 0$; therefore, $Jy_1Ky_1 = 00$ and there is no change to the present state of flip-flop y_1. For flip-flop y_2, $Jy_2 = 1$ whenever x_1 and x_2 contain different values. Since $Ky_2 = 1$, the JK values are either $JK = 01$ or 11 and the next-state for flip-flop y_2 is either $y_2 = 0$ or $y_2 = 1$. Therefore, in state a, the next state for the machine is either $y_1y_2 = 00$ or $y_1y_2 = 01$. In a similar manner, the remaining next states are obtained.

By progressing through Table 7.13 line by line, the next-state table can be easily translated into the state diagram of Figure 7.35, which depicts the same information as the next-state table, but exhibits a graphical representation in which the state transitions are more readily observed.

Table 7.13 Next-State Table for the Synchronous Sequential Machine of Figure 7.34

State Name	Present State y_1y_2	Inputs x_1x_2	Flip-Flop Inputs Jy_1Ky_1	Jy_2Ky_2	Next State y_1y_2	Outputs z_1z_2
a	0 0	0 0	0 0	0 1	0 0	0 1
		0 1	0 0	1 1	0 1	0 1
		1 0	0 0	1 1	0 1	0 1
		1 1	0 0	0 1	0 0	0 1
b	0 1	0 0	0 0	0 1	0 0	0 0
		0 1	0 0	1 1	0 0	0 0
		1 0	1 1	1 1	1 0	0 0
		1 1	1 1	0 1	1 0	0 0
c	1 0	0 0	0 0	0 1	1 0	0 0
		0 1	0 0	1 1	1 1	0 0
		1 0	0 0	1 1	1 1	0 0
		1 1	0 0	0 1	1 0	0 0
d	1 1	0 0	0 0	0 1	1 0	0 0
		0 1	0 0	1 1	1 0	0 0
		1 0	1 1	1 1	0 0	1 0
		1 1	1 1	0 1	0 0	1 0

The input maps for Jy_1Ky_1 and Jy_2Ky_2 are shown in Figure 7.36 using x_1 and x_2 as map-entered variables and yield the equations of Equation 7.19. The output maps

are shown in Figure 7.37, also using x_1 and x_2 as map-entered variables and yield the equations of Equation 7.20. All Karnaugh maps are derived from either the logic diagram, the next-state table, or the state diagram.

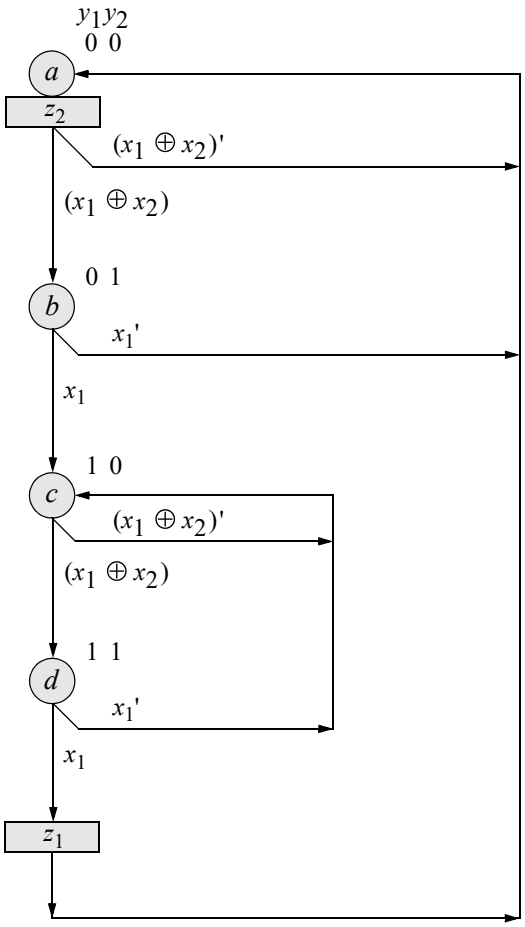
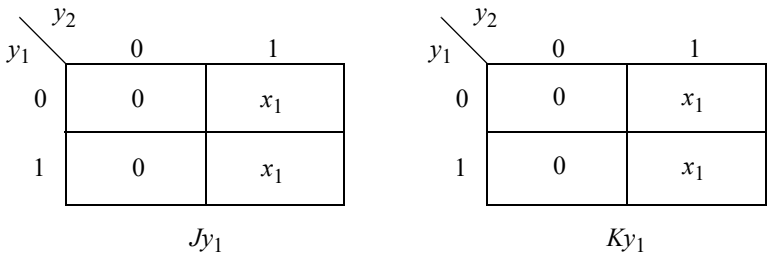


Figure 7.35 State diagram for the synchronous sequential machine of Figure 7.34.



(Continued on next page)

Figure 7.36 Input maps for the synchronous sequential machine of Figure 7.34.

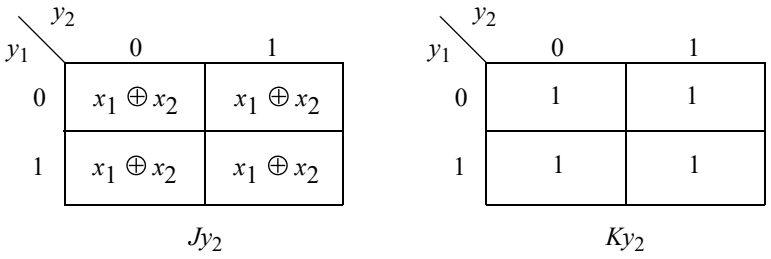


Figure 7.36 (Continued)

$$Jy_1 = y_2x_1$$
$$Ky_1 = y_2x_1$$

$$Jy_2 = x_1 \oplus x_2$$
$$Ky_2 = 1 \tag{7.19}$$

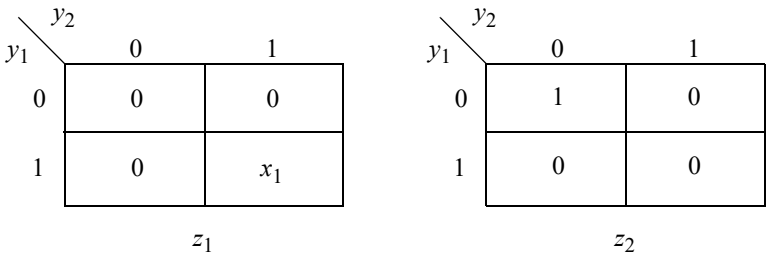


Figure 7.37 Output maps for the synchronous sequential machine of Figure 7.34.

$$z_1 = y_1y_2x_1$$
$$z_2 = y_1'y_2' \tag{7.20}$$

7.2 Synthesis of Synchronous Sequential Machines

Techniques for synthesizing (designing) synchronous sequential machines are introduced. A detailed procedure is presented to synthesize a synchronous sequential machine from a given set of machine specifications. The design process begins with a set of specifications and culminates with a logic diagram or a list of Boolean functions from which the logic can be designed.

Unlike combinational logic, which can be completely specified by a truth table, a synchronous sequential machine requires a state diagram or state table for its precise description. The state diagram depicts the sequence of events that must occur in order for the machine to perform the functions which are defined in the machine specifications. A *synchronous sequential machine* consists of storage elements, usually flip-flops, and δ next-state combinational logic that connects to the flip-flop data inputs. The machine may also contain combinational logic for the λ output function. In some cases, the output logic may require one or more storage elements, depending on the assertion and deassertion of the output signals.

The number of flip-flops is determined by the number of states required by the machine. The combinational logic is derived directly from either the state diagram or from the state table. When the type and quantity of storage elements has been determined, the design process proceeds in a manner analogous to that of combinational logic design.

The requirement for hardware minimization is of paramount importance. Hardware minimization is realized by reducing (or minimizing) the number of states in the machine, thus minimizing the number of storage elements and logic gates, while maintaining the input-output requirements. Reducing the number of states in a machine may not always reduce the number of flip-flops, since the number of eliminated states may not reduce the total state count by a power of two.

For example, if a 16-state machine (requiring $p = 4$ storage elements) is reduced to a 12-state machine, then four storage elements are still required. There will be no reduction in the number of flip-flops until the number of states has been reduced to at least eight ($2^p = 2^3 = 8$ flip-flops). However, the increased number of unused states may result in less combinational logic, because these "don't care" states can be combined with other machine states in a Karnaugh map, resulting in a reduction of combinational logic.

A proper choice of state code assignments may also reduce the number of gates in the δ next-state function logic. Since there are p storage elements, the binary values of these p-tuples can usually be chosen such that the combinational input logic is minimized. A judicious choice of state codes permits more entries in the Karnaugh map to be combined. Since the map entries represent the input logic, combining a greater number of minterm locations results in input equations with fewer terms and fewer variables per term. Thus, an overall reduction in logic gates is realized for the δ next-state logic.

The synthesis procedure utilizes a hierarchical method — also referred to as a *top-down* approach — for machine design. This is a systematic and orderly procedure that

commences with the machine specifications and advances down through increasing levels of detail to arrive at a final logic diagram. Thus, the machine is decomposed into modules which are independent of previous and following modules, yet operate together as a cohesive system to satisfy the machine's requirements.

7.2.1 Synthesis Procedure

This section develops a detailed method for designing synchronous sequential machines using various types of storage elements. The hierarchical design algorithm is shown below.

1. Develop a state diagram from the problem definition, which may be either a word description and/or a timing diagram.

2. Check for equivalent states and then eliminate redundant states.

3. Assign state codes for the storage elements in the form of a binary p-tuple.

4. Generate a next-state table.

5. Select the type of storage element to be used, then generate the input maps for the δ next-state function and derive the input equations.

6. Generate the output maps for the λ output function and derive the output equations.

7. Design the logic diagram using the input equations, the storage elements, and the output equations.

A critical step in synthesizing synchronous sequential machines is the derivation of the state diagram. The *state diagram* specifies the machine performance and gives a clear indication of the state transitions and the output assertion, both of which are a function of the input sequence. If the state diagram is correct, then the remaining steps are relatively straightforward and will result in a logic circuit that performs according to the machine specifications. If, however, the state diagram does not reflect the exact performance of the machine, then the remaining steps — although correct in themselves — will not result in a machine that adheres to the prescribed specifications.

Methods will be described to identify equivalent states, after which the redundant states can be eliminated and the state diagram redrawn as a reduced state diagram. State codes are assigned according to rules which will be defined later. If p storage elements are required to implement the machine, then a state code in the form of a binary p-tuple is assigned to each state in the state diagram. The next-state table is then derived from the state diagram. This step is not always necessary and can be eliminated

in many cases, especially when *D* flip-flops are used. A next-state map may also prove useful for completeness.

When the type of storage elements has been determined, the input maps can then be obtained using the next-state table, and from these maps the corresponding input equations. An alternative approach is to derive the input maps from the state diagram directly. The output maps can be derived directly from the state diagram. The λ output logic is usually combinational, but may require storage elements, depending upon the assertion/deassertion specifications. Finally, the logic diagram is designed using the input equations, the storage elements, and the output equations.

Equivalent states Before exemplifying the steps of the synthesis algorithm, two methods will be presented for determining equivalent states. When equivalent states have been found, all but one are redundant and should be eliminated before implementing the state diagram with hardware. At each node in the state diagram, two events occur: the outputs (if applicable) for the present state are generated as a function of the present state only (Moore) or the present state and inputs (Mealy); the next state is determined as a function of the present state only or the present state and inputs.

When deriving a state diagram from machine specifications, some states might be included which contain no new information that is pertinent to the machine's performance. These are classified as redundant states and should be eliminated since they may increase the amount of logic that is required.

If the state diagram is sufficiently small, then redundant states can be easily recognized as the state diagram is being constructed. For larger state diagrams, redundant states may be inadvertently inserted due to the complexity of the machine specifications. During construction of the state diagram, it is best to obtain a diagram that accurately reflects the machine specifications regardless of the number of states that are included. When a correct state diagram has been established, a simple algorithm can then be utilized to find equivalent states. Redundant states can then be eliminated, yielding a reduced state diagram which still completely characterizes the behavior of the machine.

Two states Y_i and Y_j of a machine are equivalent if, for every input sequence, the output sequence when started in state Y_i is identical to the output sequence when started in state Y_j. Therefore, two states Y_i and Y_j are equivalent if and only if the following two conditions are true:

1. For every input sequence X_i, the output sequence Z_i is the same whether the machine begins in state Y_i or Y_j; that is, $\lambda(Y_i, X_i) = \lambda(Y_j, X_i)$, where $\lambda(Y_i, X_i)$ is the output from present state Y_i with input vector X_i and $\lambda(Y_j, X_i)$ is the output from present state Y_j with input vector X_i.

2. Both states Y_i and Y_j have the same or equivalent next state; that is, $\delta(Y_i, X_i) \equiv \delta(Y_j, X_i)$, where $\delta(Y_i, X_i)$ is the next state for a present state of Y_i with input vector X_i and $\delta(Y_j, X_i)$ is the next state for a present state of Y_j with input vector X_i and the symbol ≡ specifies equivalence.

Row matching Finding equivalent states and then eliminating redundant states will be explained using the state diagram illustrated in Figure 7.38. The technique used in the first method is a *row-matching* procedure employing an approach that is more heuristic than algorithmic. The state diagram in Figure 7.38 depicts a Moore machine with the corresponding next-state table shown in Table 7.14. The machine examines a serial 3-bit word on an serial input line x_1 and generates an output z_1 whenever the 3-bit word is 111 or 000. There is one bit space (one clock period) between contiguous words during which time output z_1 is asserted, as shown below.

$$x_1 = \cdots \left| b_1\, b_2\, b_3 \right| \quad \left| b_1\, b_2\, b_3 \right| \cdots$$
$$\uparrow$$
$$z_1$$

where b_i is 0 or 1.

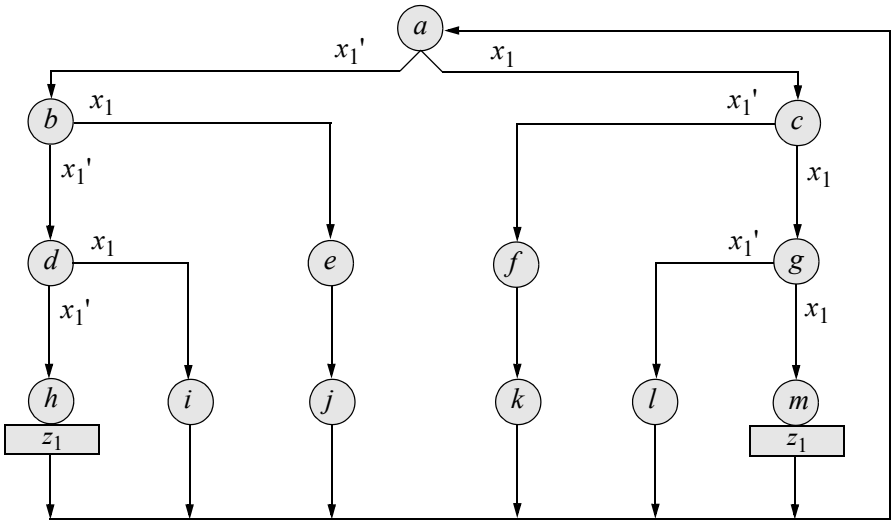


Figure 7.38 Moore machine to detect a sequence of x_1 = 111 or 000.

By carefully considering the machine specifications as the state diagram is being generated, it is relatively easy — in this example — to obtain a state diagram that has no redundant states. However, in order to illustrate the techniques used to identify equivalent states and then to eliminate redundant states, superfluous states have been deliberately inserted.

Using the two rules for equivalence, states *i*, *j*, *k*, and *l* of Table 7.14 are seen to be equivalent, because they all have state *a* as the next state and all have output $z_1 = 0$. By

convention, the lowest number or the lowest ranked letter is retained for the state name and all other equivalent states are given this state name. Therefore, all *j*s, *k*s, and *l*s are changed to *i* wherever they appear in the state diagram. This will eliminate rows *j*, *k*, and *l* from the next-state table. The renamed states are indicated by a slash followed by the new name. Although state *m* also has a next state of *a*, output $z_1 = 1$; thus, state *m* is not equivalent to states *i*, *j*, *k*, and *l*. It is also apparent that states *h* and *m* are equivalent: both have state *a* as the next state and both assert output z_1. Therefore, all *m*s are replaced with *h*s in Table 7.14.

Table 7.14 Next-State Table for the Moore Machine of Figure 7.38

Present State	Input x_1	Next State	Output z_1
a	0	*b*	0
	1	*c*	0
b	0	*d*	0
	1	*e*	0
c	0	*f* / *e*	0
	1	*g*	0
d	0	*h*	0
	1	*i*	0
e	0	*j* / *i*	0
	1	*j* / *i*	0
f / *e*	0	*k* / *i*	0
	1	*k* / *i*	0
g	0	*l* / *i*	0
	1	*m* / *h*	0
h	0	*a*	1
	1	*a*	1
i	0	*a*	0
	1	*a*	0
j / *i*	0	*a*	0
	1	*a*	0
k / *i*	0	*a*	0
	1	*a*	0
l / *i*	0	*a*	0
	1	*a*	0
m / *h*	0	*a*	1
	1	*a*	1

Equivalent states (Eliminate state *f*)

Equivalent states (Eliminate state *m*)

Equivalent states (Eliminate states *j*, *k*, and *l*)

After the equivalent states have been renamed, a check is made for equivalent states using the modified state names to determine if any further equivalences exist. In states *d* and *g*, output $z_1 = 0$. Therefore, states *d* and *g* are equivalent if $h \equiv i$. The outputs for states *h* and *i*, however, are different; thus, states *h* and *i* are not equivalent and consequently states *d* and *g* are not equivalent.

Continuing with the examination of Table 7.14, it is observed that states *e* and *f* are now equivalent, because in both states the next state is *i* and output $z_1 = 0$. Therefore, all *f*s are changed to *e*s wherever *f*s appear in Table 7.14. Further inspection produces no additional equivalent states. Equivalent states are: $e \equiv f$, $h \equiv m$, $i \equiv j \equiv k \equiv l$. The reduced next-state table is shown in Table 7.15; the reduced state diagram with only eight states is illustrated in Figure 7.39.

Table 7.15 Reduced Next-State Table for the Moore Machine of Figure 7.38

Present state	Input x_1	Next State	Output z_1
a	0	*b*	0
	1	*c*	0
b	0	*d*	0
	1	*e*	0
c	0	*e*	0
	1	*g*	0
d	0	*h*	0
	1	*i*	0
e	0	*i*	0
	1	*i*	0
g	0	*i*	0
	1	*h*	0
h	0	*a*	1
	1	*a*	1
i	0	*a*	0
	1	*a*	0

Implication table The second method for determining equivalent states is by use of an *implication table*. This technique is more algorithmic in nature than the previous method, because it follows a set of well-defined rules to find equivalent states in a finite number of steps. The example of Table 7.14 will again be used, this time to illustrate the steps required to find equivalent states using an implication table. For convenience and clarity, Table 7.14 is reproduced in Table 7.16.

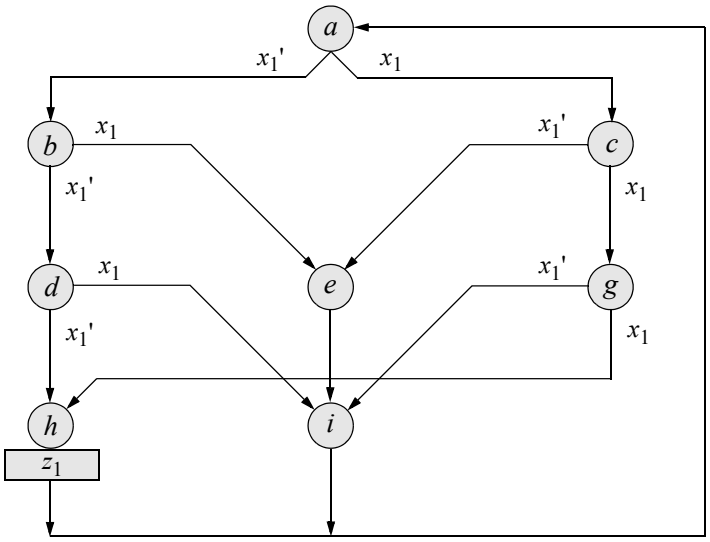


Figure 7.39 Reduced state diagram for the Moore machine of Figure 3.1.

Table 7.16 Next-State Table for the Moore Machine of Figure 7.38

Present State	Input x_1	Next State	Output z_1
a	0	*b*	0
	1	*c*	0
b	0	*d*	0
	1	*e*	0
c	0	*f*	0
	1	g	0
d	0	*h*	0
	1	*i*	0
e	0	*j*	0
	1	*j*	0
f	0	*k*	0
	1	*k*	0
g	0	*l*	0
	1	*m*	0
h	0	*a*	1
	1	*a*	1

(Continued on next page)

Table 7.16 Next-State Table for the Moore Machine of Figure 7.38

Present State	Input x_1	Next State	Output z_1
i	0	a	0
	1	a	0
j	0	a	0
	1	a	0
k	0	a	0
	1	a	0
l	0	a	0
	1	a	0
m	0	a	1
	1	a	1

The implication table is a lower-left triangular matrix whose rows are labeled with the state names in ascending sequence with the exception of the first state. The columns are also labeled with the state names in ascending sequence with the exception of the last state. The reason for omitting the first state name in the upper-left corner of the matrix is because $a \equiv a$, negating the necessity of inserting a square to determine if a is equivalent to a. The same rationale applies to omitting the last state name in the lower-right corner of the matrix. The square at the intersection of a row-column pair is marked with an × if the corresponding states are not equivalent or marked with the symbol ≡ if the states are equivalent.

The first step is to construct a chart of the form shown in Figure 7.40. The chart contains a square for every possible pair of states. The square in row f, column c, for example, corresponds to state pair (f,c). Thus, the squares in the first column correspond to state pairs (b,a), (c,a), (d,a), etc. Squares above the diagonal are not required, because they represent the symmetric property of equivalence; that is, if $c \equiv d$, then $d \equiv c$. Thus, only one of the state pairs is required. Also, squares such as (a,a), (b,b), etc. are omitted, since the state pair within the parentheses is obviously equivalent.

To fill in the first column of the chart, row a of the next-state table shown in Table 7.16 is compared with each of the remaining rows. Consider rows a and b. Since output $z_1 = 0$ for states a and b, then

$$a \equiv b \text{ if and only if } b \equiv d \ (x_1 = 0) \text{ and } c \equiv e \ (x_1 = 1)$$

This is shown in Figure 7.40 where the implied pairs (b,d) and (c,e) are placed in the square for state pair (a,b). An *implied pair* of states indicates that equivalence is implied for the pair but has not yet been verified. During later steps in the procedure, implied pairs will be shown to be either equivalent or nonequivalent. Since this is a

Moore machine, the value of x_1 is not significant when comparing states for equivalence. Next, state pair (a,c) is considered. Output $z_1 = 0$ for states a and c; therefore,

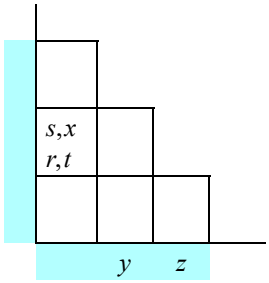
$$a \equiv c \text{ if and only if } b \equiv f \text{ and } c \equiv g$$

This fact is indicated in the table for state pair (a,c). The process continues for each square in the implication table. When two states are not equivalent, an × is placed in the square that corresponds to the state pair under consideration, for example, (a,h), where the outputs differ. When two states are equivalent, this is indicated by placing the equivalence symbol ≡ within the matrix square that corresponds to the state pair under consideration. For example, states h and m are equivalent, because output $z_1 = 1$ for both states and both states have state a as their next state. Therefore, the equivalence symbol ≡ is placed in state pair (h,m).

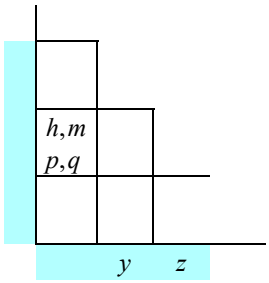
b,d *c,e*											
b,f *c,g*	*d,f* *e,g*										
b,h *c,i*	*d,h* *e,i*	*f,h* *g,i*									
b,j *c,j*	*d,j* *e,j*	*f,j* *g,j*	*h,j* *i,j*								
b,k *c,k*	*d,k* *e,k*	*f,k* *g,k*	*h,k* *i,k*	*j,k*							
b,l *c,m*	*d,l* *e,m*	*f,l* *g,m*	*h,l* *i,m*	*j,l* *j,m*	*k,l* *k,m*						
×	×	×	×	×	×	×					
a,b *a,c*	*a,d* *a,e*	*a,f* *a,g*	*a,h* *a,i*	*a,j*	*a,k*	*a,l* *a,m*	×				
a,b *a,c*	*a,d* *a,e*	*a,f* *a,g*	*a,h* *a,i*	*a,j*	*a,k*	*a,l* *a,m*	×				
a,b *a,c*	*a,d* *a,e*	*a,f* *a,g*	*a,h* *a,i*	*a,j*	*a,k*	*a,l* *a,m*	×				
a,b *a,c*	*a,d* *a,e*	*a,f* *a,g*	*a,h* *a,i*	*a,j*	*a,k*	*a,l* *a,m*	×				
×	×	×	×	×	×	×		×	×	×	×
	b	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*

Figure 7.40 Implication table for the Moore machine of Table 7.16 after the first pass. The symbol × indicates nonequivalent states; the symbol ≡ indicates equivalent states.

Self-implied pairs are redundant and need not be inserted. A self-implied pair is a state pair within a matrix square that is the same as the state pair under consideration. For example, in the entry shown below for state pair (s,x), the entry (s,x) in the matrix square can be eliminated, since (s,x) is a self-implied pair. Thus, $s \equiv x$ if and only if $s \equiv x$ and $r \equiv t$ can be reduced to $s \equiv x$ if and only if $r \equiv t$.



When all state pairs in the row-column coordinates have been compared, each square in the implication table contains implied pairs, the symbol × indicating that the state pair is not equivalent, or the symbol ≡ indicating that the state pair is equivalent. The implied pairs in each square are now checked for equivalence. If one of the implied pairs in a square is not equivalent, then the state pair under consideration is not equivalent. For example, using state pair (s,x) shown below, if implied pair (h,m) is not equivalent, then $s \neg\equiv x$, where the symbol $\neg\equiv$ is read as "not equivalent."



The second pass to find equivalent states is now performed. Beginning with state pair (a,b), continue down column a until each state pair (a,b) through (a,m) has been found either equivalent or not equivalent. It is not always possible, however, to immediately determine equivalency. For example,

$$a \equiv b \text{ if and only if } b \equiv d \text{ and } c \equiv e$$

At this point, the equivalency of state pairs (b,d) and (c,e) is not immediately evident; thus, their equivalency will be resolved at a later step.

Next, state pair (a,c) is checked for equivalence. Since

$$a \equiv c \text{ if and only if } b \equiv f \text{ and } c \equiv g$$

and since it is unknown at this time whether $b \equiv f$ or $c \equiv g$, the process proceeds to state pair (a,d), which specifies that

$$a \equiv d \text{ if and only if } b \equiv h \text{ and } c \equiv i$$

Since $b \not\equiv h$ (the outputs are different), therefore, $a \not\equiv d$ and an $\times$ is placed in the square for state pair (a,d). Continue in this manner for all columns in Figure 7.40. Notice that state pair (d,e) specifies that

$$d \equiv e \text{ if and only if } h \equiv j \text{ and } i \equiv j$$

This statement is true only if both state pairs (h,j) and (i,j) are equivalent. If either state pair is not equivalent, then $d \not\equiv e$. Although state pair (i,j) is equivalent, state pair (h,j) is not equivalent. Therefore, $d \not\equiv e$. The process of finding equivalent states continues until no more $\times$s can be inserted; that is, until no more nonequivalent states can be found. The result of the second pass is shown in Figure 7.41.

Now begin a third pass, using the implication table of Figure 7.41, to determine equivalency between the remaining state pairs. The results are presented in Figure 7.42 Finally, a fourth pass yields Figure 7.43, which illustrates the following equivalent states:

$$e \equiv f$$
$$h \equiv m$$
$$i \equiv j \equiv k \equiv l$$

The results obtained by the implication table are identical to those obtained by the row-matching procedure. Although the implication table method is tedious, it guarantees a reduced state diagram.

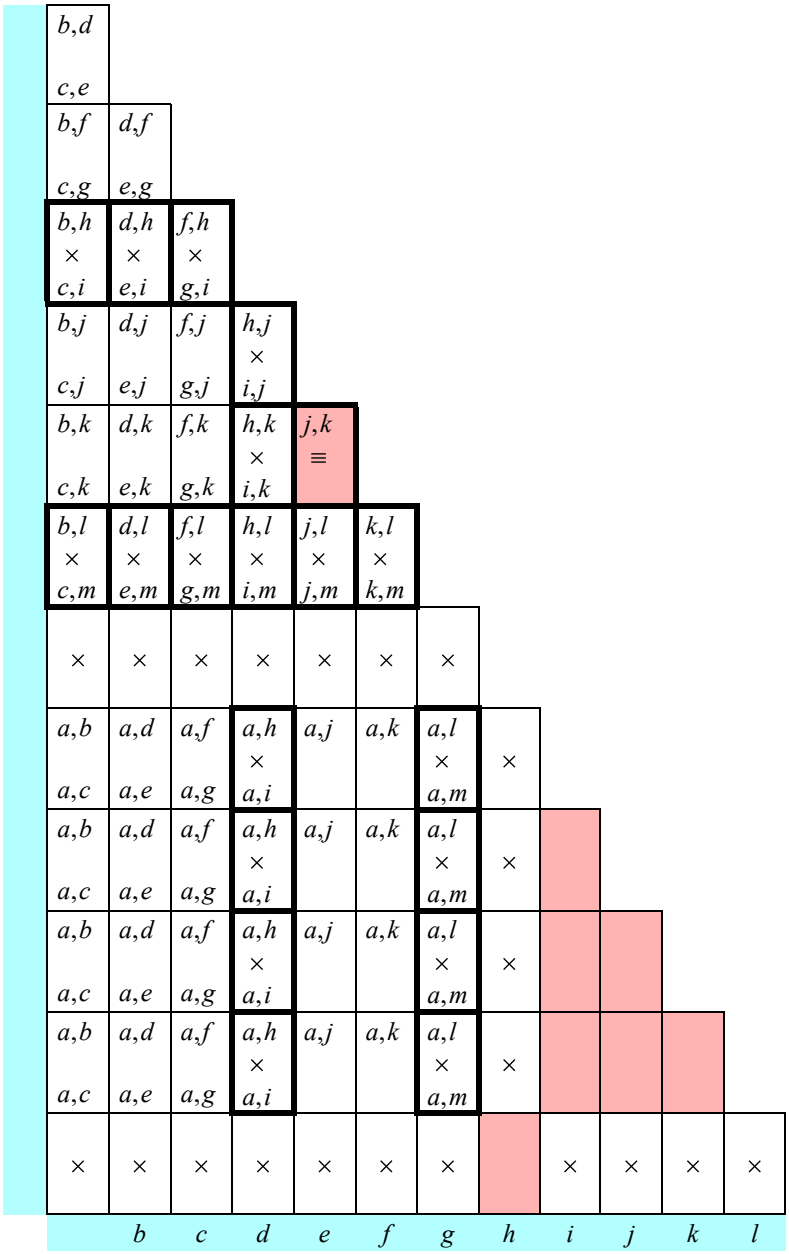


Figure 7.41 Implication table for the Moore machine of Table 7.16 after the second pass. The results of the second pass are shown in bold-lined squares. The symbol × indicates nonequivalent states; the symbol ≡ indicates equivalent states.

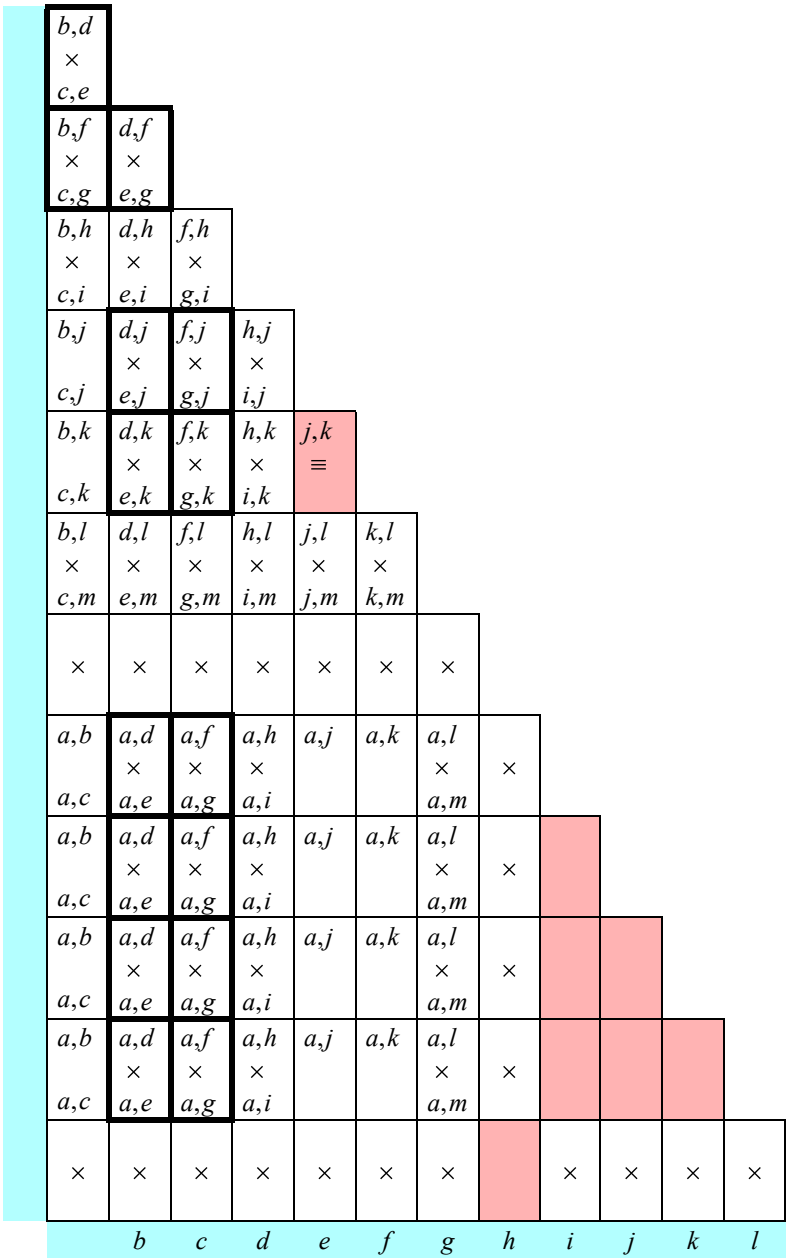


Figure 7.42 Implication table for the Moore machine of Table 7.16 after the third pass. The results of the third pass are shown in bold-lined squares. The symbol × indicates nonequivalent states; the symbol ≡ indicates equivalent states.

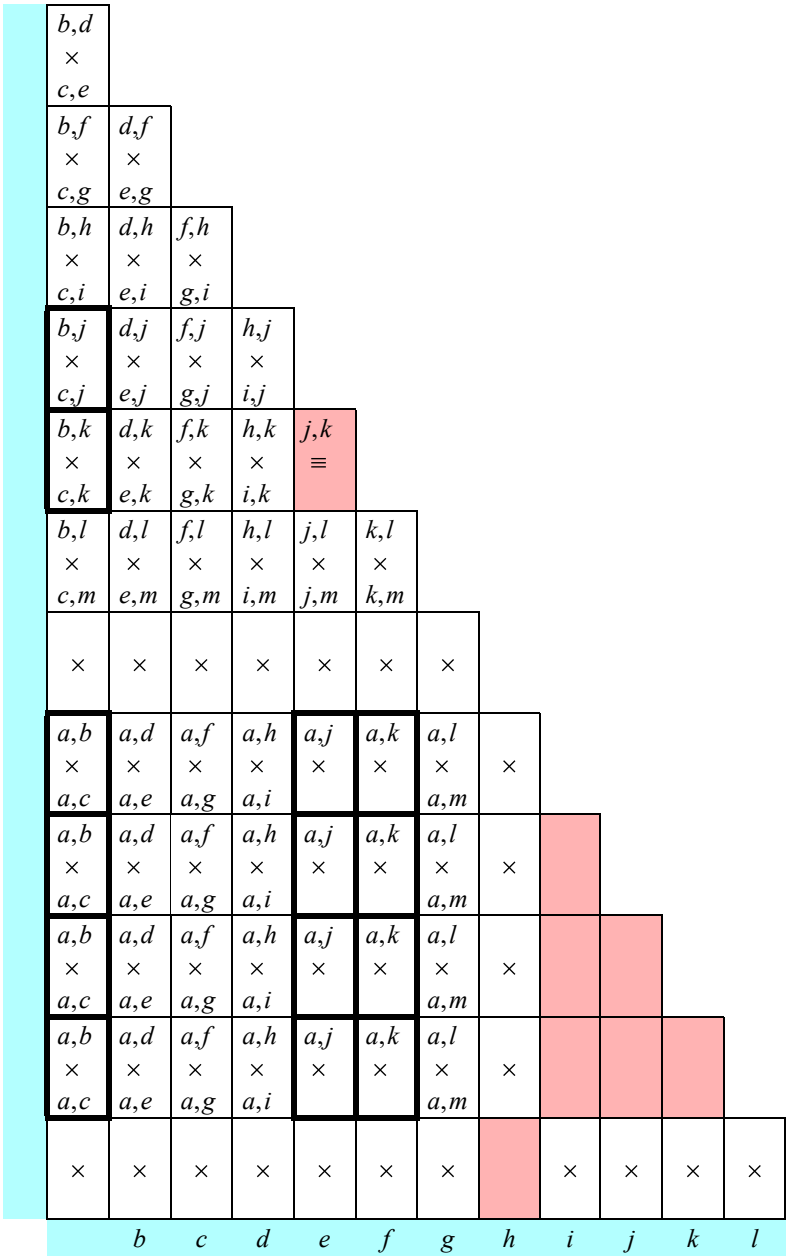
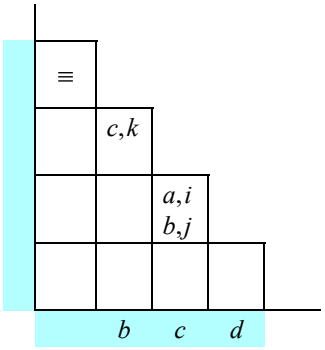


Figure 7.43 Implication table for the Moore machine of Table 7.16 after the fourth pass. The results of the fourth pass are shown in bold-lined squares. The symbol × indicates nonequivalent states; the symbol ≡ indicates equivalent states. Equivalent states are (e,f), (h,m), and (i,j,k,l).

Although the above two procedures were described for a Moore machine, the methods work equally well for a Mealy machine. When a row-column pair is compared for a Mealy machine, the outputs must be the same for each same input value.

There is one additional comment regarding row-column equivalent pairs; that is, interdependence. In the partial implication table shown below, $a \equiv i$. Also, $j \equiv b$ if $c \equiv k$.



Examining state pair (c,k) for equivalence, we see that $c \equiv k$ if $a \equiv i$ and $b \equiv j$. Since it is already known that $a \equiv i$, the equivalence of c and k depends only on the equivalence of b and j. Conversely the equivalence of b and j depends only on the equivalence of c and k. It can be stated, therefore, that $c \equiv k$ if $b \equiv j$ if $c \equiv k$. This is an equivalence condition that is based upon a mutual dependence and is called *interdependence*.

Thus, there is an interdependence between state-pairs (b,j) and (c,k). Therefore, $b \equiv j$ and $c \equiv k$. This can be verified by examining the next-state table of the machine and observing that the output sequence beginning in state b will be the same as the output sequence beginning in state j for the same input sequence. The same rationale is true for states c and k.

7.2.2 Synchronous Registers

An ordered set of storage elements and associated combinational logic, where each cell performs an identical or similar operation, is called a *register*. Each cell of a register stores one bit of binary information. There are many different types of synchronous registers, including parallel-in, parallel-out; parallel-in, serial-out; serial-in, parallel-out; and serial-in, serial-out registers. In the synthesis of synchronous registers, it is not always necessary to use the formalized design procedure previously described. Usually, intuitive reasoning and experience are sufficient requisites for the synthesis of these elementary storage devices.

The next state of a register is usually a direct correspondence to the input vector, whose binary variables connect to the flip-flop data inputs, either directly or through δ next-state logic. Most registers are used primarily for temporary storage of binary

data, either signed or unsigned, and do not modify the data internally; that is, the state of the register is unchanged until the next active clock transition. Other registers may modify the data in some elementary manner such as, shifting left or shifting right, where a left shift of one bit corresponds to a multiply-by-two operation and a right shift of one bit corresponds to a divide-by-two operation.

An n-bit register requires n storage elements, either *SR* latches, *D* flip-flops, or *JK* flip-flops. There are 2^n different states in an n-bit register, where each n-tuple corresponds to a unique state of the register.

Parallel-in, parallel-out registers The simplest register, and the most prevalent, is the *parallel-in, parallel-out* (PIPO) register used for temporary storage of binary data. The synthesis procedure is not required for this type of register. Figure 7.44 illustrates a p-bit register containing only storage elements. There is a one-to-one correspondence between the input alphabet X, the state alphabet Y, and the output alphabet Z. The values of the present inputs $X_{i(t)}$ become the next state $Y_{k(t+1)}$ of the register at the next active clock transition.

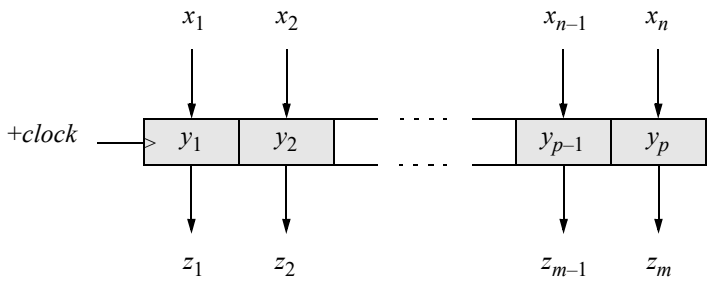


Figure 7.44 Block diagram for a parallel-in, parallel-out register.

A typical application for a PIPO register is for a general-purpose register (GPR) in an arithmetic and logic unit (ALU). General-purpose registers are used for temporary storage of data and for storing a base address or an index to be utilized in determining a memory location. They are also used as a memory address register (MAR) to address memory and as a memory data register (MDR) to contain a word of information that is sent to or received from memory.

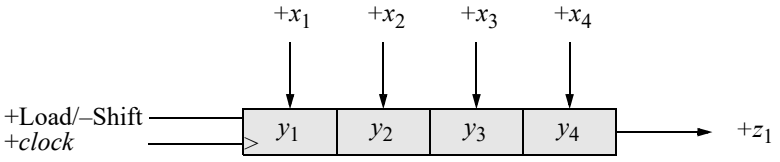
An alternative approach to a PIPO is where the storage elements are *JK* flip-flops and the δ next-state logic not only provides the next state for the register by gating the input vector X_i, but also controls the time at which the register changes state. In this design, the register is clocked continuously by the system clock, which is a free-running astable multivibrator. The register is loaded, however, only when the *load* control signal is active. When the *load* input is inactive, the data inputs of each flip-flop

are $JK = 00$, which causes no change to the state of the machine. Thus, the register remains in its present state until the *load* input changes to an active level. The new input vector X_i then replaces the previous state of the register.

Parallel-in, serial-out registers A *parallel-in, serial-out* (PISO) register accepts binary input data in parallel and generates binary output data in serial form. The binary data can be shifted either left or right under control of a shift direction signal and a clock pulse, which is applied to all flip-flops simultaneously. The register shifts left or right 1 bit position at each active clock transition. Bits shifted out of one end of the register are lost unless the register is cyclic, in which case, the bits are shifted (or rotated) into the other end.

If the register is a PISO right-shift device, then two conditions determine the value of the bits shifted into the vacated positions on the left. If the binary data represents an unsigned number, then 0s are shifted into the vacated positions. If the binary data represents a signed number — with the high-order bit specified as the sign of the number, where a 0 bit represents a positive number and a 1 bit represents a negative number — then the sign bit extends right 1 bit position for each active clock transition.

The procedure for the synthesis of a PISO register is relatively straightforward and the formalized method can be circumvented. An example is a 4-bit register that receives a parallel input vector in the form of binary bits x_1, x_2, x_3, and x_4. This operand is stored in four flip-flops y_1, y_2, y_3, and y_4, as shown in Figure 7.45. Upon application of a clock signal, the operand shifts right 1 bit position. The serial output z_1 is generated from the output of flip-flop y_4. Zeroes fill the vacated positions on the left.



Load	x_1	x_2	x_3	x_4
Shift right 1	0	x_1	x_2	x_3
Shift right 2	0	0	x_1	x_2
Shift right 3	0	0	0	x_1
Shift right 4	0	0	0	0

Figure 7.45 Block diagram for a 4-bit parallel-in, serial-out shift register.

Upon completion of the load cycle, $y_i = x_i$. During the shift sequence, $y_i = y_{i-1}$ or 0, depending on the shift count. After four shift cycles, the state of the register is $y_1y_2y_3y_4 = 0000$, and the process repeats with a new input vector X_i. One application of a PISO register is to convert data from a parallel bus into serial data for use by a

single-track device, such as a disk drive. The serialization process occurs during a write operation.

Serial-in, parallel-out registers The *serial-in, parallel-out* (SIPO) register is another typical synchronous iterative network containing p identical cells. Data enters the register from the left and shifts serially to the right through all p stages, 1 bit position per clock pulse. After p shifts, the register is fully loaded and the bits are transferred in parallel to the destination.

An example of a 4-bit SIPO register is shown in Figure 7.46, in which four bits of serial data, x_1, x_2, x_3, and x_4 are shifted into the register from the left, where x_4 is the first bit entered. The initial state of the register is either unknown or reset to $y_1y_2y_3y_4 = 0000$. During the shift sequence, $y_1 = x_i$ and $y_i = y_{i-1}$. After four shift cycles, the state of the register is $y_1y_2y_3y_4 = x_1x_2x_3x_4$ and the 4-bit word is transferred in parallel to a destination.

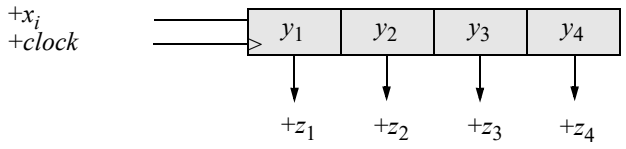


Figure 7.46 Block diagram for a 4-bit serial-in, parallel-out register.

A typical application of a serial-in, parallel-out register is to deserialize binary data from a single-track peripheral subsystem. The resulting word of parallel bits is placed on the system data bus.

A second useful application of a SIPO register is to generate a sequence of nonoverlapping pulses for system timing. This provides a simple, yet effective state machine, where each pulse represents a different state. The machine is initially reset to $y_1y_2y_3y_4 = 0000$. Whenever $y_1y_2y_3 = 000$, a 1 bit will be shifted into flip-flop y_1 at the next positive clock transition. If either y_1, y_2, or $y_3 = 1$, then a 0 bit will be shifted into flip-flop y_1, and $y_i = y_{i-1}$ at the next positive clock transition. Thus, the required four nonoverlapping pulses are generated.

Serial-in, serial-out registers The synthesis of a *serial-in, serial-out* (SISO) register is identical to that of a SIPO register, with the exception that only one output is required. The rightmost flip-flop provides the single output for the register as shown in Figure 7.47.

An important application of a SISO register is to deserialize data from a disk drive. A serial bit stream is read from a disk drive and converted into parallel bits by means of a SIPO register. When 8 bits have been shifted into the register, the bytes are shifted in parallel into a matrix of SISO registers, where each bit is shifted into a particular column. The SISO register, in this application, performs the function of a first-in,

first-out (FIFO) queue and acts as a buffer between the disk drive and the system input/output (I/O) data bus.

Information is read from a disk sector into the FIFO and then transferred to a destination by means of the data bus. The destination may be a CPU register or a storage location if direct-memory access is implemented. The mode of transfer is bit parallel, byte serial and is either synchronous, where the transfer rate is determined by a system clock, or asynchronous, where the transfer rate is determined by the disk control unit.

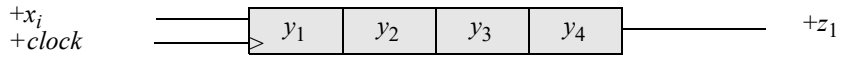


Figure 7.47 Block diagram for a 4-bit serial-in, serial-out register.

The mode of data transfer between a disk subsystem and a destination is usually in *burst mod*e, in which the disk control unit remains logically connected to the system bus for the entire data transfer sequence; that is, until the complete sector has been transferred. Some systems, however, do not allow burst mode transfer, because this would prevent other peripheral devices from gaining control of the bus. This presents no problem when the disk control unit contains a FIFO. In this situation, data continues to be read from the disk and is transferred to the FIFO, where the bytes are retained until the disk control unit again gains control of the bus. The FIFO prevents data from being lost while the control unit is arbitrating for bus control.

The same implementation of a SISO register matrix can be used as an instruction queue in a CPU instruction pipeline. The CPU prefetches instructions from memory during unused memory cycles and stores the instructions in the FIFO queue. Thus, an instruction stream can be placed in the instruction queue to wait for decoding and execution by the processor. Instruction queueing provides an effective method to increase system throughput.

7.2.3 Synchronous Counters

Counters are essential devices used in the design of digital systems. Counters have a finite number of states and represent simple Moore machines in most cases. The λ output logic is usually a function of the present state only; that is, $\lambda(Y_{j(t)})$. The state of the counter is interpreted as an integer with respect to a modulus. A number A modulo n is defined as the remainder after dividing A by n. Some counters contain a set of binary input variables from which the counter achieves an initial state.

A clock input signal causes the counter flip-flops to react only at selected discrete intervals of time; in some cases, the clock input occurs randomly. Using the clock pulses to initiate state changes, the machine counts in either an ascending or descending sequence of states. Other counters have no inputs except a clock pulse and are

usually reset to an initial state of $y_1y_2 \ldots y_p = 00 \ldots 0$. In general, a p-stage counter counts modulo 2^p.

This section will discuss only synchronous counters; asynchronous counters are inherently slow, because of the ripple effect caused by the output of stage y_i functioning as the clock input for stage y_{i+1}. The maximum time for an asynchronous counter to change state occurs when all flip-flops are set and the count increments from $2^p - 1$ to zero.

The synchronous sequential machines in this section are associated with a set of transformations on a set of states and follow a prescribed sequence of states under control of a clock input signal. When the active clock transition occurs at the input, the state of the machine changes to some predetermined value as defined by the machine specifications. Counting sequentially is completely arbitrary, although the next state is usually an increment or decrement by one, or a state in which only one flip-flop changes state, as in a Gray code counter.

Modulo-10 counter Modulo-10 counters are extensively used in digital computers when counting is required in radix 10. A modulo-10, or binary-coded decimal (BCD) decade counter, generates ten states in the following sequence: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 0000, ... Thus, each decade requires four flip-flops.

The synthesis of a modulo-10 counter is relatively straightforward. The counter is initially reset to $y_1y_2y_3y_4 = 0000$, then increments by one at each active clock transition until a state code of $y_1y_2y_3y_4 = 1001$ is reached. At the next active clock transition, the counter sequences to state $y_1y_2y_3y_4 = 0000$.

There are, however, six unused states, 1010 through 1111, that represent invalid numbers for BCD. These unused states can be regarded as "don't care" states for the purpose of minimizing the δ next-state logic, unless the counter is self-starting, in which case, all unused states contain entries which cause the counter to proceed to a predetermined state at the next active clock transition. The synthesis procedure begins below, using D flip-flops.

The state diagram for the modulo-10 counter is shown in Figure 7.48. The counter will not be self-starting; that is, all invalid BCD states will be considered as "don't care" or unused states. It is unlikely that the machine will enter an unused state; however, the possibility does exist. Digital systems enter unused states only under adverse environmental conditions such as, electrical noise, power supply voltage outside the specified operating range, or a hardware malfunction. If any of these situations occur, then the performance of the entire system is in jeopardy, not just the counter. The unused states correspond to minterms 10 through 15.

Every state in Figure 7.48 is unique. The outputs correspond to the state code of the individual states for this type of Moore machine. Therefore, no equivalent states exist. The state codes are assigned in sequence, $y_1y_2y_3y_4 = 0000$ through 1001 for the valid digits. The invalid digits of $y_1y_2y_3y_4 = 1010$ through 1111 constitute unused states.

The next-state table for this simple modulo-10 counter is not necessary — the state diagram is sufficient to obtain the input maps.

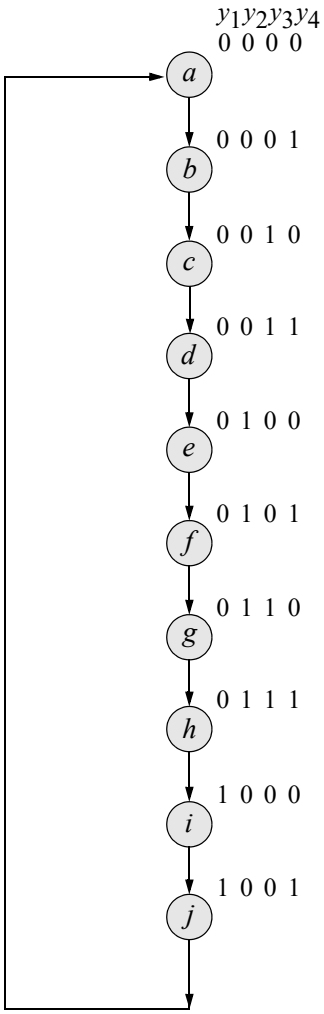


Figure 7.48 State diagram for a modulo-10 counter.

The input maps of Figure 7.49 can be derived either from the next-state table or from the state diagram. For example, using the state diagram, the entry for Dy_1 for location $y_1y_2y_3y_4 = 0111$ is obtained as follows: Flip-flop y_1 changes from 0 to 1 as the machine progresses from state *h* to state *i*. In the map for Dy_2 in minterm location $y_1y_2y_3y_4 = 0011$, a 1 is entered because flip-flop y_2 changes from 0 to 1 as the machine progresses from state *d* to state *e*. Four input maps are necessary, one for each *D* flip-flop. The input equations are presented in Equation 7.21.

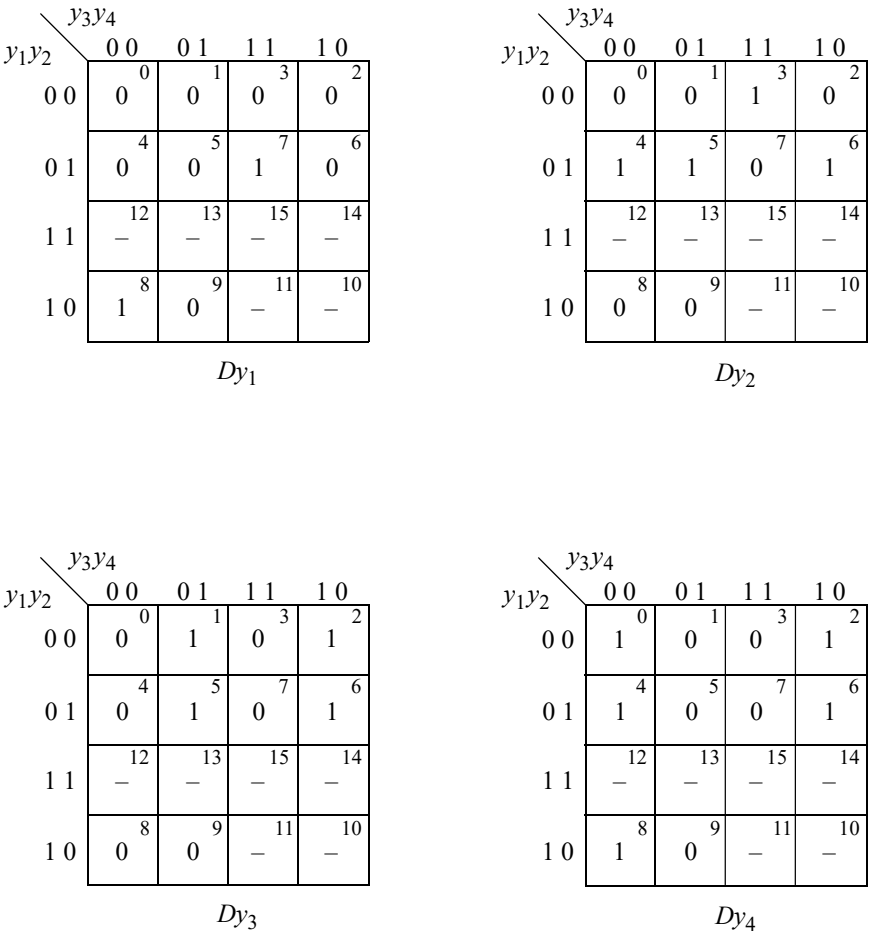


Figure 7.49 Input maps for the modulo-10 counter of Figure 7.48.

$$
\begin{aligned}
Dy_1 &= y_1y_4' + y_2y_3y_4 \\
Dy_2 &= y_2y_3' + y_2y_4' + y_2'y_3y_4 \\
Dy_3 &= y_1'y_3'y_4 + y_3y_4' \\
Dy_4 &= y_4'
\end{aligned}
\tag{7.21}
$$

No output maps are required for a single 4-bit modulo-10 counter. If, however, the counter is one decade of a multi-decade counter, then λ output logic is necessary to indicate when decade$_i$ has attained a count of $y_1y_2y_3y_4 = 1001$. The next active clock transition will reset decade$_i$ and increment by one the next higher-order decade$_{i+1}$. The design of an n-digit BCD counter is constructed from n 4-bit modulo-10 counters.

The logic diagram of Figure 7.50 is derived from the D flip-flop input equations of Equation 7.21. The counter is reset initially to $y_1y_2y_3y_4 = 0000$. The timing diagram is illustrated in Figure 7.51. The clock signal is supplied to all flip-flops simultaneously. State changes occur only on the positive clock transition. Using either Equation 7.21 or the logic diagram of Figure 7.50, the counter can be shown to increment through the modulo-10 counting sequence, then return to 0000 at the next positive clock transition.

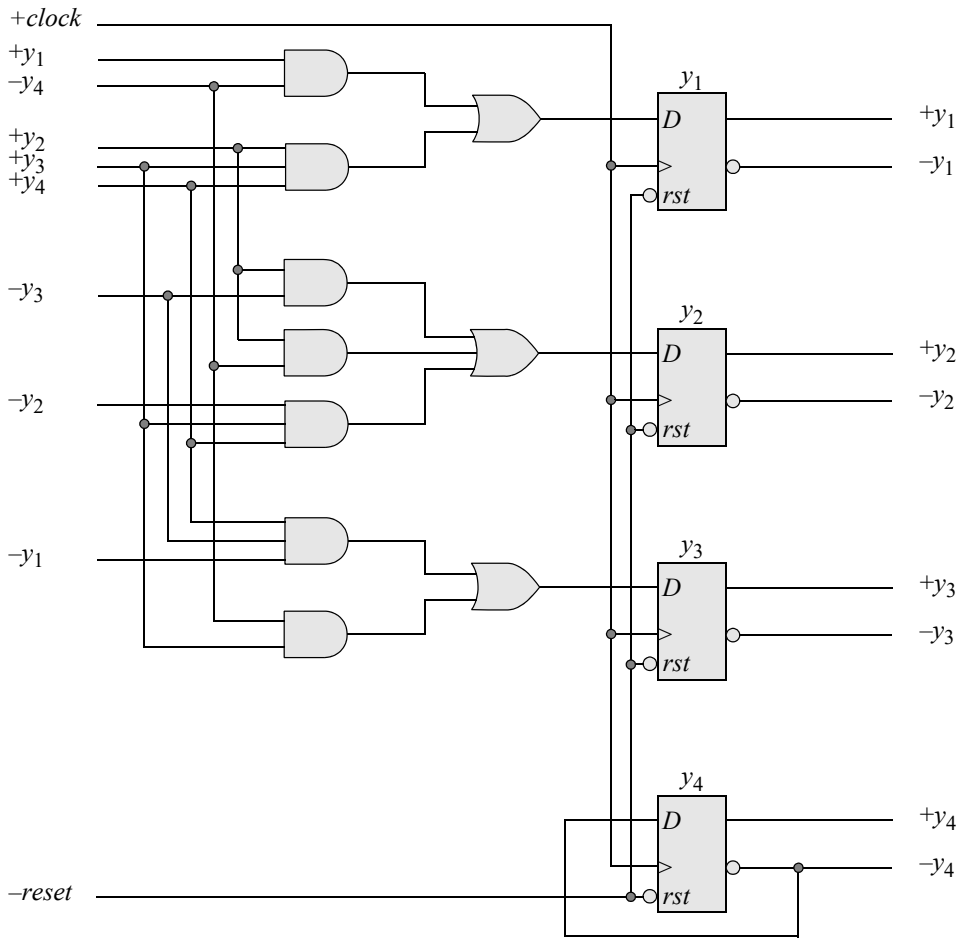


Figure 7.50 Logic diagram for the modulo-10 counter of Figure 7.48, where y_4 is the low-order flip-flop.

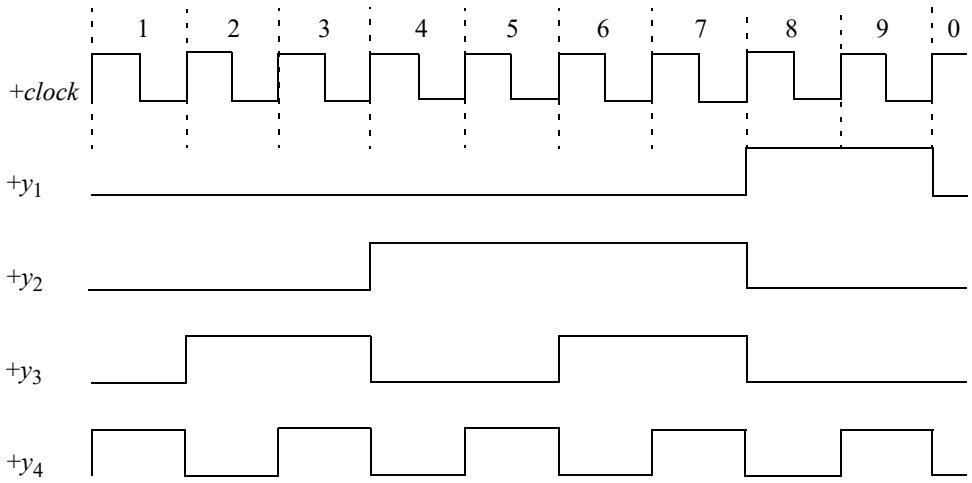


Figure 7.51 Timing diagram for the modulo-10 counter of Figure 7.50.

Johnson counter The modulo-10 counter previously described had a counting sequence that increased in a binary manner from zero to nine; however, it used only ten combinations of 16 possible states. Other counters are frequently utilized in digital computers. These are down-counters that count in a descending sequence from some preset state to zero. The state diagram contains state codes that decrement by one for each succeeding state. The synthesis of these counters is identical to the method previously presented.

Still other counters can be designed for a unique application in which the counting sequence is neither entirely up nor entirely down. These have a nonsequential counting sequence that is prescribed by external requirements. Such a counter is shown in the state diagram of Figure 7.52, in which the counting sequence is $y_1y_2y_3$ = 000, 100, 110, 111, 011, 001, 000, ... The counter is reset initially to $y_1y_2y_3$ = 000. For six of the eight possible states for three variables, the state transitions are completely defined. The remaining two states are unspecified and can be regarded as "don't care" states in order to minimize the δ next-state logic. This presents no problem under normal operating conditions where the environment is free from electrical interference.

The counter of Figure 7.52 represents a *Johnson counter* in which any two contiguous state codes (or code words) differ by only one variable. The Johnson counter is also referred to as a "Möbius counter," because the output of the last stage is inverted and fed back to the first stage. August F. Möbius was a German mathematician who discovered a one-sided surface that is constructed from a rectangle by holding one end fixed, rotating the opposite end through 180 degrees, and applying it to the first end.

It is similar, in this respect, to a Gray code counter. The Gray code concept is used in Karnaugh maps. Any physically adjacent minterm locations are also logically adjacent because they differ in only one variable and, therefore, can be combined into a

term with fewer variables. Contiguous code words that are logically adjacent is an important consideration in eliminating output glitches and will be elaborated in detail in a later section.

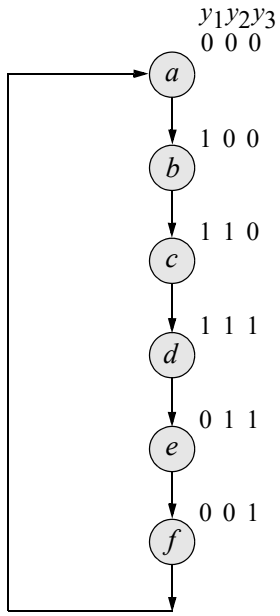
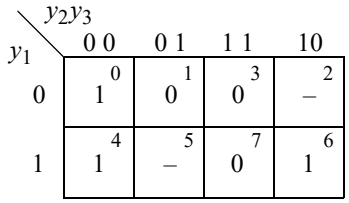


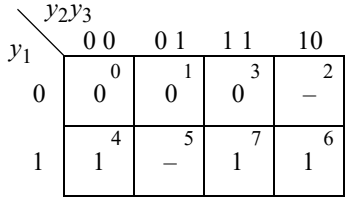
Figure 7.52 State diagram for a Johnson counter with a nonsequential counting sequence. There are two unused states: $y_1y_2y_3 = 010$ and 101.

Using the synthesis procedure previously described, the input maps are obtained using *D* flip-flops, as shown in Figure 7.53. The maps can be derived directly from the state diagram without the necessity of generating a next-state table. For example, from state *b* ($y_1y_2y_3 = 100$), the machine sequences to state *c* ($y_1y_2y_3 = 110$) where the next state for flip-flop y_1 is 1. Thus, a 1 is entered in minterm location $y_1y_2y_3 = 100$ for flip-flop y_1.

Likewise, from state *c* the machine proceeds to state *d* where the next state for y_1 is 1; therefore, a 1 is entered in minterm location $y_1y_2y_3 = 110$ for flip-flop y_1. In a similar manner, the remaining entries are obtained for the input map for y_1, as well as for the input maps for y_2 and y_3. The input equations are listed in Equation 7.22.

The logic diagram is shown in Figure 7.54. The counting sequence is easily verified by asserting the appropriate input logic levels to the flip-flop *D* inputs for each state of the counter and then applying the active clock transition. The timing diagram is shown in Figure 7.55.





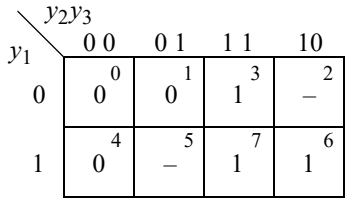


Figure 7.53 Input maps for the Johnson counter of Figure 7.52 using *D* flip-flops. The unused states are $y_1y_2y_3 = 010$ and 101.

$$Dy_1 = y_3'$$
$$Dy_2 = y_1$$
$$Dy_3 = y_2 \tag{7.22}$$

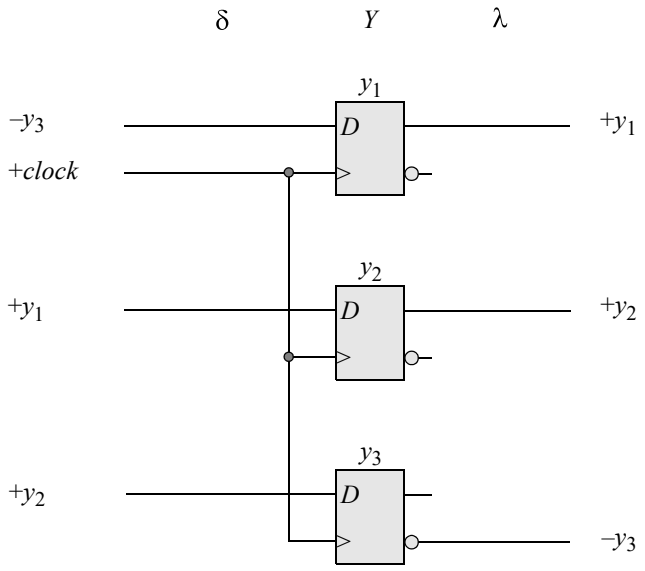


Figure 7.54 Logic diagram for the Johnson counter of Figure 7.52 using *D* flip-flops. The counting order is nonsequential. Flip-flop y_3 is the low-order stage.

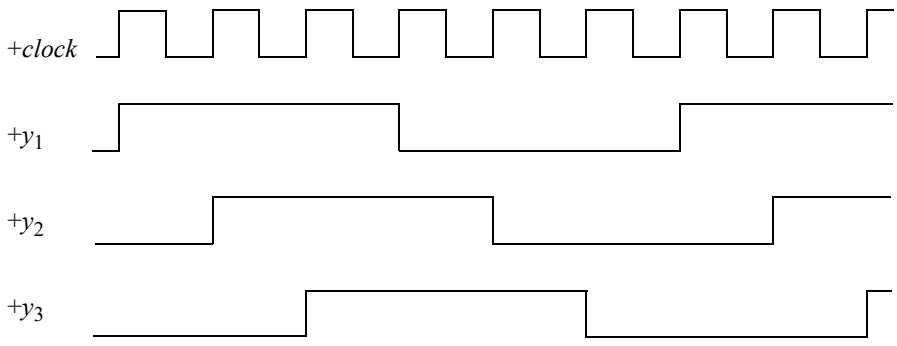


Figure 7.55 Timing diagram for the Johnson counter of Figure 7.54.

Pulse generator counter Another interesting counter is a pulse generator counter in which the counter generates a series of nonoverlapping pulses as shown in the state diagram of Figure 7.56. These four disjoint pulses are a function of the counting sequence and can be used for system timing. The counter provides a simple, yet effective state machine, where each pulse represents a different state.

The pulses can be used in a digital system to define time slots in which different operations take place. For example, a peripheral control unit may have three different time slots: one to interface with the input/output channel of the computer; one dedicated to internal control unit operations; and one to interface with the peripheral device.

The *D* flip-flop input maps are shown in Figure 7.57 in which the unused states are treated as "don't care" states. The input equations are shown in Equation 7.23, which are used to design the logic diagram of Figure 7.58. The timing diagram is shown in Figure 7.59.

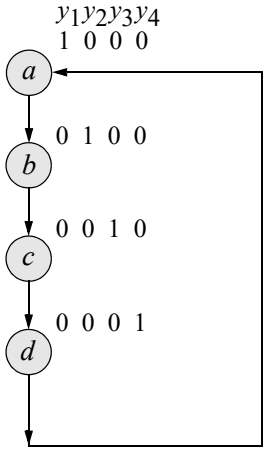


Figure 7.56 State diagram for a pulse generator counter.

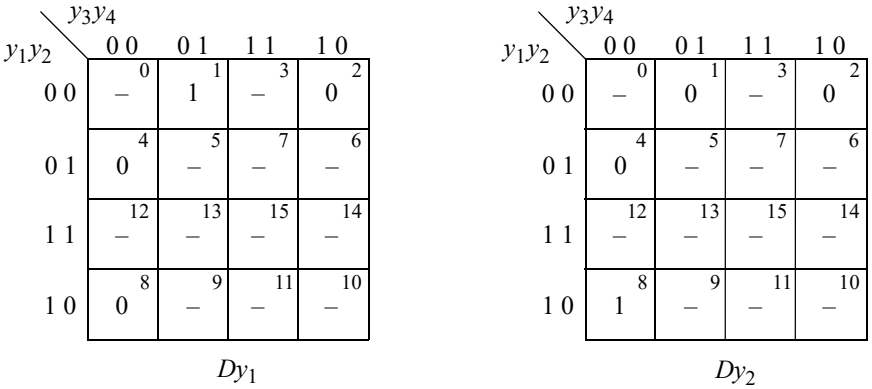


Figure 7.57 Input maps for the pulse generator counter of Figure 7.56.

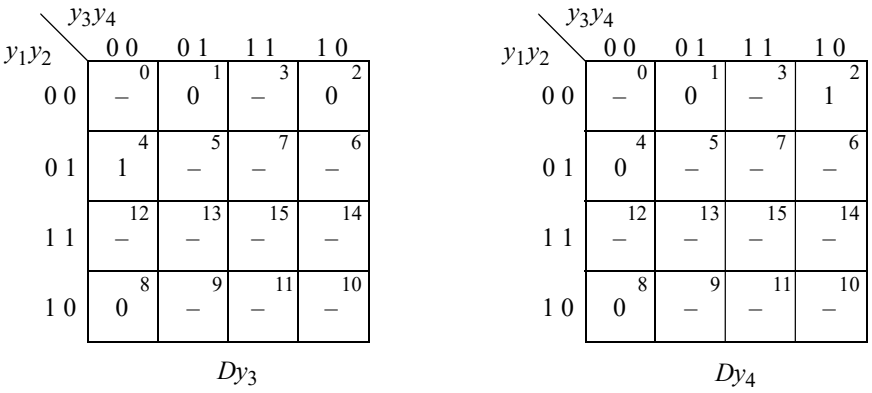


Figure 7.57 (Continued)

$$Dy_1 = y_4 \quad Dy_2 = y_1 \quad Dy_3 = y_2 \quad Dy_4 = y_3 \tag{7.23}$$

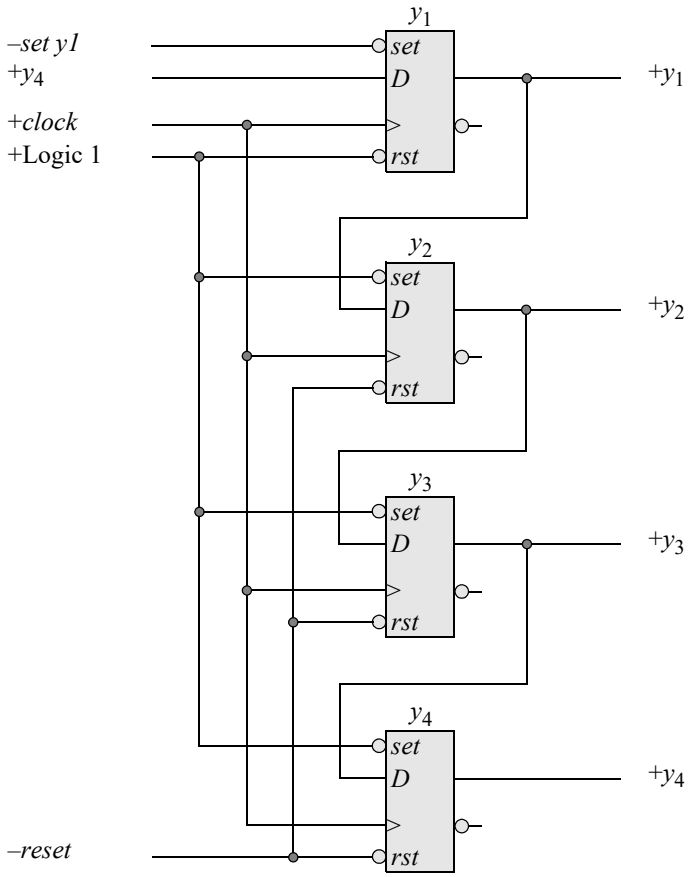


Figure 7.58 Logic diagram for the pulse generator counter of Figure 7.56.

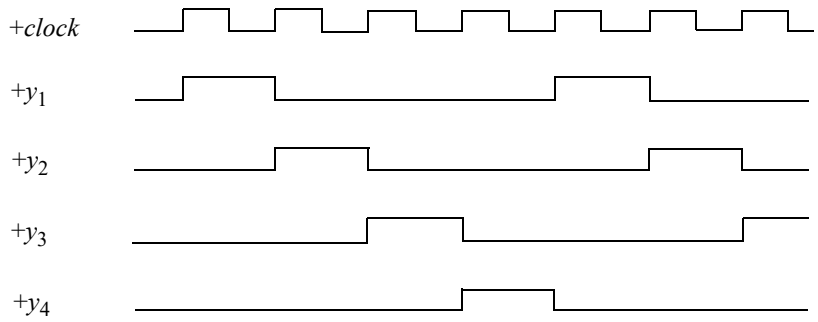


Figure 7.59 Timing diagram for the pulse generator counter of Figure 7.56.

7.2.4 Moore Machines

This section extends the concepts of Moore machines that were introduced previously and presents a procedure for synthesizing Moore machines. The primary focus of this section will be on the synthesis of *deterministic synchronous sequential machines*, in which the next state is uniquely determined by the present state $Y_{j(t)}$ and the present inputs $X_{i(t)}$.

The Moore model of sequential machines is the result of a paper by E.F. Moore in 1956. A Moore machine was formally defined in a previous section as a synchronous sequential machine characterized by the following 5-tuple:

$$M = (X,\ Y,\ Z,\ \delta,\ \lambda)$$

where X is the input alphabet, Y is the state alphabet, and Z is the output alphabet. The next-state function δ maps the Cartesian product of X and Y into Y, and thus, is determined by both the present inputs and the present state. The output function λ maps Y into Z such that the output vector is a function of the present state only and is independent of the external inputs.

Example 7.6 A Moore machine will be designed that operates according to the state diagram of Figure 7.60. There are three inputs x_1, x_2, and x_3; three *JK* state flip-flops y_1, y_2, and y_3; and two outputs z_1 that is asserted in state e ($y_1y_2y_3 = 111$) and z_2 that is asserted in state f ($y_1y_2y_3 = 110$).

The *JK* flip-flop functional characteristics are shown in Table 7.17 and the excitation table in Table 7.18. The next-state table is shown in Table 7.19 and is generated directly from the state diagram. In state a ($y_1y_2y_3 = 000$), only x_1 is a contributing factor; in state b ($y_1y_2y_3 = 001$), x_2 is the only active input, and in state c ($y_1y_2y_3 = 010$), x_3 is the only active input. Therefore, some inputs are indicated as "don't care" inputs in the next-state table.

For example, consider the present state a ($y_1y_2y_3 = 000$). The next state for flip-flop y_1 is always $y_1 = 0$, regardless of the value of x_1. Therefore, the state transition sequence for y_1 is $0 \rightarrow 0$, yielding $Jy_1 = 0$ and $Ky_1 = –$ from Table 7.18.

In state a ($y_1y_2y_3 = 000$), if $x_1 = 1$, flip-flop y_2 has a next value of $y_2 = 0$ when the machine sequences to state b ($y_1y_2y_3 = 001$). Therefore, flip-flop y_2 sequences from $0 \rightarrow 0$. If $x_1 = 0$ in state a, flip-flop y_2 has a next value of $y_2 = 1$ when the machine sequences to state c ($y_1y_2y_3 = 010$); therefore, the state transition sequence is $0 \rightarrow 1$. From Table 7.18, the input values are $Jy_2 = x_1'$ and $Ky_2 = –$.

In state a ($y_1y_2y_3 = 000$), if $x_1 = 1$, flip-flop y_3 has a next value of 1 when the machine sequences to state b ($y_1y_2y_3 = 001$). Therefore, the state transition sequence for flip-flop y_3 is $0 \rightarrow 1$. If $x_1 = 0$ in state a ($y_1y_2y_3 = 000$), then flip-flop y_3 has a next value of 0 when the machine sequences to state c ($y_1y_2y_3 = 010$), providing a state transition sequence of $0 \rightarrow 0$. From Table 7.18, this yields input values $Jy_3 = x_1$ and $Ky_3 = –$. In a similar manner, the remaining entries are determined for the next-state table.

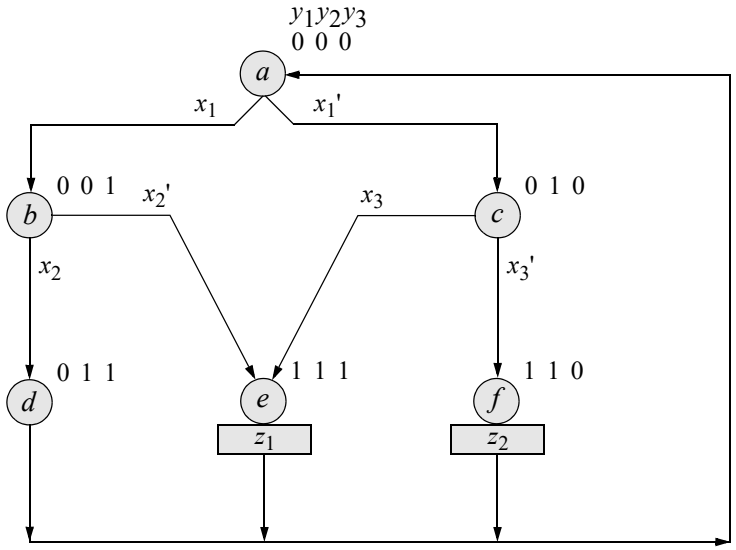


Figure 7.60 State diagram for the Moore machine of Example 7.6.

Table 7.17 *JK* Flip-Flop Functional Characteristics

J	*K*	Function
0	0	No change
0	1	Reset
1	0	Set
1	1	Toggle

Table 7.18 Excitation Table for a *JK* Flip-Flop

Present State $Y_{j(t)}$	Next State $Y_{k(t+1)}$	Data Inputs J K
0	0	0 –
0	1	1 –
1	0	– 1
1	1	– 0

Table 7.19 Next-State Table for the Moore Machine of Example 7.6

Present State			Inputs			Next State			Flip-Flop Inputs						Outputs	
y_1	y_2	y_3	x_1	x_2	x_3	y_1	y_2	y_3	Jy_1	Ky_1	Jy_2	Ky_2	Jy_3	Ky_3	z_1	z_2
0	0	0	0	–	–	0	1	0	0	–	1	–	0	–	0	0
0	0	0	1	–	–	0	0	1	0	–	0	–	1	–	0	0
0	0	1	–	0	–	1	1	1	1	–	1	–	–	0	0	0
0	0	1	–	1	–	0	1	1	0	–	1	–	–	0	0	0
0	1	0	–	–	0	1	1	0	1	–	–	0	0	–	0	0
0	1	0	–	–	1	1	1	1	1	–	–	0	1	–	0	0
0	1	1	–	–	–	0	0	0	0	–	–	1	–	1	0	0
1	0	0	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	0	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	1	0	–	–	–	0	0	0	–	1	–	1	0	–	0	1
1	1	1	–	–	–	0	0	0	–	1	–	1	–	1	1	0

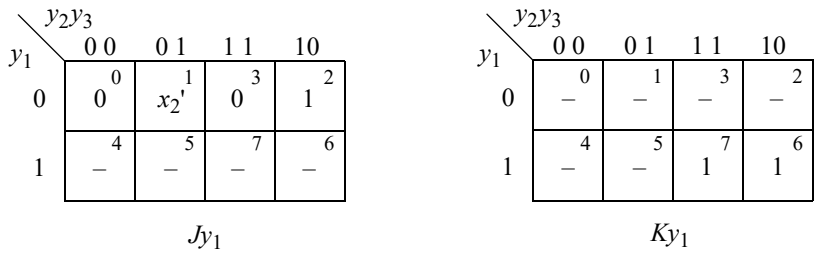


Figure 7.61 Input maps for the Moore machine of Example 7.6.

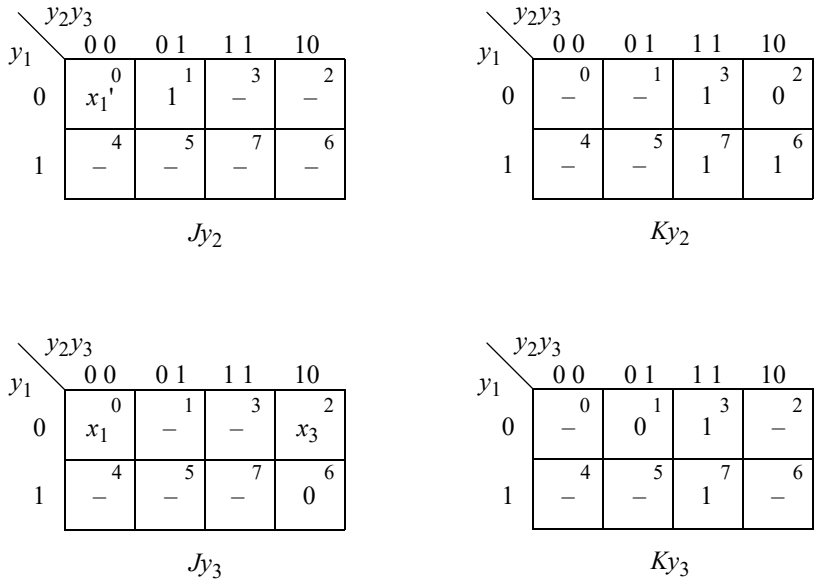


Figure 7.61 (Continued)

The input equations are listed in Equation 7.24 as obtained from the input maps. Consider the map for Jy_1. Minterm locations 1 and 5 combine to yield the term $y_2'y_3x_2'$; minterm locations 2 and 6 combine to yield y_2y_3'. The map for Ky_1 yields the term $Ky_1 = 1$, because every location contains either an entry of 1 or "don't care."

Consider the map for Jy_2. Every location contains either x_1' or a "don't care," because the entry of 1 in minterm location $1 = x_1 + x_1'$. Therefore, the first term for Jy_2 is x_1'. Now the entry of 1 in minterm location 1 combines with minterm locations 3, 5, and 7 to yield y_3. The equation for Ky_2 is $Ky_2 = y_3 + y_1$.

Consider the map for Jy_3. Minterm locations 0, 1, 4, and 5 combine to yield $y_2'x_1$. Minterms locations 2 and 3 combine to yield $y_1'y_2x_3$. The equation for Ky_3 is $Ky_3 = y_2$. The logic diagram is shown in Figure 7.62 and is designed using the equations of Equation 7.24.

$$
\begin{aligned}
Jy_1 &= y_2'y_3x_2' + y_2y_3' & Ky_1 &= 1 \\
Jy_2 &= x_1' + y_3 & Ky_2 &= y_3 + y_1 \\
Jy_3 &= y_2'x_1 + y_1'y_2x_3 & Ky_3 &= y_2
\end{aligned} \tag{7.24}
$$

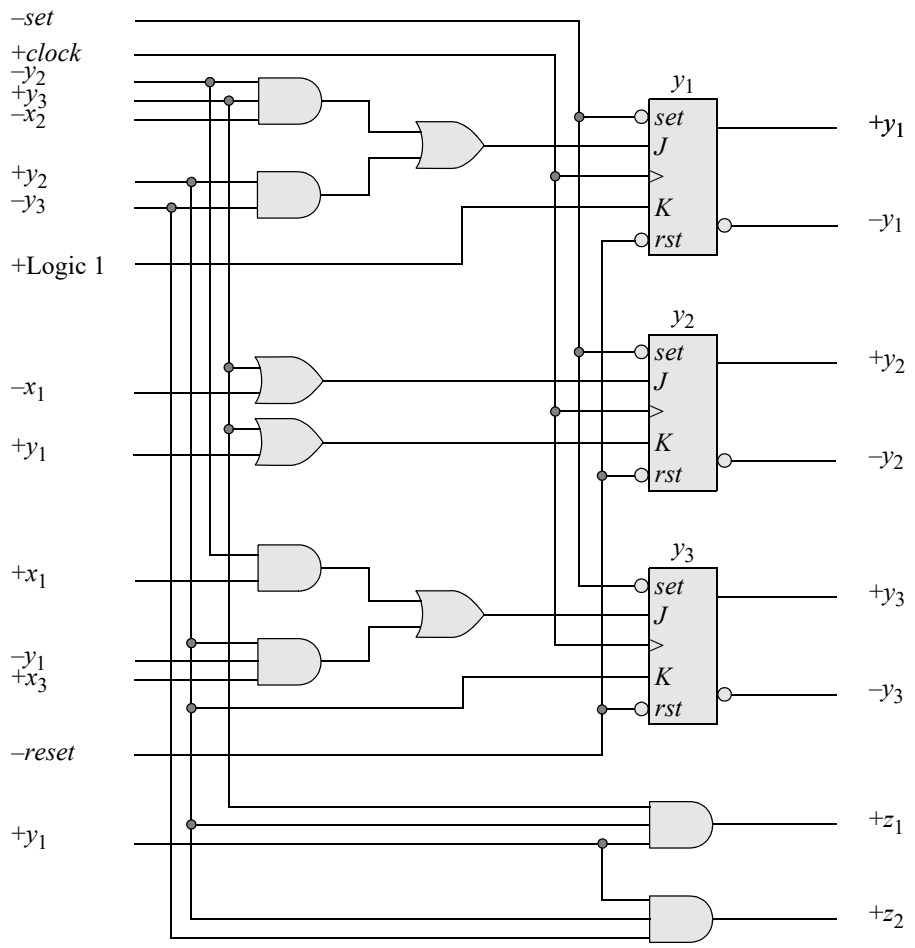


Figure 7.62 Logic diagram for the Moore machine of Example 7.6.

Example 7.7 A Moore machine will be designed that operates according to the state diagram of Figure 7.63. There are two inputs x_1, and x_2; three D state flip-flops y_1, y_2, and y_3; and three outputs z_1 that is asserted in state a ($y_1y_2y_3 = 000$), z_2 that is asserted in state c ($y_1y_2y_3 = 010$), and z_3 that is asserted in state d ($y_1y_2y_3 = 100$). There are three unused states: $y_1y_2y_3 = 001$, 101, and 110.

State codes were deliberately inserted that would cause glitches on the outputs; therefore, all inputs are asserted at time t_2 and deasserted at time t_3. This ensures that there will be no glitches on the outputs for any state transition sequence. For example, for a state transition from state b ($y_1y_2y_3 = 011$) to state d ($y_1y_2y_3 = 100$), all flip-flops change state. Therefore, all flip-flops could conceivably momentarily pass through state a ($y_1y_2y_3 = 000$) — depending on the propagation delays of the flip-flops — and assert output z_1.

It is possible that there could be three glitches on output z_2 for the following state transition sequences: state a ($y_1y_2y_3 = 000$) to state b ($y_1y_2y_3 = 011$) if y_2 sets before y_3 sets; from state b ($y_1y_2y_3 = 011$) to state d ($y_1y_2y_3 = 100$) if y_3 resets before y_1 and y_2 change state; and from state e ($y_1y_2y_3 = 111$) to state a ($y_1y_2y_3 = 000$) if y_1 and y_3 reset before y_2 resets. Output z_3 may produce a glitch when the machine sequences from state e ($y_1y_2y_3 = 111$) to state a ($y_1y_2y_3 = 000$) if y_2 and y_3 change state before y_1 changes state.

Asserting all outputs at time t_2 and deasserting all outputs at time t_3 negates the possibility of a glitch. Output glitches can occur at the active transition of the clock when the flip-flops are changing state. This is a very small interval of time and is referred to as the Δt time.

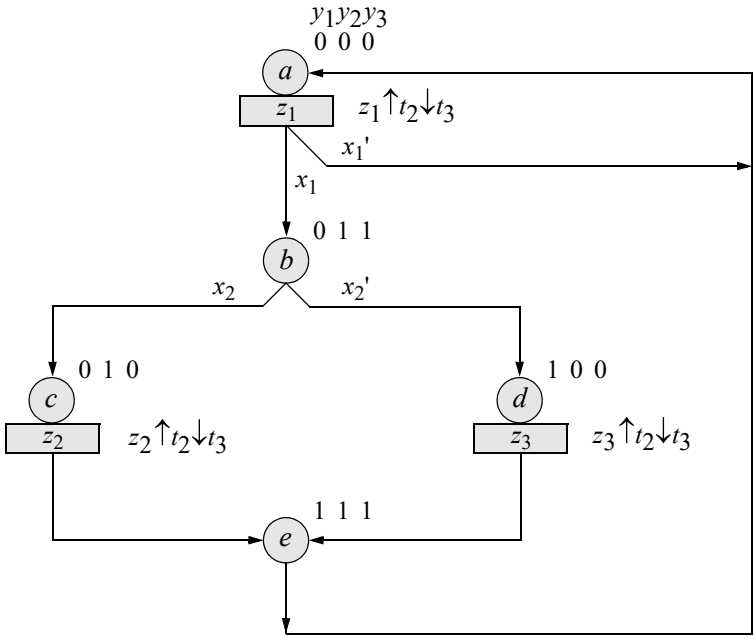


Figure 7.63 State diagram for the Moore machine of Example 7.7.

Since D flip-flops are used, there is no need for a next-state table — the input maps can be derived directly from the state diagram and are shown in Figure 7.64. For example, in state a ($y_1y_2y_3 = 000$), flip-flop y_1 always has a next value of 0 regardless of the path taken; therefore, a 0 is inserted in minterm location 0 of the map for Dy_1. In state b ($y_1y_2y_3 = 011$), flip-flop y_1 has a next value of 1 if input x_2 is deasserted; therefore, x_2' is inserted in minterm location 3 of the map for Dy_1. In a similar manner, the

remaining entries in the map for Dy_1 and the maps for Dy_2 and Dy_3 are derived. The D input equations are shown in Equation 7.25 as derived from the input maps; the logic diagram is shown in Figure 7.65 and is designed from D the input equations.

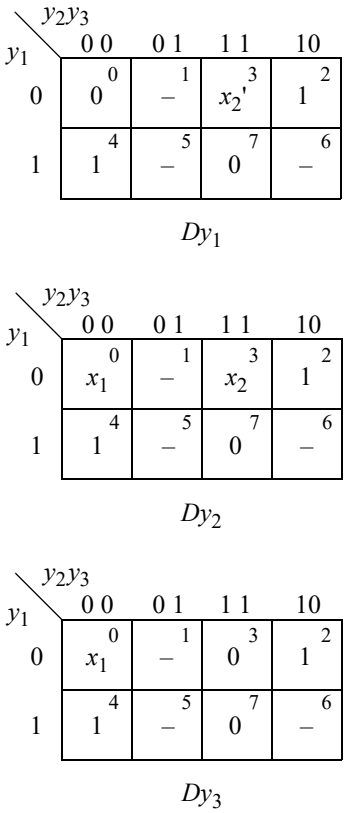


Figure 7.64 Input maps for the Moore machine of Example 7.7.

$$
\begin{aligned}
Dy_1 &= y_1'y_3x_2' + y_2y_3' + y_1y_2' \\
Dy_2 &= y_2'x_1 + y_1y_2' + y_1'y_3x_2 + y_2y_3' \\
Dy_3 &= y_2'x_1 + y_1y_2' + y_2y_3'
\end{aligned}
\tag{7.25}
$$

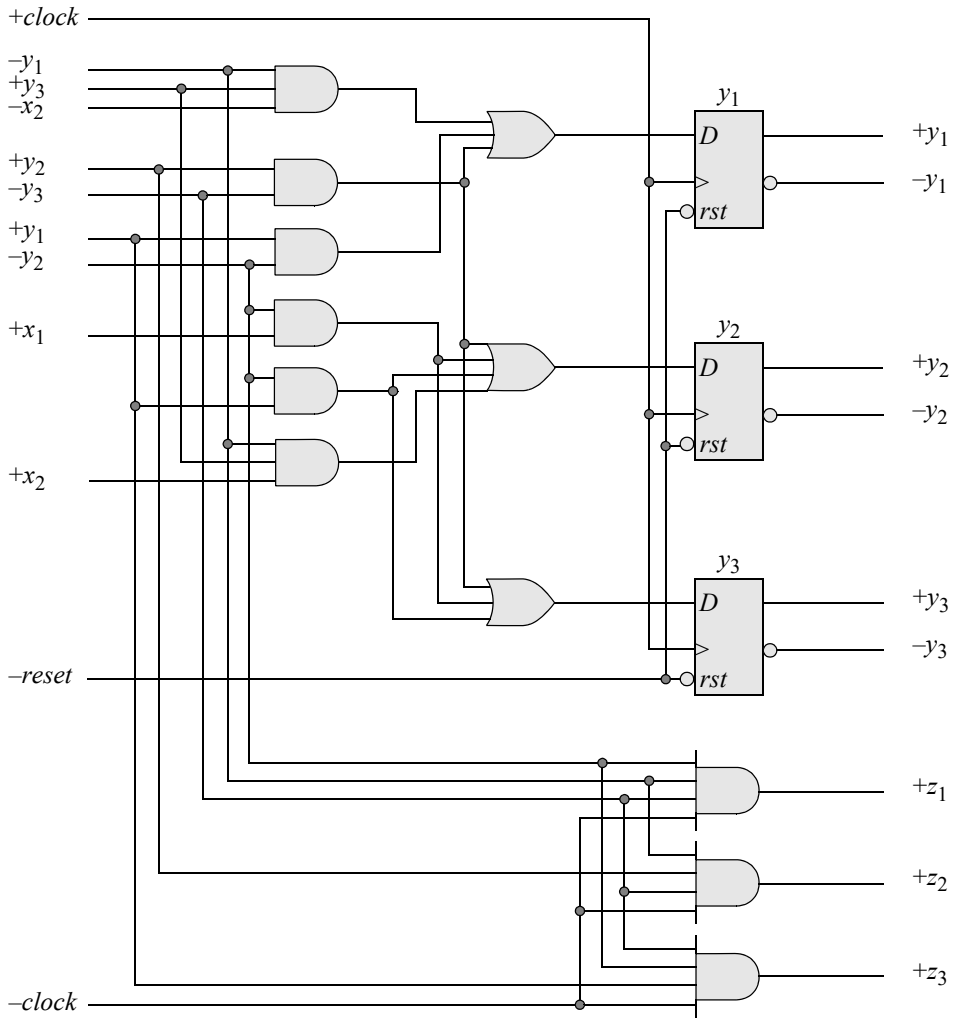


Figure 7.65 Logic diagram for the Moore machine of Example 7.7.

7.2.5 Mealy Machines

A Mealy machine is also an important class of finite-state machine and represents an alternative model that is more widely used than a Moore machine. The Mealy class of synchronous sequential machines is the result of a paper by G. H. Mealy in 1955 on the synthesis of sequential circuits. A Mealy machine is defined as a synchronous sequential machine characterized by the following 5-tuple:

$$M = (X, Y, Z, \delta, \lambda)$$

where X is the input alphabet, Y is the state alphabet, and Z is the output alphabet.

The next-state function δ maps the Cartesian product of X and Y into Y, and thus, is determined by both the present inputs and the present state. The output function λ maps the Cartesian product of X and Y into Z such that, the output vector is a function of both the present inputs and the present state.

This is the underlying difference between Moore and Mealy machines — *the outputs of a Moore machine are directly related to the present state only, whereas the outputs of a Mealy machine are a function of both the present state and the present inputs*. Examples will now be presented that illustrate the design procedure for Mealy synchronous sequential machines.

Example 7.8 A Mealy synchronous sequential machine generates an output z_1 whenever a serial data input line x_1 contains a 3-bit word with an odd number of 1s. The format for the serial data line is shown below.

$$x_1 = \left| b_1 \quad b_2 \quad b_3 \right| b_1 \quad b_2 \quad b_3 \left| b_1 \quad b_2 \quad b_3 \right| \ldots$$

where $b_i = 0$ or 1. There is no space between words. Output z_1 is asserted during the first half of the third bit time b_3; that is, $z_1 \uparrow t_1 \downarrow t_2$. The chart shown below lists the state codes in which z_1 is asserted for an odd number of 1s.

$x_1 =$	0	0	0		
	0	0	1	$\rightarrow$	z_1
	0	1	0	$\rightarrow$	z_1
	0	1	1		
	1	0	0	$\rightarrow$	z_1
	1	0	1		
	1	1	0		
	1	1	1	$\rightarrow$	z_1

The state diagram shown in Figure 7.66 in which the state codes have been selected so that there will be no glitches on output z_1 for any state transition sequence. There are three unused states: $y_1y_2y_3 = 101$, 110, and 111. If two or more flip-flops change value for a state transition sequence, then the machine may momentarily pass through either an unused state or a state in which there is no output — in both cases there will be no glitch on output z_1.

For example, for a state transition from state b ($y_1y_2y_3 = 001$) to state e ($y_1y_2y_3 = 100$), the machine may pass through state $y_1y_2y_3 = 101$ if flip-flop y_1 sets before flip-flop y_3 resets; however, state $y_1y_2y_3 = 101$ is an unused state. In a similar manner, for a transition from state c ($y_1y_2y_3 = 011$) to state a ($y_1y_2y_3 = 000$) with $x_1 = 0$, the machine may pass through state b ($y_1y_2y_3 = 001$) if y_2 resets before y_3 resets; however, state b has no output.

The *D* flip-flop input maps are shown in Figure 7.67 and the equations are shown in Equation 7.26. The map for output z_1 is shown in Figure 7.68 and the equation is shown in Equation 7.27. The logic diagram is shown in Figure 7.69.

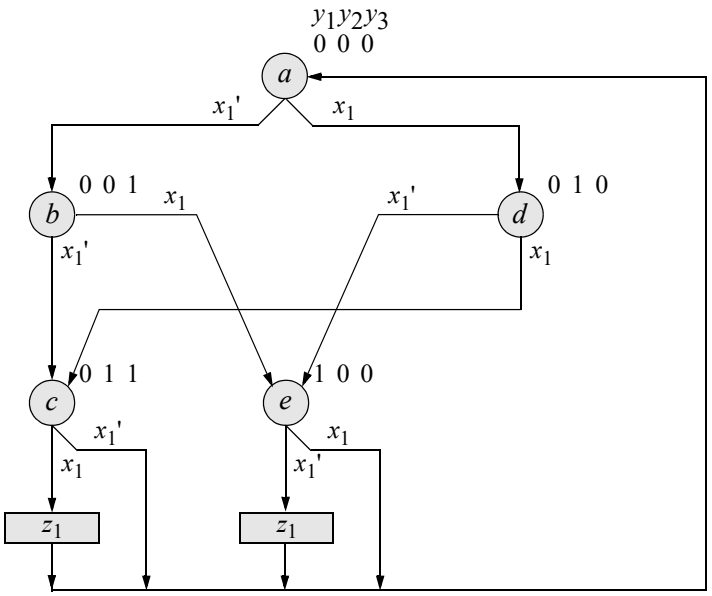


Figure 7.66 State diagram for the Mealy machine of Example 7.8.

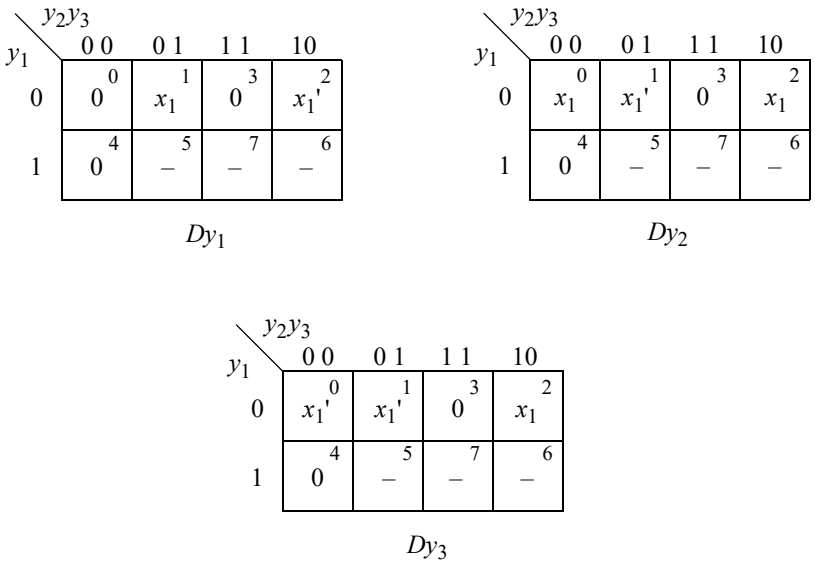


Figure 7.67 Input maps for the Mealy machine of Example 7.8.

$$Dy_1 = y_2'y_3x_1 + y_2y_3'x_1'$$
$$Dy_2 = y_1'y_3'x_1 + y_2'y_3x_1'$$
$$Dy_3 = y_1'y_2'x_1' + y_2y_3'x_1 \tag{7.26}$$

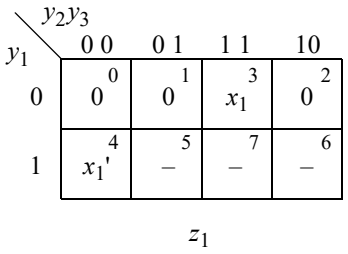


Figure 7.68 Output map for the Mealy machine of Example 7.8.

$$z_1 = y_1x_1' + y_2y_3x_1 \tag{7.27}$$

Example 7.9 The state diagram for a Mealy synchronous sequential machine is shown in Figure 7.70 and will be implemented with *JK* flip-flops. There are three inputs x_1, x_2, and x_3 and one output z_1. There are two state flip-flops y_1 and y_2 that are reset to state *a* ($y_1y_2 = 11$) and one unused state $y_1y_2 = 00$.

The next-state table is shown in Table 7.20 and is obtained directly from the state diagram. For example, consider state *a* ($y_1y_2 = 11$). Input x_3 does not contribute to a state transition to state *b* ($y_1y_2 = 10$) or to state *c* ($y_1y_2 = 01$); therefore, input x_3 is entered as a "don't care" value in the next-state table. If $x_1 = 0$ in state *a* ($y_1y_2 = 11$), then the machine proceeds to state *c* ($y_1y_2 = 01$); therefore, the next-state table contains a next state of $y_1y_2 = 01$ whenever $x_1 = 0$.

If $x_1x_2 = 10$ in state *a*, then the machine remains in state *a*. If $x_1x_2 = 11$ in state *a*, then the machine proceeds to state *b* ($y_1y_2 = 10$), where output z_1 is asserted if $x_3 = 1$, then sequences to state *a*; otherwise, the machine proceeds to state *a* without asserting output z_1.

The input maps for the *JK* flip-flops are shown in Figure 7.71 and the equations are shown in Equation 7.28. Figure 7.72 contains the Karnaugh map for output z_1 with the equation in Equation 7.29. The input and output equations are used to design the logic diagram of Figure 7.73.

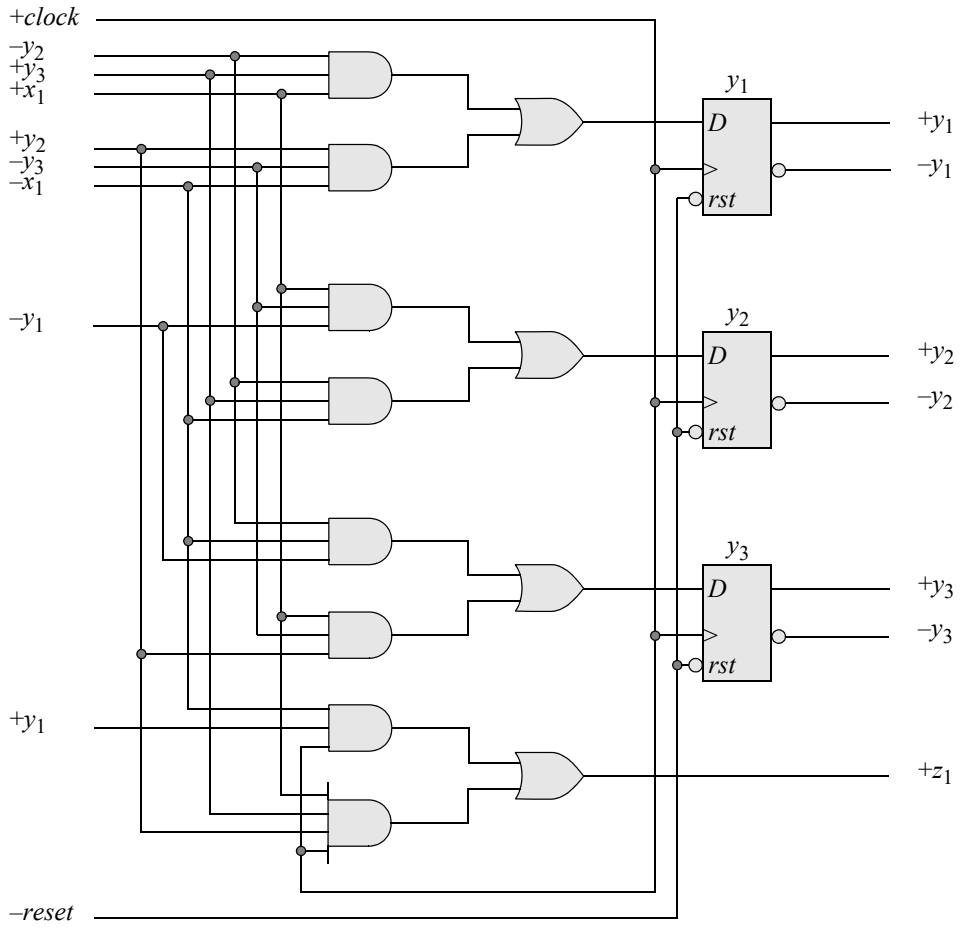


Figure 7.69 Logic diagram for the Mealy machine of Example 7.8.

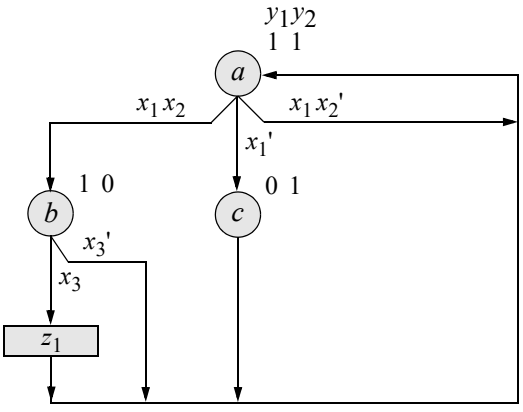


Figure 7.70 State diagram for the Mealy machine of Example 7.9.

Table 7.20 Next-State Table for the Mealy Machine of Example 7.9

Present State		Inputs			Next State		Flip-Flop Inputs				Output
y_1	y_2	x_1	x_2	x_3	y_1	y_2	Jy_1	Ky_1	Jy_2	Ky_2	z_1
0	0	–	–	–	–	–	–	–	–	–	–
0	1	–	–	–	1	1	1	–	–	0	0
1	0	–	–	0	1	1	–	0	1	–	0
		–	–	1	1	1	–	0	1	–	1
1	1	0	0	–	0	1	–	1	–	0	0
		0	1	–	0	1	–	1	–	0	0
		1	0	–	1	1	–	0	–	0	0
		1	1	–	1	0	–	0	–	1	0

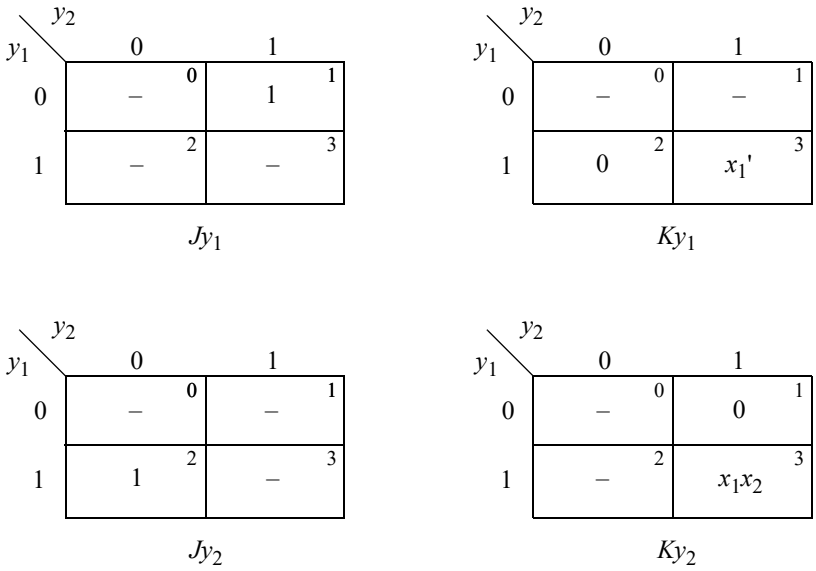


Figure 7.71 Input maps for the Mealy machine of Example 7.9.

$$Jy_1 = 1 \qquad Ky_1 = y_2x_1'$$
$$Jy_2 = 1 \qquad Ky_2 = y_1x_1x_2 \tag{7.28}$$

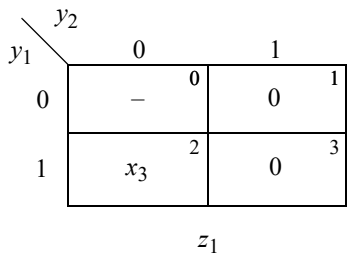


Figure 7.72 Output map for the Mealy machine of Example 7.9.

$$z_1 = y_2' x_3 \tag{7.29}$$

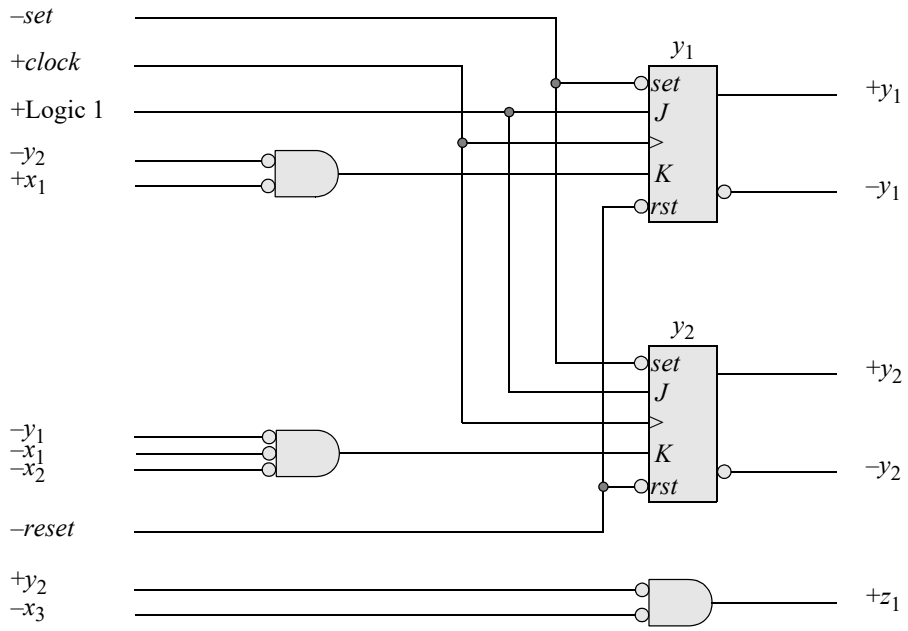


Figure 7.73 Logic diagram for the Mealy machine of Example 7.9.

7.2.6 Output Glitches

A *glitch* in synchronous sequential machines is any false or spurious electronic signal. These narrow, unwanted pulses wreak havoc in digital systems if the glitch occurs on an output signal. For example, the output signal of a logic circuit may connect to the clock input of a counter and produce erroneous results. Therefore, eliminating output glitches is extremely important, even at the expense of additional logic.

In synchronous systems, glitches can occur in the time period between the active clock transition and circuit stabilization. This is shown in Figure 7.74 and is indicated by the time duration Δt. It is during this time, when the machine is changing states, that the outputs are susceptible to glitches.

Due to varying propagation delays of the internal logic of the storage elements, the machine may enter an unstable, or transient, state. Although momentary in duration, this transient state can cause an output glitch in both Moore and Mealy machines if the output is decoded directly from the p-tuple state codes. If the outputs are enabled at time t_2, then glitches that occur during the period of instability are of no consequence — the machine has long since stabilized. Techniques will now be presented to eliminate output glitches.

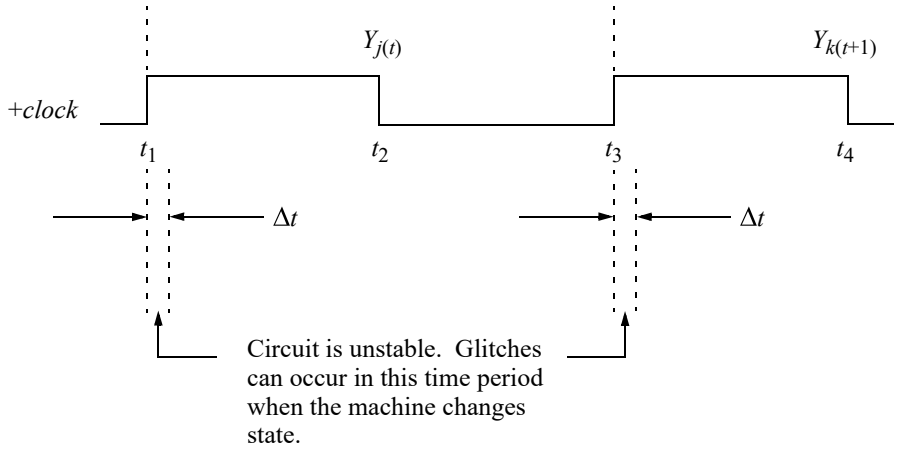


Figure 7.74 Illustration of the time intervals in which glitches are possible for synchronous sequential machines.

Glitch elimination using state code assignment It is often possible to reassign state codes to avoid output glitches. This may result in additional δ next-state logic, however, if state code adjacency cannot be maintained. Whenever possible, state codes should be assigned such that there are a maximal number of adjacent 1s in the flip-flop input maps. This allows more minterm locations to be combined, resulting in minimized input equations in a sum-of-products form. State codes are adjacent if they

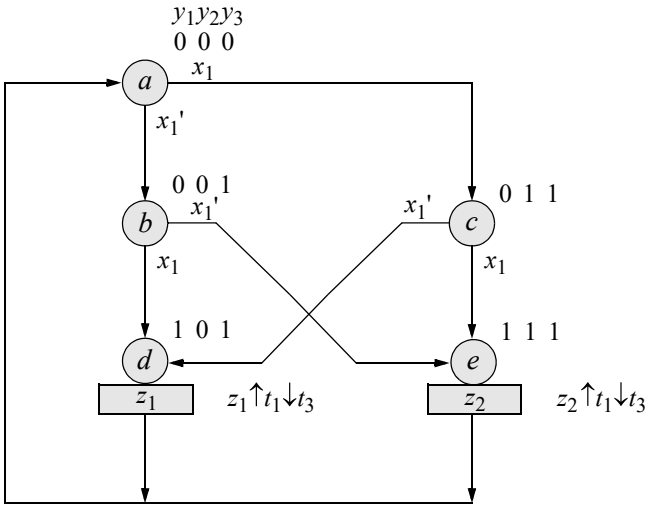
differ in only one variable. For example, state codes $y_1y_2y_3 = 101$ and 100 are adjacent because only y_3 changes. Thus, minterm locations 101 and 100 can be combined into one term. However, state codes $y_1y_2y_3 = 101$ and 110 are not adjacent because two variables change: flip-flops y_2 and y_3.

However, if the primary concern is to eliminate output glitches, then additional δ next-state logic is inconsequential in order to produce a more reliable machine. Output glitches can occur when two or more flip-flops change state and only when the machine specifications require that the outputs be asserted at time t_1.

States which have the same output should have adjacent state code assignments; that is, if states Y_i and Y_j both have z_1 as an output, then Y_i and Y_j should be adjacent. This allows for a larger grouping of 1s in the output map. If there is a contention between minimizing the δ next-state logic or the λ output logic — where it is possible to make either two nonoutput states adjacent or two output states adjacent, but not both — then a greater savings in logic is usually realized by minimizing the δ next-state input logic rather than the λ output logic.

An example of a Moore machine in which output glitches may occur is shown in the state diagram of Figure 7.75(a). In analyzing the state diagram, three paths are found which may generate glitches on outputs z_1 or z_2, depending on the propagation delays of flip-flops y_1, y_2, and y_3. Figure 7.75(b) resolves the problem by reassigning state codes.

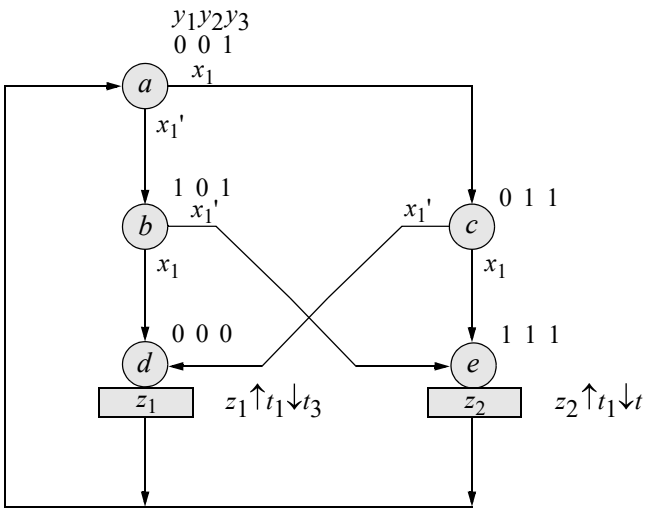
In Figure 7.75(a), the path from state b ($y_1y_2y_3 = 001$) to state e ($y_1y_2y_3 = 111$) may pass through state d ($y_1y_2y_3 = 101$) if flip-flop y_1 sets before flip-flop y_2 sets. This will assert output z_1 for a path that does not involve z_1.



(a)

(Continued on next page)

Figure 7.75 State diagram for a Moore machine: (a) may produce glitches on output z_1 and (b) glitch-free outputs due to state code reassignment.



(b)

Figure 7.75 (Continued)

A path that may result in a glitch on z_2 is the path from state c ($y_1y_2y_3 = 011$) to state d ($y_1y_2y_3 = 101$), which may pass through state e ($y_1y_2y_3 = 111$) if flip-flop y_1 sets before flip-flop y_2 resets. The path from state e ($y_1y_2y_3 = 111$) to state a ($y_1y_2y_3 = 000$) may pass through state d ($y_1y_2y_3 = 101$) and result in a glitch on output z_1 if flip-flop y_2 resets before flip-flops y_1 and y_3 reset.

Glitch elimination using storage elements It may not be possible to reassign state codes to eliminate output glitches. For example, Figure 7.76 presents a state diagram for a Moore machine in which any combination of state code assignments may still result in an output glitch. This can be verified by permuting the state codes so that all state transitions have been examined for all possible sets of codes.

In this machine, reassignment of state codes to eliminate output glitches is not possible, because all $2^p = 2^2 = 4$ state codes have been assigned and parallel paths exist for state transitions b, c, a and b, d, a. Thus, unused states cannot be utilized to avoid a state transition in which both flip-flops change state. A glitch may occur as the machine moves from state b ($y_1y_2 = 01$) to state d ($y_1y_2 = 10$) if flip-flop y_1 sets before flip-flop y_2 resets.

If the machine specifications require that $z_1\uparrow t_1\downarrow t_3$, then a D flip-flop in the λ output logic will satisfy this requirement, as shown in the logic diagram of Figure 7.77. A delayed clock signal is used to clock the D input of flip-flop z_1. The delay circuit delays the clock past the time (Δt) when a glitch could occur. Thus, although the output

of the AND gate that decodes z_1 may glitch, the active clock transition for flip-flop z_1 will not occur until after the glitch has returned to a logic 0. The clock delay is a very small delay, thus the assertion/deassertion is still considered to be $\uparrow t_1 \downarrow t_3$. Delay circuits usually require a separate driver and receiver.

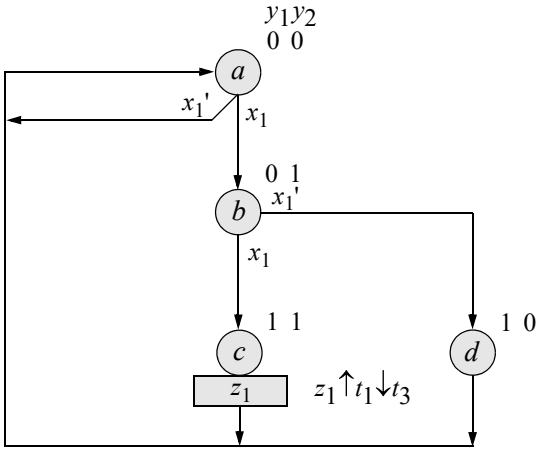


Figure 7.76 State diagram for a Moore machine with an inherent output glitch on z_1 for a state transition from state *b* to state *d*.

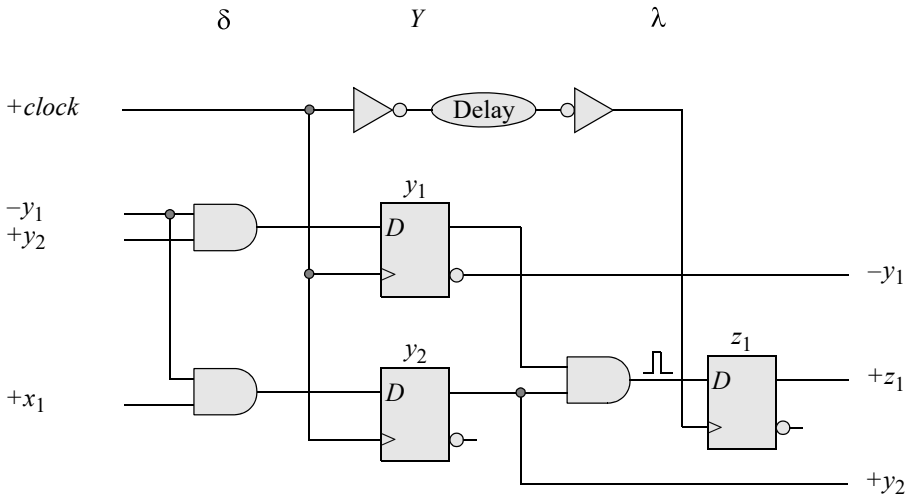


Figure 7.77 Logic diagram for the Moore machine of Figure 7.76 using a delayed clock to prevent a glitch on output z_1.

Glitch elimination using complemented clock The simplest and most inexpensive method of eliminating output glitches is to include the complement of the machine clock in the implementation of the λ output logic. The logic that generates the output will consist of an AND gate which decodes the p-tuple state codes. One input of the AND gate is connected to the complement of the machine clock; that is, the negation of the clock signal which drives the state flip-flops. This will generate an output signal that is only one-half the duration of the clock cycle, but guarantees that the output is free from any erroneous assertions. The output is asserted at time t_2 and deasserted at time t_3.

Figure 7.78 shows a general block diagram for a Moore machine using the complement of the machine clock as an output gating function. A glitch that is caused by a state transition in which two or more flip-flops change state has no effect on the output — the glitch has returned to a logic 0 before the active level of the complemented clock occurs. Since the output assertion is for only one-half a clock period, storage elements are not required for the λ output logic. This method also applies to Mealy machines; the inputs, however, must remain active during the last half of the clock cycle.

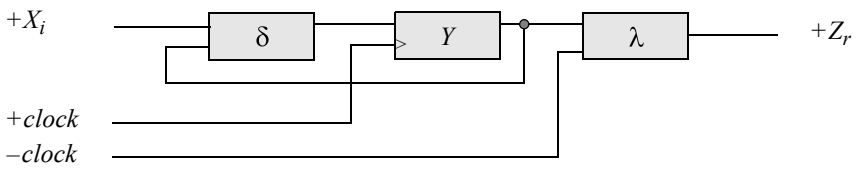


Figure 7.78 A generalized Moore machine which uses the complement of the machine clock as a gating function to eliminate output glitches.

Glitch elimination using delayed clock One final technique is presented to circumvent the negative effects of glitches. If the machine specifications require that outputs be asserted at time t_1 and deasserted at t_2, then glitches are again possible, because output assertion occurs at the active clock transition. This technique applies to both Moore and Mealy machines.

Figure 7.79 shows a general block diagram for a Mealy machine which uses the active level of the delayed machine clock to enable the λ output logic. The state flip-flops are clocked by the +*clock* signal, whereas the λ output logic is enabled by the +*clock* delayed signal. The duration of the delay circuit must be equal to or greater than the time Δt — the time when glitches can occur. The machine has stabilized at the termination of the Δt period. The delay circuit can be either a delay element with a dedicated driver and receiver or simply an even number of inverters.

Since the outputs are asserted for one-half a clock period, storage elements are not required. Storage elements are necessary only when the outputs are active for a duration of one clock period, either $\uparrow t_1 \downarrow t_3$ or $\uparrow t_2 \downarrow t_4$. Since both clock and complemented

clock are active for only one-half a clock cycle, they cannot be used to enable combinational λ output logic for one clock period.

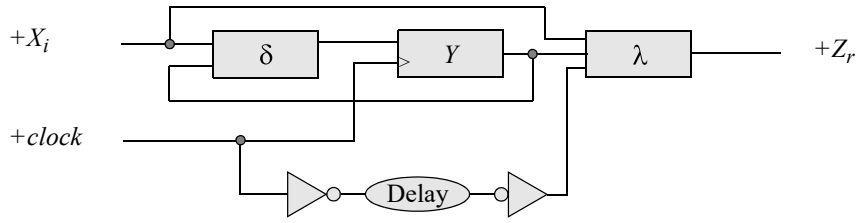


Figure 7.79 A general Mealy machine using a delayed clock as an output gating function, where $Z_r\uparrow t_1\downarrow t_2$.

Glitches and output maps Care must be taken when using "don't care" states in an output map in order to minimize the λ output logic. Any state transition that passes through an unused state — in which the beginning state and the destination state have the same output values — must have that same value placed in the unused state location. Since the unused state is a transient state, the output will glitch as the machine passes through that unused state en route to the destination state if a value is inserted that is different than the beginning and destination output value.

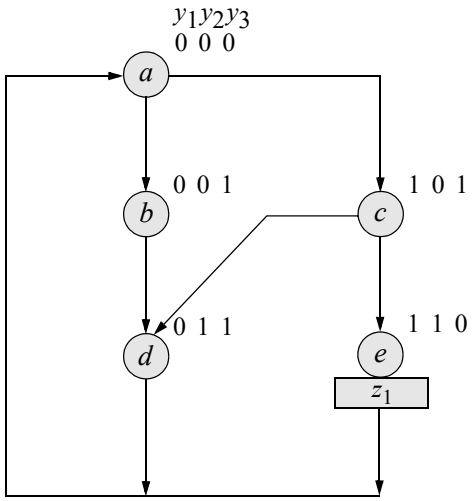
Consider the state diagram and output map for the Moore machine shown in Figure 7.80. The input variables have been omitted since they are not pertinent to the present discussion. When a reduced state diagram has been derived, the next step is to assign state codes. The state codes should be assigned so that no outputs will glitch as the machine progresses through the required state transition sequences. The state code assignment of Figure 7.80 results in an output that is free from transient assertions. This can be verified by checking all possible state transitions.

Referring to Figure 7.80(b), the equation for output z_1 can be minimized by combining the 1 in minterm 6 with one of the unused states in minterm 2, 4, or 7. Before using a "don't care" state, however, the state diagram must be checked to be certain that no state transition will occur that causes the machine to pass through the unused state en route to a destination state for a path that does not include z_1. Every path in the state diagram must be checked for this possible occurrence. If one path is found that may cause the machine to pass through an unused state that does not involve z_1, then a 0 must be inserted in the corresponding unused state.

If the "don't care" in minterm location 7 ($y_1y_2y_3 = 111$) combines with 1 entry in minterm location 6 ($y_1y_2y_3 = 110$) to yield $z_1 = y_1y_2$, then the path from state c ($y_1y_2y_3 = 101$) to state d ($y_1y_2y_3 = 011$) may pass through the unused state in minterm location 7 and generate a glitch on output z_1 if flip-flop y_2 sets before flip-flop y_1 resets.

In a similar manner, combining minterm locations 2 and 6 to yield $z_1 = y_2y_3'$ may produce a glitch on output z_1 for a transition from state d ($y_1y_2y_3 = 011$) to state a ($y_1y_2y_3 = 000$) if flip-flop y_3 resets before flip-flop y_2 resets. The problem of output glitches for this machine can be resolved by reassigning state codes as follows:

a: $(y_1y_2y_3 = 000)$
b: $(y_1y_2y_3 = 001)$
c: $(y_1y_2y_3 = 101)$
d: $(y_1y_2y_3 = 100)$
e: $(y_1y_2y_3 = 010)$



(a)

y_1 \ y_2y_3	0 0	0 1	1 1	10
0	0 (0)	0 (1)	0 (3)	– (2)
1	– (4)	0 (5)	– (7)	1 (6)

z_1

(b)

Figure 7.80 Moore machine: (a) state diagram and (b) output map.

7.3 Analysis of Asynchronous Sequential Machines

Another type of finite-state machine is an *asynchronous sequential machine*, where state changes occur on the application of input signals only — there is no machine clock. Like synchronous machines, the outputs of asynchronous machines are a function of either the present state only, or of the present state and the present inputs, corresponding to Moore and Mealy machines, respectively. The present state is directly related to the preceding sequence of inputs and states.

When an input variable changes in an asynchronous machine, the machine begins sequencing to a new state immediately. The input change may cause more than one storage element to alter its state, in which case the machine may sequence through various transient states before entering a stable state. This is caused by different propagation delays in the logic gates and storage elements.

The operational speed of a synchronous machine is regulated by, and is coincident with, a machine clock. Thus, the machine can change states only on the active transition of the clock signal. Because the speed of an asynchronous machine is not limited by a clock frequency, the operating characteristics are usually faster than those of a corresponding synchronous machine. The operational speed is limited only by the propagation delay of the longest path. The procedure for synthesizing a reliable asynchronous sequential machine is much more difficult and challenging than for a comparable synchronous sequential machine.

In the operation of synchronous sequential machines, the effect of transient output signals caused by varying circuit delays was negated by selecting an appropriate clock signal — either true or complemented — for the λ output logic. This is not possible, however, for asynchronous machines, because there is no system clock. The values placed in the output maps, therefore, must be carefully chosen.

Synchronous sequential machines occur more frequently in digital systems than their asynchronous counterparts, because of the ease with which the machines can be synthesized and the reliability imposed by a regularly occurring clock signal. In some cases, however, a machine clock may not be available, thus necessitating an asynchronous design. For extremely large synchronous machines using high-speed logic, the time required for the clock signal to propagate along the network of wires may be inordinately long. Thus, it is difficult to ensure the simultaneous arrival of the clock signal at all flip-flop clock inputs, which may hinder a reliable synchronous operation. In this situation also, an asynchronous design may be more appropriate.

Asynchronous machines are used extensively in the interface control logic of peripheral devices which attach to asynchronous interfaces between a channel (or I/O processor) and the device control unit. Since the interface contains no clock signal, the control unit utilizes randomly occurring interface control signals to transfer the data. During a write operation, the data bytes are transferred from the channel to the control unit interface logic which then synchronizes the data transfer rate to the speed of the peripheral device. During a read operation, the procedure is reversed.

Asynchronous sequential machines are implemented with set/reset (*SR*) latches as the storage elements. Thus, at least one *feedback* path is required in the synthesis of

asynchronous machines. Figure 7.81(a) illustrates an asynchronous machine which contains one feedback path from output z_1 to the input logic. Asynchronous machines are implemented in a sum-of-products form. Figure 7.81(b) shows the same circuit redrawn in the conventional latch configuration.

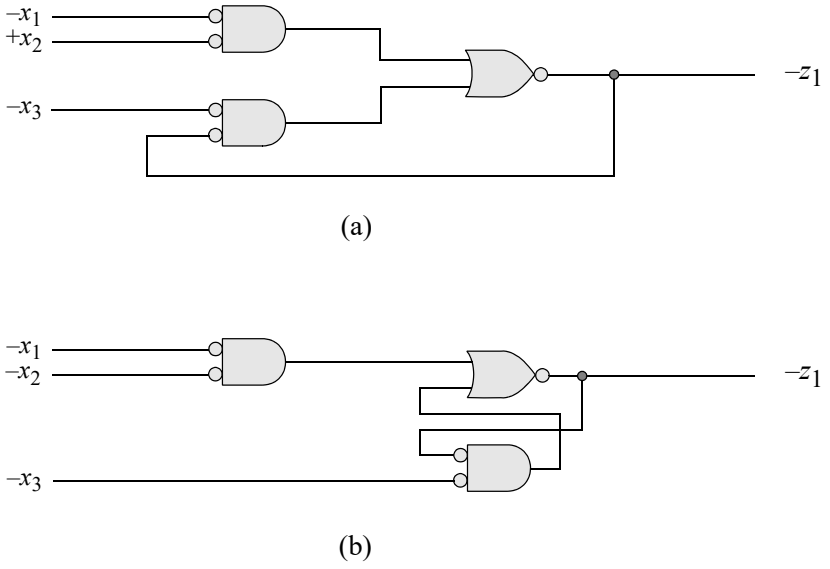


Figure 7.81 Asynchronous sequential machine with one feedback path: (a) sum-of-products and (b) conventional latch configuration.

7.3.1 Fundamental-Mode Model

Both synchronous and asynchronous sequential machines use feedback paths in their implementation. The feedback in synchronous machines combines with the input variables to form the δ next-state logic, which provides the next state for the machine at the next active clock transition. The feedback in asynchronous machines also combines with the δ next-state logic; however, this formation provides the necessary conditions to implement *SR* latches, and thus, generates a stable next state.

The primary approach to modeling and analyzing asynchronous sequential machines is to use a single delay element that represents the total delay of the entire machine and is placed in series with the feedback path. This simplifies the analysis procedure by specifying zero delay for all logic gates and interconnecting wires.

A general block diagram for an asynchronous sequential machine is shown in Figure 7.82. The input alphabet X consists of binary input variables $x_1, x_2, \cdots, x_n$ that can change value at any time and are represented as voltage levels rather than pulses. The state alphabet Y is characterized by p storage elements, where $Y_{1e}, Y_{2e}, \cdots, Y_{pe}$

are the *excitation* variables and $y_{1f}, y_{2f}, \cdots, y_{pf}$ are the *feedback* or *secondary* variables. The output alphabet Z is represented by $z_1, z_2, \cdots, z_m$.

Both the δ next-state logic and the λ output logic are composed of combinational logic circuits. The delay element in Figure 7.82 represents the total delay of the machine from the time an input changes until the machine has stabilized in the next state, and is represented as a time delay of Δt. The time correlation between the excitation variables Y_{ie} and the feedback variables y_{if} is specified by Equation 7.30.

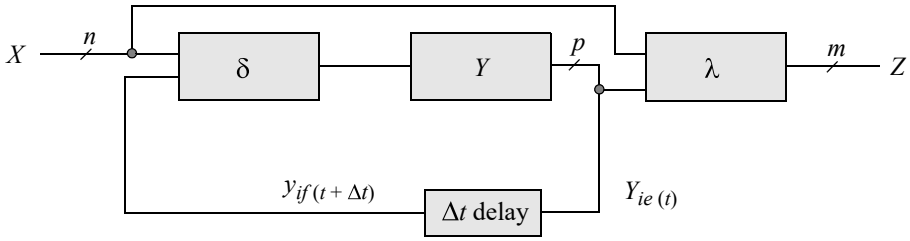


Figure 7.82 General block diagram of an asynchronous sequential machine.

$$y_{if}(t + \Delta t) = Y_{ie}(t) \tag{7.30}$$

For a given set of inputs, the machine will be in a stable state if and only if $y_{if} = Y_{ie}$, for $i = 1, 2, \ldots, p$; that is, after an appropriate delay, the feedback variables will be equal to the excitation variables, indicating that the machine has entered a *stable state*. When an input changes value, the δ next-state combinational logic produces a new set of values for the excitation variables $Y_{1e}, Y_{2e}, \cdots, Y_{pe}$. The machine then sequences through one or more unstable states. When the values of the feedback variables $y_{1f}, y_{2f}, \cdots, y_{pf}$ are equal to the values of the excitation variables $Y_{1e}, Y_{2e}, \cdots, Y_{pe}$, then the machine is stable in that state.

The transition from one stable state to another stable state is the result of a single change to an input variable. To ensure a *deterministic operation*, it is assumed that only one input changes at a time. If two inputs change simultaneously, then this will result in a *race* condition in which the machine may sequence to an incorrect next state. Race conditions are discussed in a later section.

Postulating that only one input changes value at a time should present no undo rigor in the synthesis procedure. It is always possible to affect a modification to the external circuitry so that only a single input changes value. This can be accomplished without altering the machine specifications. Also, when a binary input variable changes, the new value must be of sufficient duration to cause the machine to

change state and proceed to a stable state. It is also assumed that no other input will change until the machine has entered a stable state. The previous two conditions specify a *fundamental-mode* operation. Thus, a fundamental-mode model has the following characteristics:

1. Only one input changes at a time.

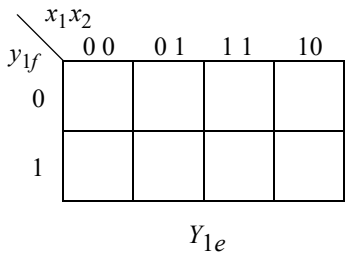
2. No other input will change until the machine has sequenced to a stable state.

Whenever a single input changes, a new input vector is generated, which in turn produces a new set of excitation variables. At that instant in time, however, the feedback variables are not yet equal to the excitation variables because the Δt delay has not yet occurred. Thus, $y_{if} \neq Y_{ie}$ and the machine enters an unstable state. After a delay of Δt, all circuit changes have transpired and the machine enters a stable state where $y_{if} = Y_{ie}$. The machine remains in this stable state until another input change causes a new set of excitation variables to be generated. The machine will exit a stable state only when an input variable changes value.

7.3.2 Methods of Analysis

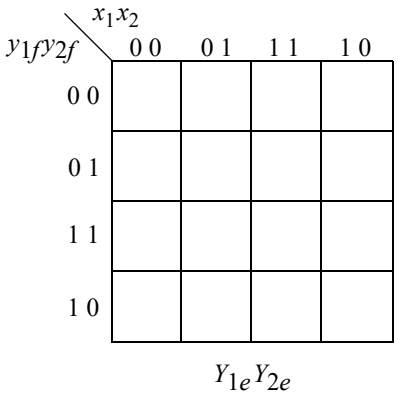
This section presents mechanisms for analyzing asynchronous sequential machines. These analysis techniques are similar in concept to those used for analyzing synchronous sequential machines. The behavior of an asynchronous machine can be completely specified by an excitation map, a next-state table, a state diagram, and a flow table. Also included in the analysis are the excitation and output equations.

In general, the excitation variables are a Boolean function of the inputs and the feedback variables. A Karnaugh map is a convenient method of representing the excitation variables. An excitation map is a Karnaugh map in which the columns are specified by the input variables and the rows by the feedback variables. The entries in the minterm locations represent the values of the excitation variables. The formats for representative excitation maps are shown in Figure 7.83 for one and two feedback variables.



(a)

Figure 7.83 Formats for representative excitation maps: (a) two inputs and one excitation variable and (b) two inputs and two excitation variables.



(b)

Figure 7.83 (Continued)

Example 7.10 Figure 7.84 illustrates a simple asynchronous sequential machine which will be used as an introductory example for analysis. The output of the circuit is the excitation variable Y_{1e}, which also corresponds to the machine output z_1. After a delay of Δt, the feedback variable y_{1f} becomes equal to the excitation variable Y_{1e} and the machine is stable.

The equation for Y_{1e} is obtained directly from the logic diagram as a function of the input variables x_1 and x_2 and of the feedback variable y_{1f}, as shown in Equation 7.31. The excitation variable is usually portrayed in the sum-of-products format. In this simple asynchronous machine, output z_1 has the same characteristics as the excitation variable Y_{1e}. Thus, the machine operates as a Moore model, because the output is a function of the state alphabet only and is independent of the input alphabet.

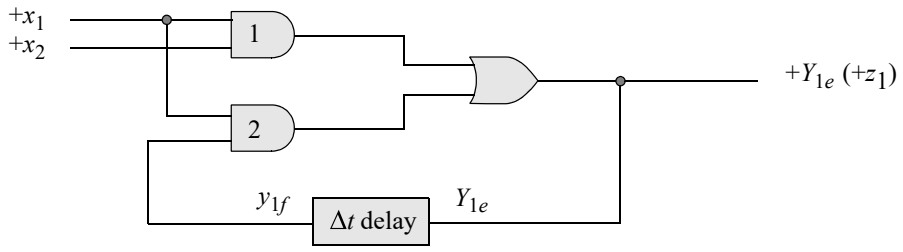


Figure 7.84 Asynchronous sequential machine with one feedback path.

The excitation map is obtained by plotting the equation for Y_{1e} in a Karnaugh map, as shown in Figure 7.85(a). The input variables x_1 and x_2 specify the columns of

the map and are enumerated in the Gray code format in the same manner as for all Karnaugh maps of two or more variables. The feedback variable y_{1f} defines the values of the rows.

$$Y_{1e} = x_1x_2 + x_1y_{1f}$$
$$z_1 = Y_{1e} \tag{7.31}$$

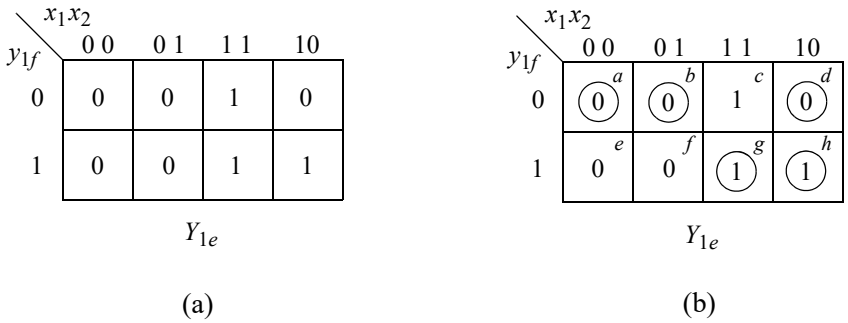


Figure 7.85 Karnaugh maps for the Moore machine of Figure 7.84: (a) excitation map and (b) excitation map showing stable states, which are indicated by circled entries.

In order to characterize the operation of the machine, the stable states must be identified. Recall that an asynchronous sequential machine is stable in a particular state only if the feedback variable is equal to the excitation variable; that is, $y_{1f} = Y_{1e}$. By convention, a *stable state* is indicated by circling the corresponding map entry, as shown in Figure 7.85(b). Since the values inserted in the minterm locations correspond to the values of Y_{1e}, any map entry that is equal to the row value of y_{1f} will indicate a stable state.

To facilitate the analysis of the machine, state names are inserted in the minterm locations together with the value of the excitation variable. When specifying a stable state, a circle is placed around the state variable or state name. Thus, in the row corresponding to $y_{1f} = 0$, states ⓐ, ⓑ, and ⓓ are stable, because in all three states $y_{1f} = Y_{1e} = 0$. Similarly, in the row corresponding to $y_{1f} = 1$, states ⓖ and ⓗ are stable, because in both states $y_{1f} = Y_{1e} = 1$.

The state transition sequences for the machine can now be determined. An asynchronous machine will remain in a stable state until an input changes. A change of a single input causes a horizontal movement in the excitation map. Assume that the machine is presently in stable state ⓐ ($y_{1f}x_1x_2 = 000$) and that input x_2 changes from 0 to 1. The machine will proceed horizontally to state ⓑ ($y_{1f}x_1x_2 = 001$) which is also stable, because $y_{1f} = Y_{1e} = 0$. The machine will remain in state ⓑ until an input again changes.

Similarly, if the machine is in state ⓐ and x_1 changes from 0 to 1, then the machine sequences horizontally in row $y_{1f} = 0$ and enters state ⓓ. Since the excitation value does not change, the feedback value will not change. Thus, the entire operation takes place only in row $y_{1f} = 0$. Note that it is not possible to sequence from state ⓓ ($y_{1f}x_1x_2 = 010$) to state ⓗ ($y_{1f}x_1x_2 = 110$) directly, because inputs x_1 and x_2 remain unchanged. In order to leave a stable state, an input must change value. State ⓗ can be entered only from state ⓖ. A state transition will occur from state ⓖ to state ⓗ if x_2 changes from 1 to 0. The machine will remain in state ⓗ until another change of input occurs, either to x_1 or x_2.

It is also not possible for the machine to sequence from state ⓑ to state ⓓ directly, because this transition requires a change of two input variables, a situation that is not allowed in asynchronous sequential machine operation. That is, $x_1x_2 = 01$ cannot change to $x_1x_2 = 10$ directly, or conversely. For the same reason, a simultaneous change from $x_1x_2 = 00$ to $x_1x_2 = 11$, or conversely, is not allowed.

When a single input changes, the machine can sequence to only those states that are contained in the column specified by the new input vector. Therefore, in state ⓑ, when input x_1 changes from 0 to 1, the machine moves horizontally to unstable state *c*, then vertically in column $x_1x_2 = 11$, and settles in state ⓖ. The state transition sequence for this input change is ⓑ $\rightarrow c \rightarrow$ ⓖ. In unstable states, the inputs do not change — only the feedback variables change. To cause a feedback variable to change, the machine must pass through an unstable state.

If the next state is unstable, then the machine is in a transient state in the same feedback row and in the column specified by the new input values. There will be no vertical movement until the feedback variable begins to change in order to assume the value of the excitation variable. After a delay of Δt, the machine will move vertically in the same column and enter a row in which the value of the feedback variable is equal to the value of the previous excitation variable.

An alternative method of illustrating the flow of the machine is to tabulate the stable and unstable states as shown in the flow table of Figure 7.86. The flow table can be generated directly from the excitation map of Figure 7.85. The state names are chosen arbitrarily and only the stable states are assigned unique names. The flow table directly corresponds to the excitation map of Figure 7.85(b). The excitation variables, however, are replaced with state names, allowing for easier interpretation of machine operation.

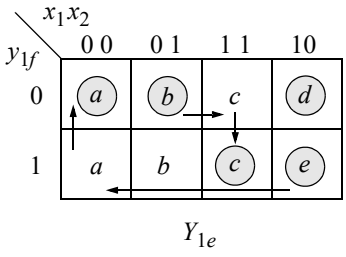


Figure 7.86 Flow table for the Moore machine of Figure 7.84.

The flow table provides a convenient notational technique for analyzing machine operation. The table specifies the state transition sequences resulting from an input change. Thus, the behavior of the machine is completely specified by the flow table. The flow table is an extremely useful instrument for analysis and will be used extensively in the synthesis of asynchronous sequential machines.

The circled states in Figure 7.86 represent stable states; the uncircled states represent unstable states and indicate the name of the destination stable state to which the machine will sequence after a delay of Δt. Thus, if the machine is presently in state ⓑ and input x_1 changes from 0 to 1, then the operation proceeds horizontally in row $y_{1f} = 0$ and the machine will sequence through transient state c and proceed to stable state ⓒ after a delay of Δt.

Similarly, the transition from state ⓔ to state ⓐ is the result of input x_1 changing from 1 to 0, which causes the operation to move horizontally in row $y_{1f} = 1$ to unstable state a. After a delay of Δt, the machine will pass through transient state a and enter state ⓐ. All state transition sequences can be observed in the flow table, in which any transition is the result of a single input change.

Example 7.11 The logic diagram for a Moore asynchronous sequential machine is shown in Figure 7.87. The machine will be analyzed by obtaining the excitation map and the flow table. The excitation map is shown in Figure 7.88 and is constructed by applying all combinations of the inputs to the machine in conjunction with the feedback variable.

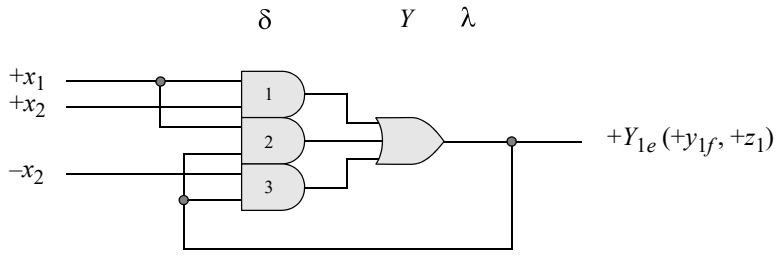


Figure 7.87 Logic diagram for the Moore machine of Example 7.11.

y_{1f} \ x_1x_2	0 0	0 1	1 1	1 0
0	(0)	(0)	1	(0)
1	(1)	0	(1)	(1)

Y_{1e}

Figure 7.88 Excitation map for the Moore machine of Example 7.11.

For example, if the feedback variable $y_{1f} = 0$, then the only condition that will cause the excitation variable Y_{1e} to be equal to 1 is when $x_1x_2 = 11$; otherwise, $Y_{1e} = 0$. In a similar manner, if $y_{1f} = 1$, then the only condition that will cause the excitation variable Y_{1e} to be equal to 0 is when $x_1x_2 = 01$; in all other cases, $Y_{1e} = 1$.

The flow table is shown in Figure 7.89 in which the state names for the stable states are arbitrarily chosen and correspond to the stable state locations in the excitation map. Assume that the machine is stable in state ⓑ where $y_{1f} = 0$ and input x_1 changes from 0 to 1. In Figure 7.87, gate 1 now has both inputs at a logic 1, which will cause Y_{1e} to be equal to a logic 1 indicated by unstable state *e* in Figure 7.89. After a delay of Δt, the feedback variable $y_{1f} = 1$ and the machine enters stable state ⓔ.

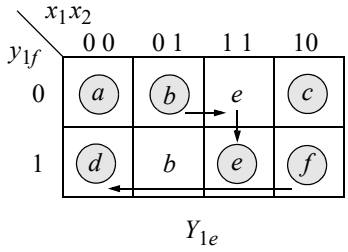


Figure 7.89 Flow table for the Moore machine of Example 7.11.

Consider the case where the machine is stable in state ⓒ with $x_1x_2 = 10$. If x_2 changes from 0 to 1, then gate 1 now has both inputs at a logic 1, which will cause Y_{1e} to be equal to a logic 1. As before, the machine passes through unstable state *e* and enters stable state ⓔ.

Now assume that the machine is in stable state ⓕ where $x_1x_2 = 10$ and $Y_{1e} = 1$. The feedback paths through gate 2 and gate 3 maintain $Y_{1e} = 1$. If input x_1 changes from 1 to 0, then the outputs of gate 1 and gate 2 become a logic 0; however, since $x_2 = 0$, the feedback path through gate 3 maintains the excitation variable $Y_{1e} = 1$ and the machine sequences to stable state ⓓ.

7.3.3 Hazards

The signal propagation delay must be considered in the analysis and synthesis of asynchronous sequential machines. When an input variable changes value, varying propagation delays caused by logic gates, wires, and different path lengths can produce erroneous transient signals on the outputs. These spurious signals are referred to as *hazards*. If the hazard occurs in the feedback path, then an incorrect state transition sequence may result.

Hazards are not usually a concern in synchronous sequential machines because the clock negates the effects of a hazard. A hazard may appear at the input of a flip-flop, but is deasserted before the machine changes state at the next active clock

transition. In asynchronous sequential machines, however, where the next state is not synchronized with a clock signal, a hazard can cause an incorrect next state or an erroneous output, and must be eliminated.

These transitory signals generate a condition which is not specified in the expression for the machine, because Boolean algebra does not take into account the propagation delay of switching circuits. In this section, hazards will be examined and methods presented for detecting and correcting these transient phenomena so that correct operation of an asynchronous sequential machine can be assured.

Static hazards Figure 7.90 illustrates a classic example of a combinational circuit with an inherent hazard. Although the network is combinational in structure, it may be an integral part of the δ next-state logic for an asynchronous sequential machine. The Karnaugh map which represents the circuit is shown in Figure 7.91 and the equation for output z_1 is shown in Equation 7.32. The map entries indicate that z_1 is asserted in minterm locations 1, 5, 6, and 7; thus, the function $z_1(x_1, x_2, x_3) = \Sigma_m(1, 5, 6, 7)$. In particular, z_1 is active if $x_1x_2x_3$ = 111 or 101; that is, output z_1 is set to a value of 1 regardless of the value of input x_2.

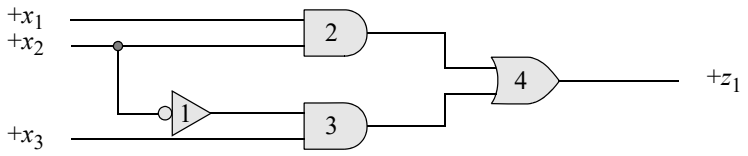


Figure 7.90 Combinational logic circuit that contains a potential static hazard.

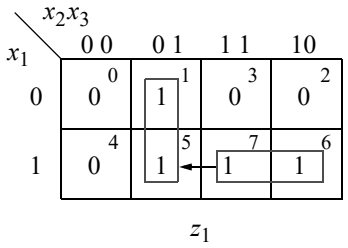


Figure 7.91 Karnaugh map for the logic diagram of Figure 7.90.

$$z_1 = x_1x_2 + x_2'x_3 \tag{7.32}$$

Thus, if x_2 changes from 0 to 1 or from 1 to 0, the state of z_1 should remain constant (or static) during the transition. Any deviation from this static condition is referred to as a "static hazard." If an input variable changes value causing an output to be deasserted momentarily when the output should remain asserted, then this is classified as a *static-1 hazard.* Conversely, when an input change causes an output to become asserted momentarily when it should remain deasserted, a *static-0 hazard* results. Figure 7.92 illustrates the two types of static hazards. In general, a static hazard is a singularity in which a single change to an input vector causes a momentary output transition to an incorrect state.

(a) (b)

Figure 7.92 Static hazards: (a) static-1 hazard and (b) static-0 hazard.

If two adjacent minterm locations both contain a value of 1 and are covered by the same prime implicant, then a single change within that grouping of 1s will not produce a hazard. For example, in the Karnaugh map of Figure 7.91, changing the input vector from $x_1x_2x_3 = 111$ to 110 will not cause a static hazard because the corresponding 1s are covered by the term x_1x_2. Thus, when x_3 changes from 1 to 0, the x_1x_2 term maintains the output at a logic 1. Therefore, all groups of 1s in a Karnaugh map must be connected by redundant prime implicants (or consensus terms) to avoid possible hazards.

Referring to the logic diagram of Figure 7.90, when input x_2 changes from 1 to 0, the change is transmitted to the output along two paths: logic blocks 2 and 4; and logic blocks 1, 3, and 4. If the delay of block 2 is less than the combined delay of blocks 1 and 3, then all inputs to block 4 will be at a low level for a short duration. If the block 4 inputs remain at a low level for a sufficient duration, then output z_1 will generate a static-1 hazard.

Figure 7.93 illustrates an approach to analyzing the circuit of Figure 7.90 for hazards. For input x_2, the top row of +/– symbols refers to the logic levels produced as a result of input x_2 changing from a high to a low level. The bottom row of +/– symbols specify the logic levels which result when x_2 changes from a low to a high level.

Refer to the top row of +/– symbols for the discussion which follows. Assume that x_2 changes from a high to a low level. The first column in each of the seven sets of columns specifies a stable state, where $x_1x_2x_3 = 111$. The second column represents the state of the circuit after one gate delay, in which the inverter and gate 2 are in the same time relationship. The third column indicates the state of the circuit after two gate delays, in which gate 3 and gate 4 (from the top input) produce concurrent delays. At this

time, output z_1 changes to a low voltage level. Finally, after three gate delays — the inverter, gate 3, and gate 4 (from the bottom input) — the circuit resumes a stable state, where z_1 returns to a high level.

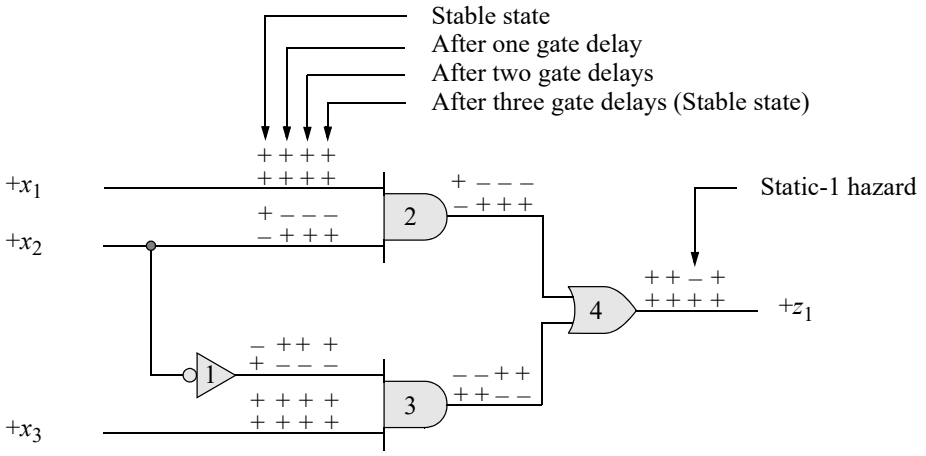


Figure 7.93 A method for analyzing a logic circuit for static hazards.

Now consider the case where input x_2 changes from a low to a high logic level. If the initial state of the circuit is $x_1x_2x_3 = 101$ and x_2 changes from a low to a high level, then no hazard will occur on output z_1, as indicated by the bottom row of +/– symbols. There will never be a time period when both inputs of the OR gate are at a low logic level. Thus, the circuit will not generate a static-1 hazard on z_1.

Because the map of Figure 7.91 specifies that output z_1 should remain at a constant high level when $x_1x_3 = 11$, the effects of the hazard can be eliminated by adding a third term to the equation for z_1. The output can be made independent of the value of x_2 by including the redundant prime implicant x_1x_3, which covers both the initial and terminal state of the transition. The redundant prime implicant will maintain the output at a constant high level during the transition.

The term x_1x_3 is called a *hazard cover*, since it covers the detrimental effects of the hazard. The effects of a static hazard can be negated by combining adjacent groups of 1s in a Karnaugh map as shown in Figure 7.94 for the Karnaugh map of Figure 7.91. The new equation for z_1 is shown in Equation 7.33. The hazard will still occur at the OR gate inputs, but its effect will be negated due to the addition of gate 5, as shown in Figure 7.95. Although the circuit no longer contains a minimal number of gates, the operation is reliable.

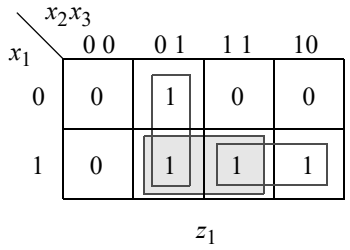


Figure 7.94 Negating the effects of a static hazard by combining adjacent groups of 1s with a redundant prime implicant term.

$$z_1 = x_2'x_3 + x_1x_2 + x_1x_3 \qquad (7.33)$$

Hazard cover

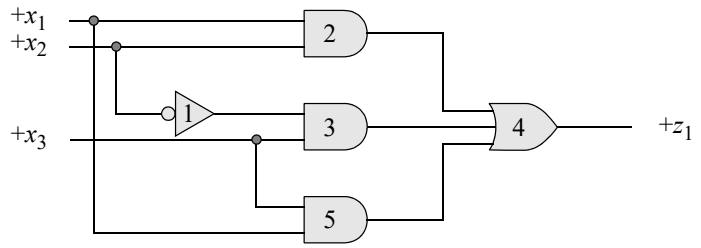


Figure 7.95 Eliminating the effects of a static-1 hazard for Figure 7.90 by adding the redundant prime implicant term x_1x_3 to the equation for z_1 represented by gate 5.

Static-0 hazards can be analyzed and eliminated by using the same technique, except that adjacent groups of 0s are combined. Consider the Karnaugh map of Figure 7.96 in which 0s are grouped to form the product-of-sums expression shown in Equation 7.34. In minterm location $x_1x_2x_3 = 100$, if x_2 changes from 0 to 1, then the machine will generate a static-0 hazard as it sequences to minterm location $x_1x_2x_3 = 110$, as shown in Figure 7.97. However, the map indicates that z_1 should not change value during this transition; that is, z_1 should remain at a value of 0.

As in the sum-of-products implementation, the hazard can be eliminated by adding a redundant term. The additional sum term $(x_1' + x_3)$ is appended to the product-of-sums expression as shown in Equation 7.35. Applying the product terms $x_1'x_2'$ or x_2x_3 to Equation 7.35 will yield a value of 1 for z_1 as specified by the Karnaugh map of Figure 7.96 by combining groups of 1s.

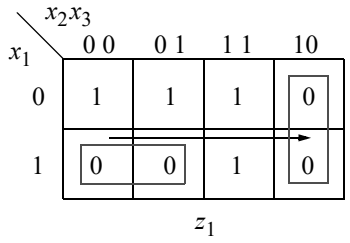


Figure 7.96 Karnaugh map for a product-of-sums with a potential static-0 hazard.

$$z_1 = (x_1' + x_2)(x_2' + x_3) \tag{7.34}$$

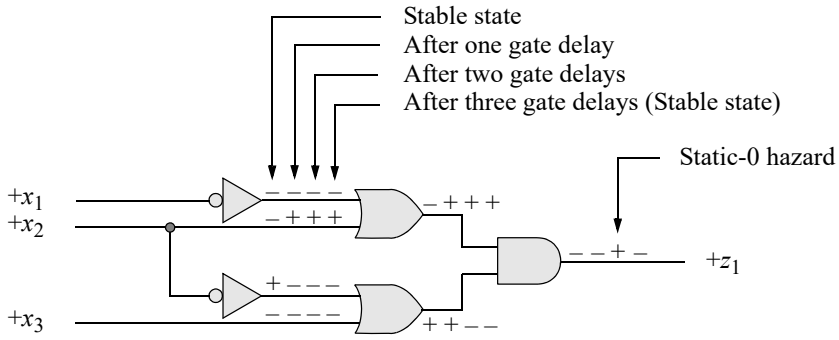


Figure 7.97 A static-0 hazard exhibited on output z_1 for the product-of-sums implementation of Figure 7.96.

$$z_1 = (x_1' + x_2)(x_2' + x_3)(x_1' + x_3) \tag{7.35}$$

Hazard cover

In order for there to be a potential static hazard, the groups of 1s or 0s must be adjacent. For example, the following equation will be used to generate a Karnaugh map which will be analyzed for possible static-1 and static-0 hazards:

$$z_1 = x_1'x_2x_3 + x_1x_3' + x_1'x_2'x_3$$

The Karnaugh map is shown in Figure 7.98, which clearly shows that the groups of 1s and 0s are not adjacent to other groups of 1s or 0s.

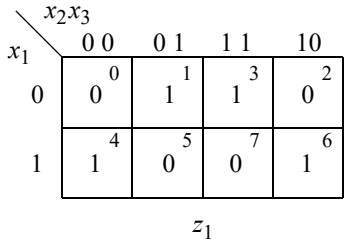


Figure 7.98 Karnaugh map which has no static-1 or static-0 hazards.

Dynamic hazards Dynamic hazards, like static hazards, may also cause erroneous outputs in combinational circuits. If the combinational network is incorporated in the δ next-state logic of an asynchronous sequential machine, then an incorrect next state may result. The same rationale applies to dynamic hazards in the λ output combinational logic. A *dynamic hazard* is characterized by multiple output pulses resulting from a single change to the input vector.

When a single input variable changes value, an odd number of transitions may occur on the output signal, where the number of transitions is greater than one. Thus, the input change propagates toward the output signal along 3, 5, 7, ... paths. In order for a dynamic hazard to be realized, at least three different path lengths must be encountered. The first path will constitute a minimum propagation delay and cause the output to change value; the second path will cause an intermediate delay in which the output will return to its previous value; and the third path results in a maximum delay which causes the output to make a third transition. The last two transitions also generate a static hazard. Figure 7.99 illustrates two typical types of dynamic hazards.

Figure 7.99 Dynamic hazards with three transitions.

Dynamic hazards can be eliminated in a manner analogous to that used for static hazards; that is, by adding redundant terms to the output equation. It may also be possible to change the form of the equation to eliminate a dynamic hazard. Figure 7.100 illustrates a nonminimized combinational circuit with an inherent dynamic hazard. The equation for output z_1 is shown in Equation 7.36. Let x_1, x_3, and x_4 be active low inputs and x_2 be active high. With all inputs active at their respective levels, the circuit is stable and z_1 is at a high logic level.

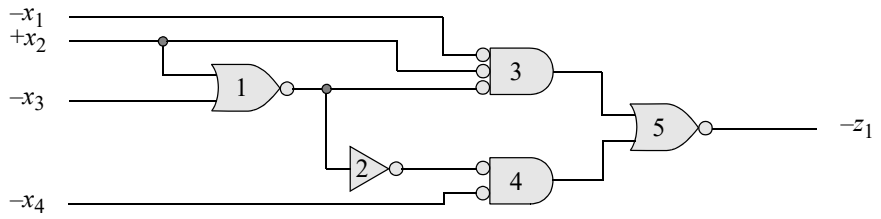


Figure 7.100 Combinational circuit with an inherent dynamic hazard.

$$z_1 = x_1 x_2'(x_2 + x_3') + x_2' x_3 x_4 \tag{7.36}$$

The timing diagram of Figure 7.101 depicts the sequence of events that occur when input x_2 changes from a high to a low logic level. The waveforms labeled OR1, INV2, AND3, AND4, and OR5 refer to the outputs of their respective logic blocks. The deassertion of x_2 is immediate. The new value of x_2 propagates to the output terminal along three paths.

The first path involves two gate delays: gates 3 and 5. After one gate delay, the output of gate 3 changes from a low to a high level. This change is reflected on output z_1 after a second gate delay through gate 5, at which time z_1 changes from a high to a low level.

The second path involves three gate delays: gates 1, 3, and 5. After one gate delay, the output of gate 1 changes from a low to a high level. This change is reflected on the output of gate 3 after a second delay (through gate 3) which now changes from a high to a low level. Both inputs to gate 5 are now at a low logic level. Therefore, after a third delay (through gate 5), output z_1 changes from a low to a high level.

The third path entails four gate delays: gate 1, the inverter, gate 4, and gate 5. After one gate delay, the output of gate 1 changes from a low to a high level. This change is manifested on the output of the inverter, which changes from a high to a low level. Both inputs to gate 4 are now at a low logic level. Therefore, after a third delay (through gate 4), the output of gate 4 changes from a low to a high level. This change encounters the fourth delay for this path as the signal propagates through gate 5 causing output z_1 to change from a high to a low logic level. The output eventually changes

state, as it should, but only after the occurrence of two extra transitions. The resulting effect on z_1 from this single input change is a dynamic hazard with a triple change of state.

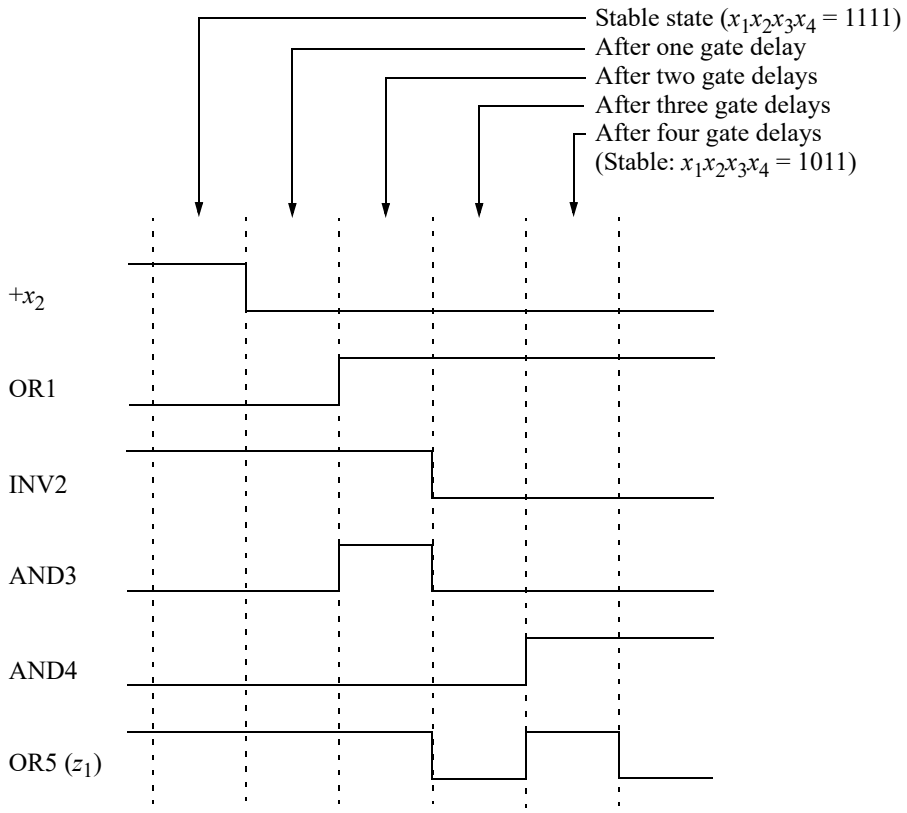


Figure 7.101 Timing diagram for the circuit of Figure 7.100 illustrating a dynamic hazard on output z_1.

Dynamic hazards, which are less common than static hazards, result from a single change to the input vector in which the change propagates to the output terminal along three or more different path lengths. Reconfiguring the logic equation, without changing the functionality of the circuit, will eliminate dynamic hazards. Dynamic hazards can also be removed as a direct result of eliminating static hazards, in which redundant terms are added to the output equation.

Essential hazards An asynchronous sequential machine may be free of static and dynamic hazards in the δ next-state logic and the λ output logic, but may still sequence to an incorrect next state due to inordinately long propagation delays in certain circuit

elements. The propagation delay through a logic gate may increase due to temperature change, power supply output variation, or component aging.

An *essential hazard* is caused not by an incorrectly synthesized network, but by the operation of the circuit itself; that is, it is inherent in the machine's design. Thus, the essential hazard cannot be eliminated by adding redundant terms in the network equation or by changing the form of the equation. Since an essential hazard is the result of excessive propagation delay in certain network elements, the effect of the hazard can be nullified by introducing appropriate delays in other components, such that the cause of the hazard will be negated.

Essential hazards occur specifically in fundamental-mode machines and are characterized by two propagation paths: one path affecting storage element y_j, the other path affecting storage element y_k. Essential hazards can be detected by analyzing the excitation map or flow table of an asynchronous sequential machine.

The machine has a possible essential hazard if, beginning in a stable state, input x_i changes value and sequences the machine to a stable state that is different than the stable state reached after three successive changes to x_i. That is, an asynchronous sequential machine contains a possible essential hazard if a single change to an input variable x_i results in a transition from state S_j to state S_k, whereas three consecutive changes to x_i results in a state transition sequence which terminates in state S_l, where $S_k \neq S_l$.

Thus, when the machine is implemented, there may be a series of propagation delays that will cause the machine to sequence to an incorrect next stable state resulting from a single change to input x_i. This incorrect state change occurs because the change to x_i propagates to different circuit elements at different times. Consider the asynchronous sequential machine specified by the excitation maps and flow table of Figure 7.102 and Equation 7.37.

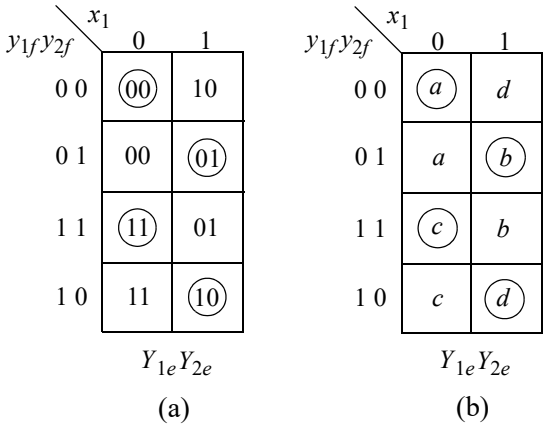


Figure 7.102 Karnaugh maps for an asynchronous sequential machine containing a possible essential hazard: (a) combined excitation map and (b) flow table.

$$Y_{1e} = y_{2f}'x_1 + y_{1f}x_1'$$
$$Y_{2e} = y_{1f}x_1' + y_{2f}x_1 \tag{7.37}$$

The logic diagram is shown in Figure 7.103. If the machine is stable in state ⓐ ($y_{1f}y_{2f}x_1 = 000$) and input x_1 changes from a low to a high logic level, then an essential hazard is possible if the delay through the inverter is excessive; that is,

Inverter delay > delay of gate 1 + delay of gate 2 + delay of gate 4

The machine should sequence from state ⓐ ($y_{1f}y_{2f}x_1 = 000$) to state ⓓ ($y_{1f}y_{2f}x_1 = 101$) if the gate delays are within their respective specified range. However, the sequence described below is possible if the delay through the inverter greatly exceeds its maximum propagation delay.

Refer to the logic diagram of Figure 7.103, which is designed from the equations of Equation 7.37. If the machine is stable in state ⓐ ($y_{1f}y_{2f}x_1 = 000$) and input x_1 changes from a low to a high level, then the output of gate 1 changes to a low level after an appropriate delay. Latch y_1 will then set after a delay through gate 2. The circuit is now in a transient state of $y_{1f}y_{2f}x_1 = 101$.

The $+y_{1f}$ output of gate 2 connects to the input of gate 4. The inverter would normally have disabled gate 4 by this time and the machine would have stabilized in state $y_{1f}y_{2f}x_1 = 101$. The output of the inverter, however, is still at a high logic level due to excessive propagation delay. Therefore, the output of gate 4 changes to a low level and sets latch y_2 after a delay through gate 5. During this time period, the inverter propagates the change incurred by input x_1.

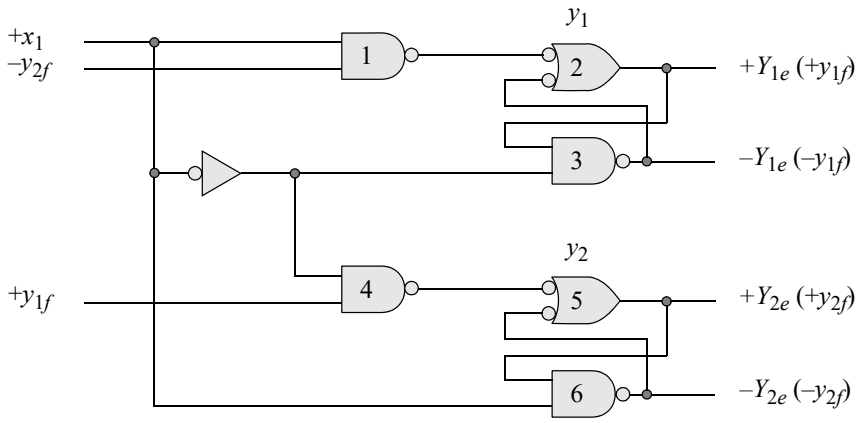


Figure 7.103 An asynchronous sequential machine with a possible essential hazard.

Even though the inverter has disabled the output of gate 4, latch y_2 remains set as a result of the feedback path ($+y_{2f}$) and the connection between the output of gate 6 and the input to gate 5. The inverter output also applies a low level to reset latch y_1. Since the output of gate 1 is disabled because of the $-y_{2f}$ signal from latch y_2, therefore latch y_1 is reset. After all propagation delays have elapsed, the machine is stable in state ⓑ ($y_{1f}y_{2f}x_1 = 011$), which is an incorrect terminal state for the state transition sequence ⓐ → ⓓ.

The hazard thus realized is essential; that is, the design of the machine cannot be changed to eliminate the hazard and still conform to the machine specifications. The essential hazard occurs because the change to input x_1 is transmitted to different parts of the circuit at different times. The change of state for y_1 is received at gate 4 before the inverter has propagated the change from x_1. The inverter output would normally have applied a low level to the input of gate 4, effectively preventing latch y_2 from being set.

Essential hazards can be eliminated in an asynchronous sequential machine — without affecting the logical operation of the machine — by inserting appropriate delay circuits in strategic locations within the machine. For example, if sufficient delay was added to the output of latch y_1, then the change to input x_1 would propagate to all the appropriate gates before the change to y_1 was received at those gates. Thus, the essential hazard is eliminated and proper operation of the machine is assured.

7.3.4 Oscillations

An *oscillation* occurs in an asynchronous sequential machine when a single input change results in an input vector in which there is no stable state. Consider the excitation map of Figure 7.104 for Y_{1e}. There are two input variables x_1 and x_2 and one feedback variable y_{1f}. If the machine is in stable state ⓑ ($y_{1f}x_1x_2 = 001$) where $y_{1f} = Y_{1e} = 0$ and x_1 changes from 0 to 1, then the machine sequences to transient state c ($y_{1f}x_1x_2 = 011$). In state c, the excitation variable $Y_{1e} = 1$ and the feedback variable $y_{1f} = 0$.

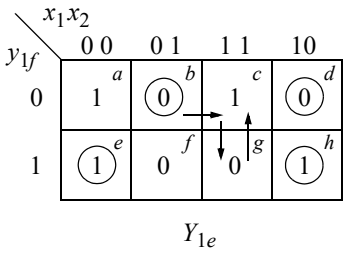


Figure 7.104 Excitation map for an asynchronous sequential machine containing an oscillation.

Thus, after a delay of Δt, the feedback variable becomes equal to the excitation variable and the machine proceeds to state g, where the feedback variable $y_{1f} = 1$. In state g, however, the excitation variable $Y_{1e} = 0$, designating state g as an unstable (or transient) state, because $y_{1f} \neq Y_{1e}$. After a further delay of Δt, the feedback variable becomes equal to the excitation variable and the machine sequences to state c, where the feedback variable $y_{1f} = 0$. Since the input vector $x_1x_2 = 11$ provides no stable state, the machine will oscillate between transient states c and g.

Another example depicting oscillations is shown in the excitation map for excitation variables Y_{1e} and Y_{2e} of Figure 7.105. All state transition sequences must be considered when analyzing the machine for oscillations. In this asynchronous sequential machine, there are multiple oscillations.

$y_{1f}y_{2f}$ \ x_1x_2	0 0	0 1	1 1	1 0
0 0	*a* 01	*b* 01	*c* 01	*d* 01
0 1	*e* (01)	*f* (01)	*g* 00	*h* 11
1 1	*i* 01	*j* (11)	*k* 01	*l* 10
1 0	*m* 11	*n* 11	*o* (10)	*p* (10)

$Y_{1e}Y_{2e}$

Figure 7.105 Excitation map containing multiple oscillations.

One oscillation occurs when starting in stable state (*f*) when x_1 changes from 0 to 1. The machine sequences to unstable state g, where $Y_{1e}Y_{2e} = 00$. After a delay of Δt, the feedback variables become equal to the excitation variables and the machine enters unstable state c, where $Y_{1e}Y_{2e} = 01$. After a further delay of Δt, the machine enters transient state g and the process repeats, causing the machine to oscillate between state g and state c. This oscillation is represented as (*f*) $\rightarrow g \leftrightarrow c$.

A second oscillation occurs when beginning in state (*j*) and x_1 changes from 0 to 1. The machine sequences to unstable state k, where $Y_{1e}Y_{2e} = 01$. After a delay of Δt, $y_{1f}y_{2f} = Y_{1e}Y_{2e} = 01$ and the machine proceeds to state g. The machine then oscillates between unstable states g *and* c. This oscillation is represented as (*j*) $\rightarrow k \rightarrow g \leftrightarrow c$.

In the synthesis of asynchronous sequential machines, the oscillation phenomenon should be avoided. The machine specifications can be modified slightly such that every input vector will provide at least one stable state. This modification should not drastically alter the general functional operation of the machine.

7.3.5 Races

In the analysis of synchronous sequential machines, if a change of state occurs between two states with nonadjacent state codes, then the machine may sequence through a transient state before entering the destination stable state. If the transient state contains a Moore-type output, then a transitory erroneous signal may be generated on the output. This glitch results from two or more variables changing state in a single state transition sequence in which the variables change values at different times.

A similar situation occurs in asynchronous sequential machines. If a single change to the input vector causes two or more excitation variables to change state, then multiple paths exist from the source stable state to the destination stable state. This is called a *race* condition.

There are two types of race conditions: noncritical and critical. A *noncritical race* is one that may cycle through one or more transient states before entering the correct destination stable state. A *critical race* is one that terminates in an incorrect destination stable state. Figure 7.106 shows a Karnaugh map for excitation variables Y_{1e} and Y_{2e} in which there is one noncritical race and one critical race.

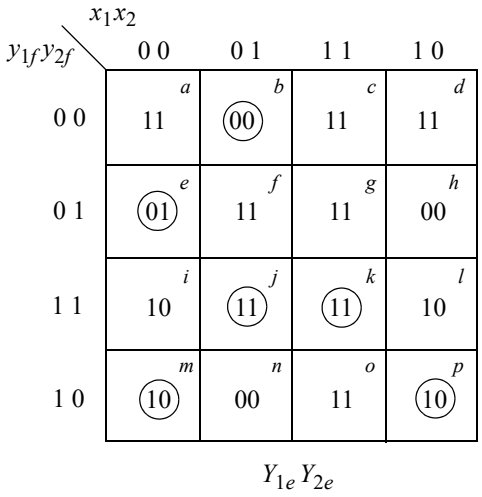


Figure 7.106 Excitation map for $Y_{1e}Y_{2e}$ that has one noncritical race and one critical race.

The noncritical race is described first. Beginning in stable state ⓑ ($Y_{1e}Y_{2e} = 00$), if x_1 changes from 0 to 1, the machine proceeds to transient state c ($Y_{1e}Y_{2e} = 11$). Since both excitation variables change state, this constitutes a race condition. If Y_{1e} sets before Y_{2e} sets; that is, $Y_{1e}Y_{2e} = 10$, then after a delay of Δt, $y_{1f}y_{2f} = 10$ and the

machine enters transient state o where $Y_{1e}Y_{2e} = 11$. After a further delay of Δt, $y_{1f}y_{2f} = 11$ and the machine enters and remains in stable state ⓚ.

A second path for the race condition is when Y_{2e} sets before Y_{1e} sets; that is, $Y_{1e}Y_{2e} = 01$. After a delay of Δt, the feedback variables become equal to the excitation variables and the machine enters transient state g ($Y_{1e}Y_{2e} = 11$). After a further delay of Δt, $y_{1f}y_{2f} = 11$ and the machine enters and remains in stable state ⓚ.

A third path for the race condition is when both excitation variables change simultaneously from $Y_{1e}Y_{2e} = 00$ to $Y_{1e}Y_{2e} = 11$. After a delay of Δt, the feedback variables become equal to the excitation variables and the machine proceeds directly to the destination stable state ⓚ. The race conditions are summarized as follows and Figure 7.107 provides more detail for the intermediate steps:

ⓑ → c → o → ⓚ
ⓑ → c → g → ⓚ
ⓑ → c → ⓚ

ⓑ → c → o → ⓚ

ⓑ $Y_{1e}Y_{2e} = 00 \rightarrow c$ ($Y_{1e}Y_{2e} = 11$)

↓

$Y_{1e}Y_{2e} = 10$ (Y_{1e} sets before Y_{2e} sets)

↓ Δt_1

$y_{1f}y_{2f} = 10$ (o: $Y_{1e}Y_{2e} = 11$)

↓ Δt_2

$y_{1f}y_{2f} = 11$ (ⓚ: $Y_{1e}Y_{2e} = 11$)

ⓑ → c → g → ⓚ

ⓑ $Y_{1e}Y_{2e} = 00 \rightarrow c$ ($Y_{1e}Y_{2e} = 11$)

↓

$Y_{1e}Y_{2e} = 01$ (Y_{2e} sets before Y_{1e} sets)

↓ Δt_1

$y_{1f}y_{2f} = 01$ (g: $Y_{1e}Y_{2e} = 11$)

↓ Δt_2

$y_{1f}y_{2f} = 11$ (ⓚ: $Y_{1e}Y_{2e} = 11$)

(Continued on next page)

Figure 7.107 Three possible paths for the state transition sequence ⓑ → ⓚ.

$\textcircled{b} \rightarrow c \rightarrow \textcircled{k}$

$\textcircled{b}\ Y_{1e}Y_{2e} = 00 \rightarrow c\ (Y_{1e}Y_{2e} = 11;\ Y_{1e}$ and Y_{2e} change state simultaneously)

$\downarrow \Delta t$

$y_{1f}y_{2f} = 11\ (\textcircled{k}\colon Y_{1e}Y_{2e} = 11)$

Figure 7.107 (Continued)

Thus, three state transition sequences are possible, where each sequence results from a different propagation delay time through the excitation variable storage elements. Although a race condition exists, stable state $\textcircled{k}$ is the destination state for all three sequences. This is referred to as a *noncritical race* condition, since the destination stable state is the same regardless of the path taken.

There is also one critical race condition in Figure 7.106 when — beginning in stable state $\textcircled{b}$ — x_2 changes from 1 to 0. The machine then proceeds to transient state a ($Y_{1e}Y_{2e} = 11$). If Y_{1e} sets before Y_{2e} sets; that is, $Y_{1e}Y_{2e} = 10$, then after a delay of Δt, $y_{1f}y_{2f} = 10$ and the machine enters stable state $\textcircled{m}$ where $Y_{1e}Y_{2e} = 10$.

A second path for the race condition is when Y_{2e} sets before Y_{1e} sets; that is, $Y_{1e}Y_{2e} = 01$. After a delay of Δt, the feedback variables become equal to the excitation variables and the machine enters stable state $\textcircled{e}$ ($Y_{1e}Y_{2e} = 01$).

A third path for the race condition is when both excitation variables change simultaneously from $Y_{1e}Y_{2e} = 00$ to $Y_{1e}Y_{2e} = 11$. After a delay of Δt, the feedback variables become equal to the excitation variables and the machine proceeds to transient state i ($Y_{1e}Y_{2e} = 10$). After a further delay of Δt, the feedback variables become equal to the excitation variables and the machine enters stable state $\textcircled{m}$ ($Y_{1e}Y_{2e} = 10$). The race conditions are summarized as follows:

$\textcircled{b} \rightarrow a \rightarrow \textcircled{m}$
$\textcircled{b} \rightarrow a \rightarrow \textcircled{e}$
$\textcircled{b} \rightarrow a \rightarrow i \rightarrow \textcircled{m}$

The single change to the input vector results in a race condition because both excitation variables change value ($Y_{1e}Y_{2e} = 00$ to 11). However, since the destination stable state cannot be predicted, the machine will proceed to either stable state $\textcircled{m}$ or to stable state $\textcircled{e}$. This is termed a *critical race* condition and must be avoided.

Races can be avoided when it is possible to direct the machine through intermediate unstable states before reaching the destination stable state. This can be achieved by utilizing some of the unspecified entries in the excitation map. Also, it may be possible to add rows to the excitation map without increasing the number of excitation and feedback variables.

7.4 Synthesis of Asynchronous Sequential Machines

The synthesis (design) of asynchronous sequential machines is one of the most interesting and certainly the most challenging concepts of sequential machine design. In many situations, a synchronous clock is not available. The interface between an input/output processor (IOP) — or channel — and an input/output (I/O) subsystem control unit is an example of an asynchronous condition. Many large computer channels communicate with an I/O subsystem by means of a signal interlocking protocol on the interface.

The control unit requests a word of data during a write operation by asserting an identifying signal called a "tag-in signal." The channel then places the word on the data bus and asserts an acknowledging tag-out signal. The device control unit accepts the data then deasserts the in-tag, allowing the channel to deassert the corresponding out-tag, completing the data transfer sequence for one word. An analogous situation occurs for a read operation in which the tag-in signal now indicates that a word is available on the data bus for the channel. The channel accepts the word and responds with the tag-out signal.

The data transfer sequence for the write and read operations was initiated, executed, and completed without utilizing a synchronizing clock signal. This technique permits not only a higher data transfer rate between the channel and an I/O device, but also allows the channel to communicate with I/O devices having a wide range of data transfer rates. The interface control logic in the device control unit is usually implemented as an asynchronous sequential machine. Even in large synchronous systems, it is often advantageous to allow certain subsystems to operate in an asynchronous manner, thereby increasing the overall speed of the system.

Transient signals are handled differently in the synthesis procedure for asynchronous machines. Techniques will be presented in this chapter to synthesize fundamental-mode asynchronous sequential machines irrespective of the varying delays of circuit components.

7.4.1 Synthesis Procedure

The synthesis procedure for asynchronous sequential machines is similar in many respects to that described for synchronous sequential machines. This section develops a systematic method for the synthesis of fundamental-mode asynchronous sequential machines using *SR* latches as the storage elements. The machine operation is specified as a timing diagram and/or verbal statements.

The synthesis procedure is summarized below. The key step in the synthesis of synchronous sequential machines is the derivation of the state diagram, whereas the key step in the synthesis of asynchronous sequential machines is the derivation of the primitive flow table. The synthesis procedure will result in a machine that operates according to the prescribed specifications. The solution, however, may not be unique.

1. **State diagram** The machine specifications are converted into a state diagram. A timing diagram and/or a verbal statement of the machine specifications is converted into a precise delineation which specifies the machine's operation for all applicable input sequences. This step is not a necessary requirement and is usually omitted; however, the state diagram characterizes the machine's operation in a graphical representation and adds completeness to the synthesis procedure.

2. **Primitive flow table** The machine specifications are converted to a state transition table called a "primitive flow table." This is the least methodical step in the synthesis procedure and the most important. The primitive flow table depicts the state transition sequences and output assertions for all valid input vectors. The flow table must correctly represent the machine's operation for all applicable input sequences, even those that are not initially apparent from the machine specifications.

3. **Equivalent states** The primitive flow table may have an inordinate number of rows. The number of rows can be reduced by finding equivalent states and then eliminating redundant states. If the machine's operation is indistinguishable whether commencing in state Y_i or state Y_j, then one of the states is redundant and can be eliminated. The flow table thus obtained, is a *reduced primitive flow table*.

4. **Merger diagram** The merger diagram graphically portrays the result of the merging process in which an attempt is made to combine two or more rows of the reduced primitive flow table into a single row. The result of the merging technique is analogous to that of finding equivalent states; that is, the merging process can also reduce the number of rows in the table and hence, reduce the number of feedback variables that are required. Fewer feedback variables will result in a machine with less logic and, therefore, less cost.

5. **Merged flow table** The merged flow table is constructed from the merger diagram. The table represents the culmination of the merging process in which two or more rows of a primitive flow table are replaced by a single equivalent row which contains one stable state for each merged row.

6. **Excitation maps and equations** An excitation map is generated for each excitation variable. Then the transient states are encoded, where applicable, to avoid critical race conditions. Appropriate assignment of the excitation variables for the transient states can minimize the δ next-state logic for the excitation variables. The operational speed of the machine can also be established at this step by reducing the number of transient states through which the machine must sequence during a cycle. Then the excitation equations are derived from the excitation maps. All static-1 and static-0 hazards are eliminated

from the network for a sum-of-products or product-of-sums implementation, respectively.

7. **Output maps and equations** An output map is generated for each machine output. Output values are assigned for all nonstable states so that no transient signals will appear on the outputs. In this step, the speed of circuit operation can also be established. Then the output equations are derived from the output maps ensuring that all outputs will be free of momentary false outputs.

8. **Logic diagram** The logic diagram is implemented from the excitation and output equations using an appropriate logic family.

7.4.2 Synthesis Examples

This section presents examples illustrating the synthesis of asynchronous sequential machines in their entirety. The machine specifications for these examples will consist of a verbal description only or a verbal description in conjunction with a representative timing diagram.

If the verbal description is the only means of conveying the operational characteristics of the machine, then the description must be comprehensive and precise. Sufficient detail must be provided in the description so that no additional information is required to elucidate the machine specifications. If the machine specifications are delineated in sufficient detail, then a timing diagram can be created as a further aid, if necessary.

When defining the operational characteristics by means of a timing diagram with two or more input variables and at least one output variable, a comprehensive representation of machine characteristics is usually prohibitive. This is due, in part, to space limitations for the diagram in order to show the arrangement and relationship of all combinations of the input and output binary variables.

In the following examples, the solution may not be unique. The synthesis specifications will stipulate whether the λ output logic is to be as fast as possible, as slow as possible, or in minimal form. In all examples, however, there must be no race conditions, no static-1 or static-0 hazards, and no momentary false outputs.

Example 7.12 This design has two inputs x_1 and x_2 and one output z_1, as shown in the timing diagram of Figure 7.108. Input x_1 acts as a gate for x_2; that is, x_2 will be gated to output z_1 only if x_1 precedes the assertion of x_2. If x_1 becomes deasserted while x_2 is asserted, then the full width of the x_2 pulse will appear on z_1 — the width of the x_2 pulse will not be decreased.

Primitive flow table The primitive flow table transforms the machine specifications into a tabular representation which specifies the state transition sequences for all valid input combinations. The primitive flow table is characterized by having only one stable state in each row. This allows the output for that row to be uniquely specified as a function of a particular stable state. The stable states are indicated by circled

entries, whereas the unstable, or transient states, are uncircled. The transient states specify the next stable state for a particular state transition sequence. The unspecified entries are indicated by a dash (–) and are the result of the fundamental-mode operation or an input vector that is invalid from a particular stable state.

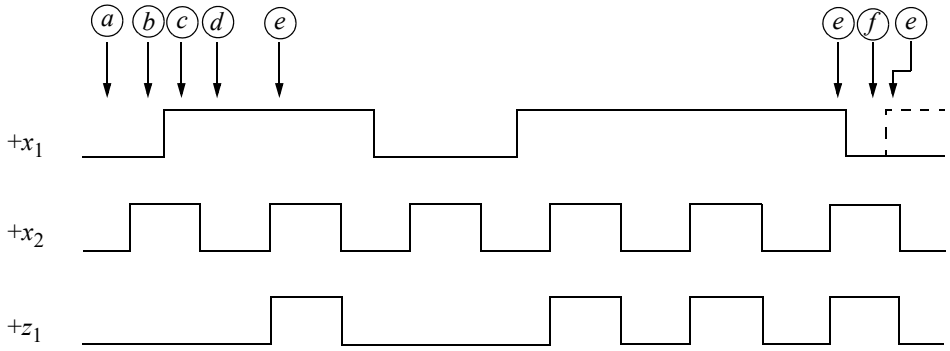


Figure 7.108 Timing diagram for the asynchronous sequential machine of Example 7.12.

Constructing the primitive flow table is a two-pass procedure. The first pass lists all stable states and their associated next states as obtained from the machine specifications such as, a timing diagram and/or a verbal description. These are the more obvious entries. Refer to the partial primitive flow table of Figure 7.109 for the discussion which follows.

Beginning at the leftmost section of the timing diagram, where the initial conditions are specified, proceed left to right assigning a stable state to each different combination of the input vector. The column headings represent the input vector. The table entries specify the stable states, transient states, invalid state transitions which are represented as dashes, and outputs.

The machine begins in stable state ⓐ where $x_1x_2z_1 = 000$. Input x_2 is then asserted, which takes the machine through transient state b, terminating in stable state ⓑ, as shown in the partial primitive flow table. The next change is the assertion of x_1. Since the assertion of x_1 did not precede the assertion of x_2, output z_1 remains deasserted and the machine enters transient state c and ends in stable state ⓒ, where $x_1x_2z_1 = 110$.

The next change occurs when x_2 becomes deasserted, causing the machine to enter stable state ⓓ where output z_1 remains deasserted. The second x_2 pulse is then asserted, which sequences the machine to state ⓔ. Because the assertion of x_1 preceded the assertion of x_2, output z_1 is asserted. If x_1 is still asserted when x_2 becomes deasserted, then this is equivalent to state ⓓ, where $x_1x_2z_1 = 100$; therefore, the machine proceeds to state ⓓ.

Consider stable state ⓔ again. If x_1 becomes deasserted while x_2 is asserted, then this represents a new state in which z_1 remains asserted due to the machine

specifications. This new state is labeled stable state (f). This completes the first pass through the timing diagram and is shown in the partial primitive flow table of Figure 7.109.

x_1x_2 0 0	0 1	1 1	1 0	z_1
(a)	b	–		0
	(b)	c	–	0
–		(c)	d	0
	–	e	(d)	0
–	f	(e)	d	1
	(f)		–	1

Figure 7.109 Partial primitive flow table for the asynchronous sequential machine of Example 7.12.

The second pass establishes the entries for any unspecified transient states and may necessitate creating additional rows. The second pass also establishes subsequences that are not expressly specified by the timing diagram. Careful consideration must be given to these subsequences to ensure that the machine operates according to the functional specifications. The complete primitive flow table is shown in Figure 7.110 and its construction is described in the following paragraphs.

x_1x_2 0 0	0 1	1 1	1 0	z_1
(a)	b	–	d	0
a	(b)	c	–	0
–	b	(c)	d	0
a	–	e	(d)	0
–	f	(e)	d	1
a	(f)	e	–	1

Figure 7.110 Complete primitive flow table for Example 7.12.

In stable state ⓐ, if x_1 becomes asserted while x_2 remains deasserted, then this is equivalent to state ⓓ, where z_1 is deasserted. Therefore, an entry of unstable state *d* is entered in column $x_1x_2 = 10$. Also in state ⓐ, a fundamental-mode model does not allow simultaneous changes to the inputs; that is, a change from $x_1x_2 = 00$ to 11. Therefore, a "don't care" entry is inserted in column $x_1x_2 = 11$.

In state ⓑ of the second row of the partial primitive flow table, if x_2 becomes deasserted, then this represents the same conditions as shown in state ⓐ; therefore, an entry of *a* is inserted in that location. Due to a double change of inputs from $x_1x_2 = 01$ to 10, a "don't care" entry is inserted in column $x_1x_2 = 10$.

In state ⓒ of the third row of the Figure 7.109, if x_1 becomes deasserted, then this is equivalent to state ⓑ, where $x_1x_2z_1 = 010$; therefore, an entry of *b* is inserted in column $x_1x_2 = 01$. A "don't care" is inserted in column $x_1x_2 = 00$ due to a simultaneous change of inputs.

In state ⓓ of the fourth row, if x_1 becomes deasserted, then this is equivalent to the conditions for stable state ⓐ, which places an entry of *a* in column $x_1x_2 = 00$ together with a "don't care" in column $x_1x_2 = 01$. The fifth row containing stable state ⓔ has already been defined except for the "don't care" entry in column $x_1x_2 = 00$.

Consider now stable state ⓕ in the sixth row in conjunction with the timing diagram. If x_2 becomes deasserted, then this is equivalent to state ⓐ, where $x_1x_2z_1 = 000$. All situations must be considered when constructing a primitive flow table. Therefore, in state ⓕ, it is possible for x_1 to become asserted while x_2 is still asserted. The machine then returns to state ⓔ and maintains output z_1 active.

Generation of the primitive flow table is the most critical step in the synthesis of asynchronous sequential machines. If the primitive flow table is correctly constructed, then the remaining steps in the synthesis procedure, if properly executed, will result in a machine that meets the performance criteria of the machine specifications. If, however, the primitive flow table does not delineate all transitions for every possible input sequence, then the remaining steps, although correct in themselves, will result in a machine that does not operate according to the machine specifications.

Equivalent states Eliminating redundant states from a primitive flow table will generate a reduced primitive flow table which is equivalent to the original table. A primitive flow table — unless already containing a minimal number of rows — can be reduced to provide a table that contains fewer rows than the original table and yet completely characterizes the operation of the machine.

The definition of equivalence can be stated as shown below for asynchronous sequential machines. Two stable states Y_i and Y_j in a primitive flow table are defined to be equivalent if and only if all of the following rules apply:

1. The stable states have the same input vector; that is, the states are in the same column.

2. The outputs associated with Y_i and Y_j have the same value; that is, $Z_r(Y_i) = Z_r(Y_j)$.

3. The next states for Y_i and Y_j are the same or equivalent for every column in the two rows of the primitive flow table. That is, for each input combination in the rows of the two stable states, the following is observed:

   (a) Identical or equivalent state names, or
   (b) Two dashes.

In Figure 7.110, the only states that are potentially equivalent are:

ⓑ and ⓕ

ⓒ and ⓔ

Stable states ⓑ and ⓕ have different outputs, contradicting Rule 2 for equivalence. Therefore, states ⓑ and ⓕ are not equivalent. The same is true for states ⓒ and ⓔ; therefore, the primitive flow of Figure 7.110 is also the reduced primitive flow table. Each row of the primitive flow table corresponds to a different combination of feedback variables.

Merger diagram Merging is a process of combining two or more rows of a reduced primitive flow table into a single row. The merging process reduces the number of rows in the flow table and, thus, may reduce the number of feedback variables. A reduction of feedback variables decreases the number of storage elements in the machine. The stable states in the rows that are merged are entered in the same location in the single merged row. When merging, the outputs associated with each row are disregarded. Thus, two rows of a reduced primitive flow table can merge, regardless of the output values of the rows under consideration.

Two rows can merge into a single row if the entries in the same column of each row satisfy the requirements of one of the following three merging rules:

1. Identical state entries, either stable or unstable
2. A state entry and a "don't care"
3. Two "don't care" entries

Merging rule 1 specifies that there must be no conflict in state name entries in the same column of the two rows under consideration. That is, two different states cannot both be active for the same input vector and the same combination of feedback variables. Three or more rows can merge into a single row if and only if all pairs of rows satisfy the conditions of the merging rules.

A stable state entry and an unstable state entry of the same name in the same column of two different rows are merged as the stable state entry, since the resultant state must be stable. Thus, in many cases, the merging process eliminates the transient unstable states. Two identical unstable states merge as an unstable state. Both stable and unstable states merge with a "don't care" entry as a stable and unstable state, respectively.

The merging process is facilitated by means of a merger diagram. The merger diagram depicts all rows of the reduced primitive flow table in a graphical representation. Each row of the table is portrayed as a vertex in the merger diagram. The rows in which merging is possible are connected by lines.

A set of rows in a reduced primitive flow table can merge into a single row if and only if the rows are strongly connected. That is, every row in the set must merge with all other rows in the set. For example, Figure 7.111 illustrates strongly connected sets of three, four, and five rows each. Each set in Figure 7.111 is a maximal compatible set, in which compatible pairs {ⓐ, ⓑ}, and {ⓑ, ⓒ} implies the compatibility of {ⓐ, ⓒ}. Thus, the transitive property applies to maximal compatible sets. Adding another state to a maximal compatible set negates the transitive property on the new set, unless the set remains strongly connected.

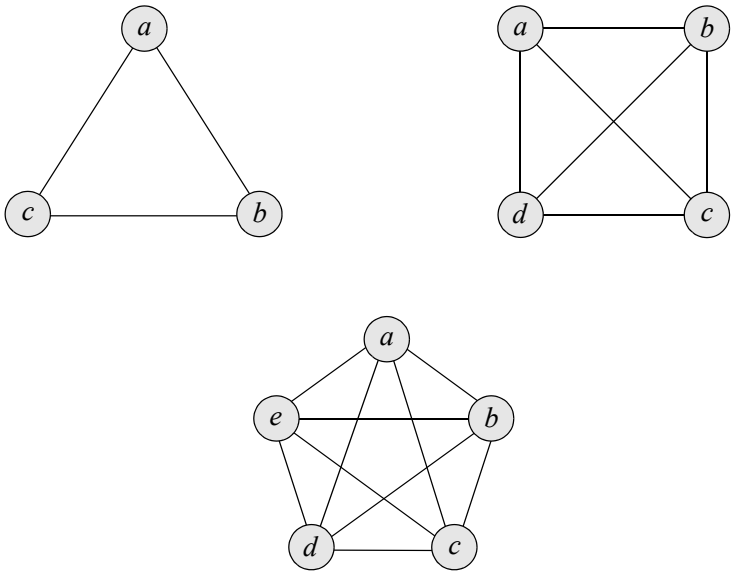


Figure 7.111 Strongly connected sets in which each row in the set can merge with all other rows in the set.

The objective of merging is to combine the maximal number of rows into a single merged row while maintaining the fewest number of merged rows. All strongly connected sets can be combined into single individual rows, one set per row. Figure 7.112 shows the merger diagram for the asynchronous sequential machine of Example 7.12. Using the rules for merging in Figure 7.110, it is seen that rows ⓐ and ⓑ can merge because there is no conflict in state names. Rows ⓐ and ⓒ can also merge as well as

rows ⓐ and ⓓ. Therefore, rows ⓐ, ⓑ, and ⓒ can merge into a single row. Row ⓐ cannot merge with any other row. In a similar manner, it is evident that rows ⓓ, ⓔ, and ⓕ can also merge into a single row.

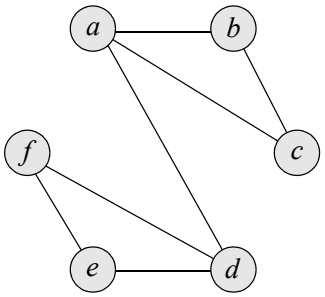


Figure 7.112 Merger diagram for the asynchronous sequential machine of Example 7.12.

Merged flow table The next step in the synthesis procedure is the generation of a merged flow table. The merged flow table specifies the operational characteristics of the machine in a manner analogous to that of the primitive flow table and the reduced primitive flow table, but in a more compact form. Each row in a merged flow table represents a set of maximal compatible rows.

Most unspecified entries in the reduced primitive flow table are replaced with either a stable or an unstable state entry in the merged flow table. Since more than one stable state is usually present in each row of a merged flow table, many state transition sequences do not cause a change to the feedback variables. Thus, faster operational speed is realized.

The merged flow table is derived from the merger diagram in conjunction with the reduced primitive flow table. The partition of sets of maximal compatible rows obtained from the merger diagram dictates the minimal number of rows in the merged flow table.

To merge rows ⓐ, ⓑ, and ⓒ into a single row, simply transcribe each row, one row at a time, from the reduced primitive flow table to the merged flow table. For example, row ⓐ transfers as shown in Figure 7.113(a). Then transfer row ⓑ to the same row in the merged flow table, superimposing row ⓑ on row ⓐ, as shown in Figure 7.113(b). Finally, transfer row ⓒ to the merged flow table, superimposing row ⓒ on previously transferred rows ⓐ and ⓑ, as shown in Figure 7.113(c). Notice that no conflict in state names occurs during the merging of rows ⓐ, ⓑ and ⓒ into a single merged row. This is a necessary requirement and exemplifies the rationale for merging. The merged flow table is shown in Figure 7.114.

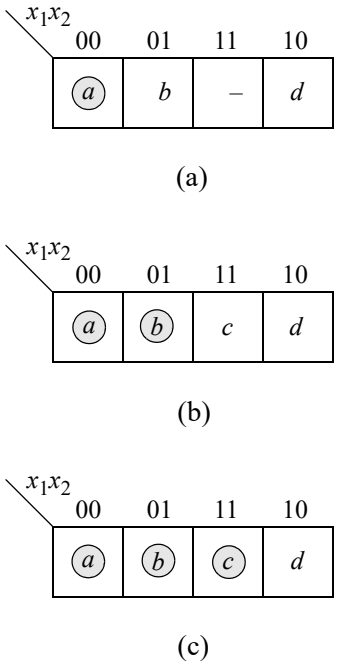


Figure 7.113 Top row of the merged flow table illustrating the transcribing of rows ⓐ, ⓑ, and ⓒ singly, from the reduced primitive flow table to the merged flow table: (a) row ⓐ transferred; (b) row ⓑ transferred and superimposed on row ⓐ; and (c) row ⓒ transferred and superimposed on rows ⓐ and ⓑ.

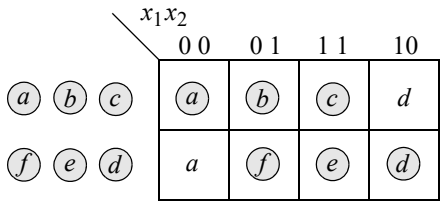


Figure 7.114 Merged flow table constructed from the merger diagram and the reduced primitive flow table.

Excitation map and equation The merged flow table is the foundation from which the excitation maps are derived. The excitation map directly formulates the equations that are necessary to implement the logic for the excitation variables.

Each row of the merged flow table is assigned a unique combination of values for the feedback (or secondary) variables. The values of the feedback variables for each

row then determine the values of the excitation variables for the stable state entries in the corresponding row of the excitation map.

Recall that a machine is stable in a particular state when the feedback variables are equal to the excitation variables. Thus, the values assigned to the excitation variables in each stable state of the excitation map are identical to those of the feedback variables for that row. The placement of each stable state of the excitation map represents a one-to-one mapping of the stable states in the corresponding locations of the merged flow table. The entries in the excitation map that correspond to unstable states in the merged flow table specify the next state to which the machine will sequence due to a change in the input vector. The excitation map for Y_{1e} is shown in Figure 7.115 and the equation for Y_{1e} is shown in Equation 7.38.

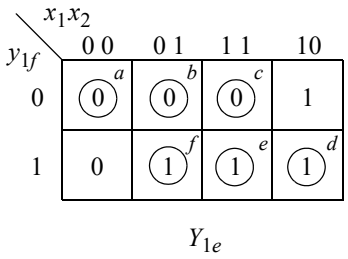


Figure 7.115 Excitation map for the asynchronous sequential machine of Example 7.12.

$$Y_{1e} = x_1x_2' + x_2y_{1f} + x_1y_{1f} \tag{7.38}$$

Hazard cover

Output map and equation The next step in the synthesis procedure is to assign values to the output variables. A Karnaugh map — referred to as an output map — facilitates this process. This step utilizes the results of two previous operations: the merged flow table and the reduced primitive flow table.

The merged flow table indicates the location of all stable states. Since the outputs are associated with stable states, the merged flow table indicates the location of the stable state output variables in their respective output maps. The merged flow table defines the format of the output map, which is constructed directly from the merged flow table and the reduced primitive flow table. The reduced primitive flow table specifies the output values of the corresponding stable states in the output map. The merged flow table, the excitation map, and the output map have the same number of inputs — and thus, the same number of columns — and the same number of feedback variables,

necessitating the same number of rows. The values assigned to the input variables and the feedback variables are the same in both the excitation map and the output map.

The speed of circuit operation can be established during this phase. If different output values are associated with the initial and destination stable states for a state transition with only one intermediate state, then the intermediate unstable state can be assigned a value that is equal to either the initial or the destination stable state output value.

By assigning appropriate values to the unstable states in the output map, the outputs can be made to change value as soon as possible or as late as possible for a particular state transition. If the output value of the initial stable state is assigned to the intermediate state, then the change to the output is delayed until the machine enters the destination stable state. If, however, the output value of the destination stable state is assigned to the intermediate state, then the output value changes before the machine reaches the destination stable state.

The output map for z_1 is shown in Figure 7.116. Since stable state ⓒ sequences to stable state ⓓ and both states have an output value of $z_1 = 0$ — as shown in the primitive flow table — the intermediate state of $x_1x_2y_{1f} = 100$ must contain a value of 0; otherwise, there would be a glitch on output z_1. The same is true for the sequence ⓓ Æ ⓐ, necessitating a 0 in location $x_1x_2y_{1f} = 001$. The equation for output z_1 is shown in Equation 7.39. The logic diagram is constructed from the excitation equation as a sum of products and the output equation, which is one of the product terms from the d next-state logic.

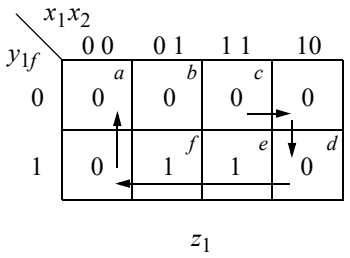


Figure 7.116 Output map for z_1 for the asynchronous sequential machine of Example 7.12.

$$z_1 = x_2 y_{1f} \tag{7.39}$$

Example 7.13 A timing diagram for an asynchronous sequential machine that has one input x_1 and two outputs z_1 and z_2 is shown in Figure 7.117. Output z_1 is asserted for the duration of every second x_1 pulse; output z_2 is asserted for the duration of every second z_1 pulse. The outputs are to respond as fast as possible to changes that occur on input x_1.

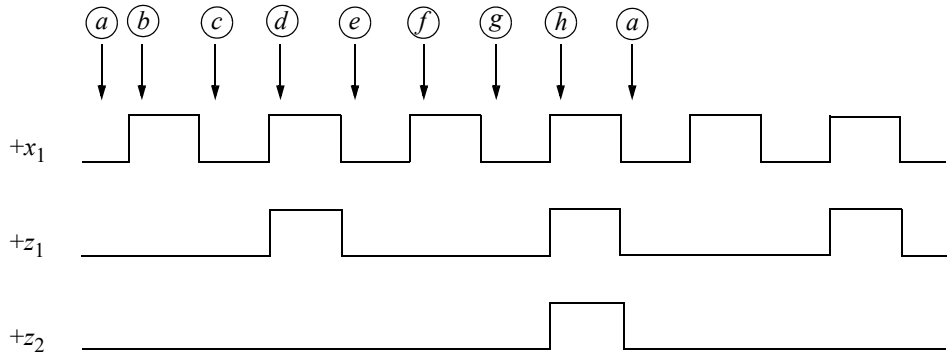


Figure 7.117 Timing diagram for the asynchronous sequential machine of Example 7.13.

The primitive flow table is shown in Figure 7.118. There are no equivalent states and no rows can merge. Therefore, the primitive flow table is also the reduced primitive flow table and the merged flow table. Since there are eight rows, there are three feedback variables y_{1f}, y_{2f}, and y_{3f}. The combined excitation map for excitation variables Y_{1e}, Y_{2e}, and Y_{3e} is shown in Figure 7.119.

x_1	0	1	z_1	z_2
(a)	b		0	0
c	(b)		0	0
(c)	d		0	0
e	(d)		1	0
(e)	f		0	0
g	(f)		0	0
(g)	h		0	0
a	(h)		1	1

Figure 7.118 Primitive flow table for Example 7.13.

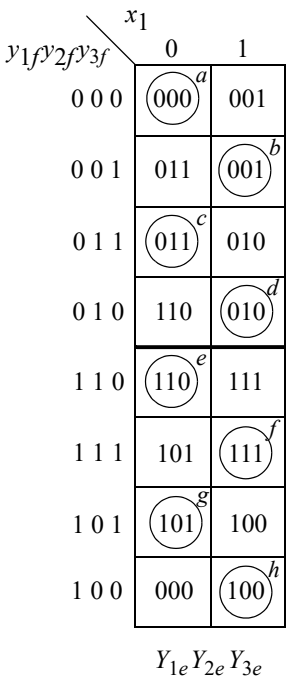
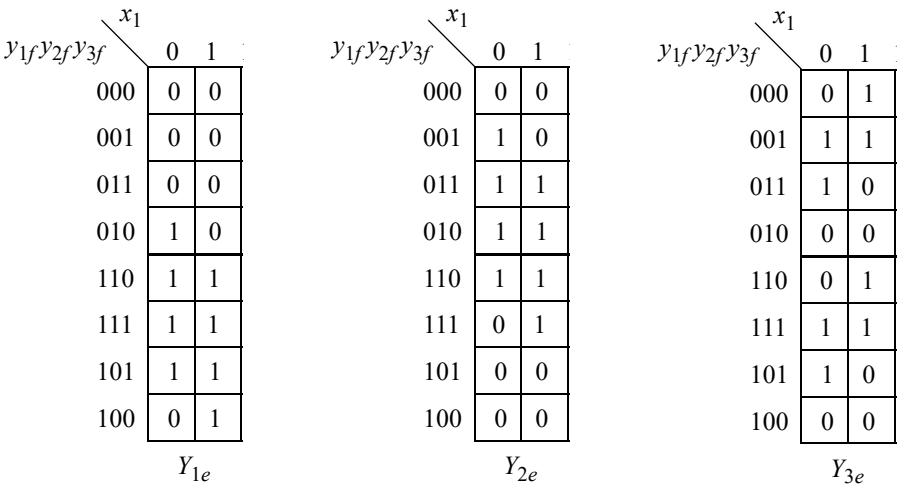


Figure 7.119 Combined excitation map for the asynchronous sequential machine of Example 7.13.

The individual excitation maps are shown in Figure 7.120 and are derived directly from the combined excitation maps by copying the individual columns of the corresponding excitation variables to the appropriate map. The equations for the excitation variables are shown in Equation 7.40 and include redundant prime implicants to negate any potential hazards.

The output maps are shown in Figure 7.121 and are derived from the reduced primitive flow table. In state ⓒ, output $z_1 = 0$; in state ⓓ, output $z_1 = 1$. Therefore, as the machine sequences from state ⓒ to state ⓓ transient state *d* can have an output value for z_1 that is "don't care" — either 0 or 1. However, the machine specifications stipulate that the outputs must respond to input changes as quickly as possible. Therefore, an entry of 1 is inserted in location $y_{1f}y_{2f}y_{3f}x_1 = 0111$ in order to assert output z_1 as soon as possible. The same is true for z_1 in location $y_{1f}y_{2f}y_{3f}x_1 = 0100$ by inserting a value of $z_1 = 0$ and location $y_{1f}y_{2f}y_{3f}x_1 = 1000$ by inserting a 0.

The same rationale applies to output z_2 also in location $y_{1f}y_{2f}y_{3f}x_1 = 1011$ by inserting a value of 1 and in location $y_{1f}y_{2f}y_{3f}x_1 = 1000$ by inserting a value of 0. The logic diagram is shown in Figure 7.122 using AND gates and OR gates. Since all AND gates have inputs connected to the feedback variables, it is appropriate to connect an active-low reset signal to the AND gates; otherwise, the latches may assume a random state when power is turned on. A reset pulse guarantees that the machine will commence operation in state ⓐ.



$y_{1f}y_{2f}y_{3f}$ \ x_1	0	1
000	0	0
001	0	0
011	0	0
010	1	0
110	1	1
111	1	1
101	1	1
100	0	1

Y_{1e}

$y_{1f}y_{2f}y_{3f}$ \ x_1	0	1
000	0	0
001	1	0
011	1	1
010	1	1
110	1	1
111	0	1
101	0	0
100	0	0

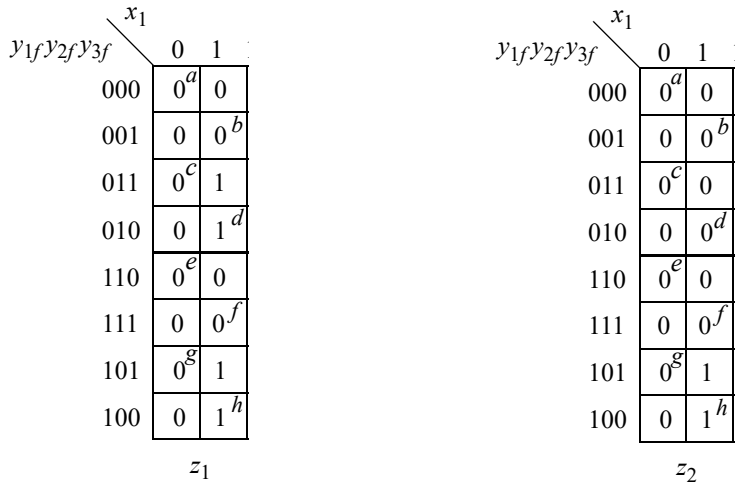
Y_{2e}

$y_{1f}y_{2f}y_{3f}$ \ x_1	0	1
000	0	1
001	1	1
011	1	0
010	0	0
110	0	1
111	1	1
101	1	0
100	0	0

Y_{3e}

Figure 7.120 Individual excitation maps for the asynchronous sequential machine of Example 7.13.

$$
\begin{aligned}
Y_{1e} &= y_{1f}x_1 + y_{1f}y_{3f} + y_{2f}y_{3f}'x_1' + y_{1f}y_{2f} \\
Y_{2e} &= y_{2f}y_{3f}' + y_{2f}x_1 + y_{1f}'y_{3f}x_1' + y_{1f}'y_{2f} \\
Y_{3e} &= y_{3f}x_1' + y_{1f}'y_{2f}'x_1 + y_{1f}y_{2f}x_1 + y_{1f}'y_{2f}'y_{3f} + y_{1f}y_{2f}y_{3f}
\end{aligned} \tag{7.40}
$$



$y_{1f}y_{2f}y_{3f}$ \ x_1	0	1
000	0^a	0
001	0	0^b
011	0^c	1
010	0	1^d
110	0^e	0
111	0	0^f
101	0^g	1
100	0	1^h

z_1

$y_{1f}y_{2f}y_{3f}$ \ x_1	0	1
000	0^a	0
001	0	0^b
011	0^c	0
010	0	0^d
110	0^e	0
111	0	0^f
101	0^g	1
100	0	1^h

z_2

Figure 7.121 Output maps for Example 7.13.

$$
\begin{aligned}
z_1 &= y_{1f}'y_{2f}x_1 + y_{1f}y_{2f}'x_1 \\
z_2 &= y_{1f}y_{2f}'x_1
\end{aligned} \tag{7.41}
$$

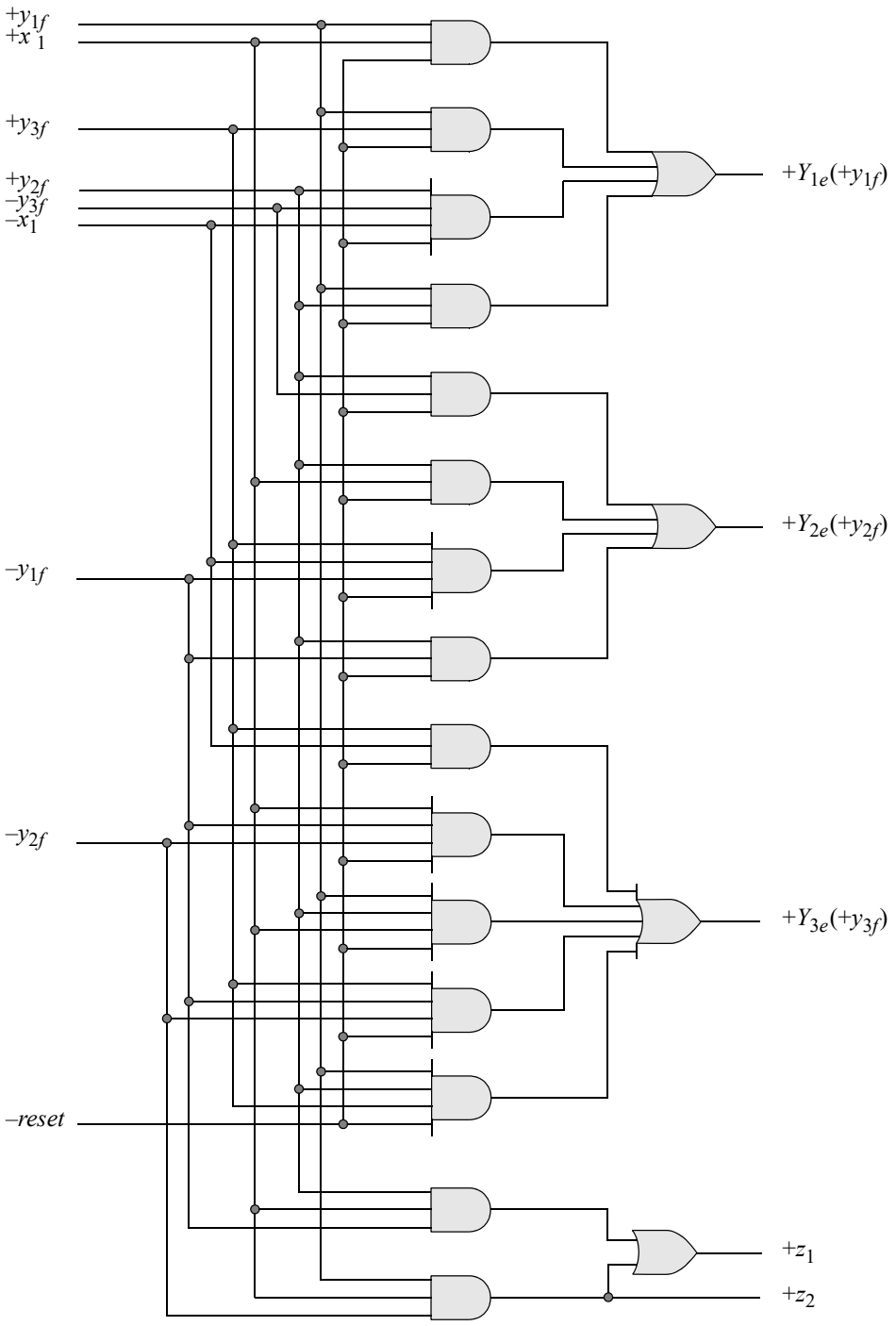


Figure 7.122 Logic diagram for the asynchronous sequential machine of Example 7.13.

Example 7.14 The timing diagram for an asynchronous sequential machine is shown in Figure 7.123 with one input x_1 and two outputs z_1 and z_2. The outputs will be implemented with the least amount of logic which may not necessarily be the fastest response time.

The primitive flow table is shown in Figure 7.124 in which there are no equivalent states, because each state has different outputs; therefore, this is also the reduced primitive flow table. The primitive flow table is generated directly from the timing diagram beginning in stable state ⓐ, where $x_1z_1z_2 = 000$. A new stable state is inserted for each new combination of the variables $x_1z_1z_2$. The merger diagram is shown in Figure 7.125 in which no rows can merge because both columns for x_1 have at least one different state in each selected row. The combined excitation map — obtained from the primitive flow table — and the individual excitation maps are shown in Figure 7.126. The excitation equations are shown in Equation 7.42. The combined output map and the individual output maps are shown in Figure 7.127. The output equations are shown in Equation 7.43. The logic diagram is shown in Figure 7.128.

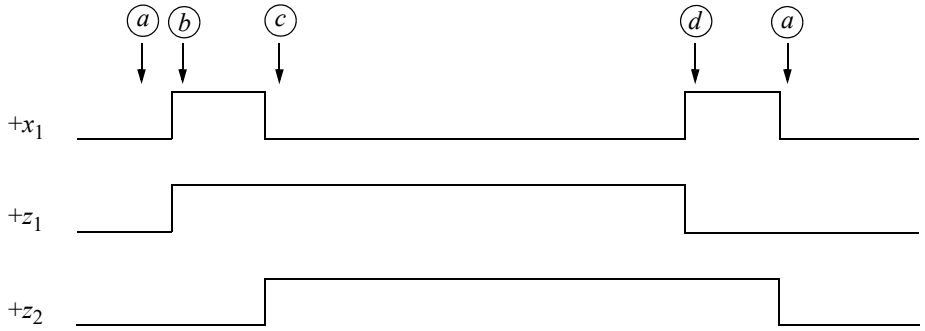


Figure 7.123 Timing diagram for the asynchronous sequential machine of Example 7.14.

x_1	0	1	z_1	z_2
	ⓐ	b	0	0
	c	ⓑ	1	0
	ⓒ	d	1	1
	a	ⓓ	0	1

Figure 7.124 Primitive flow table for Example 7.14.

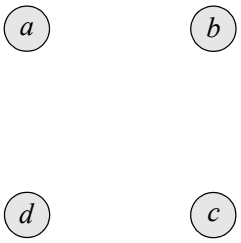


Figure 7.125 Merger diagram for Example 7.14.

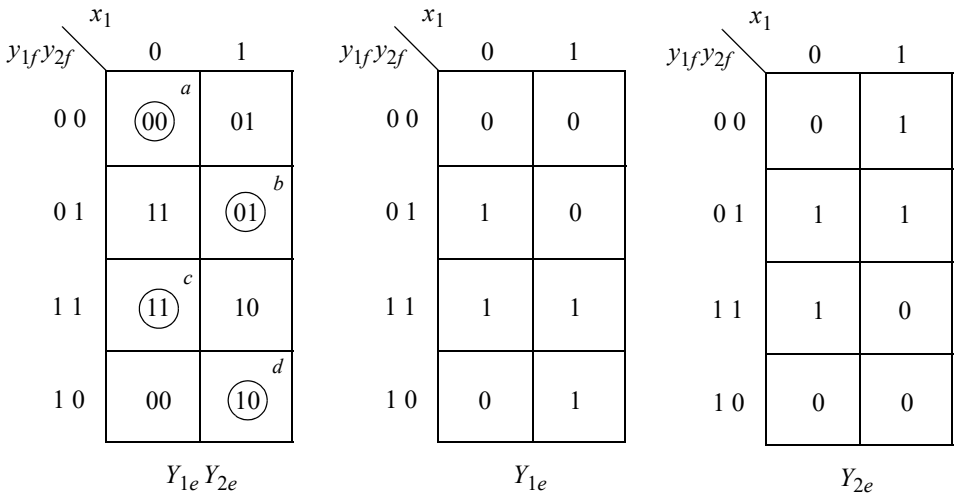


Figure 7.126 Combined excitation map and individual excitation maps for Example 7.14.

$$Y_{1e} = x_1' y_{2f} + x_1 y_{1f} + y_{1f} y_{2f}$$
$$Y_{2e} = x_1' y_{2f} + x_1 y_{1f}' + y_{1f}' y_{2f} \tag{7.42}$$

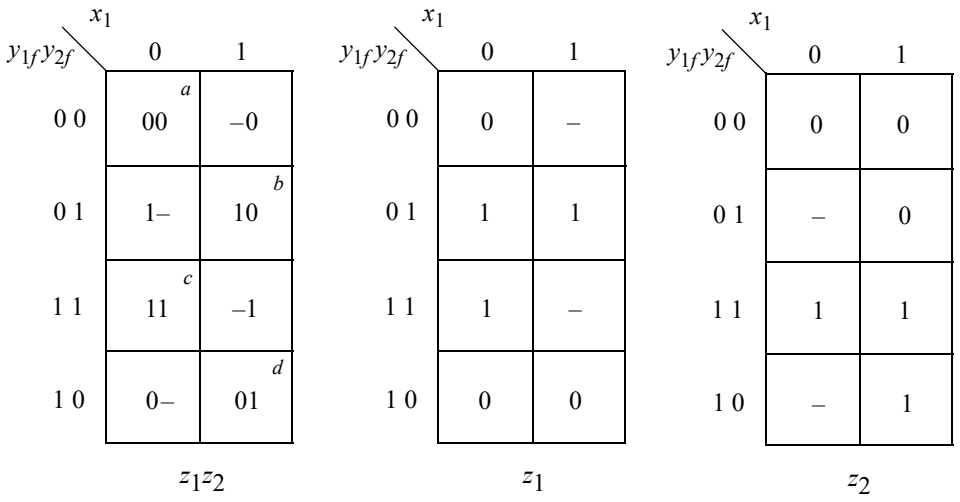


Figure 7.127 Combined output map and individual output maps for Example 7.14.

$$z_1 = y_{2f}$$
$$z_2 = y_{1f} \tag{7.43}$$

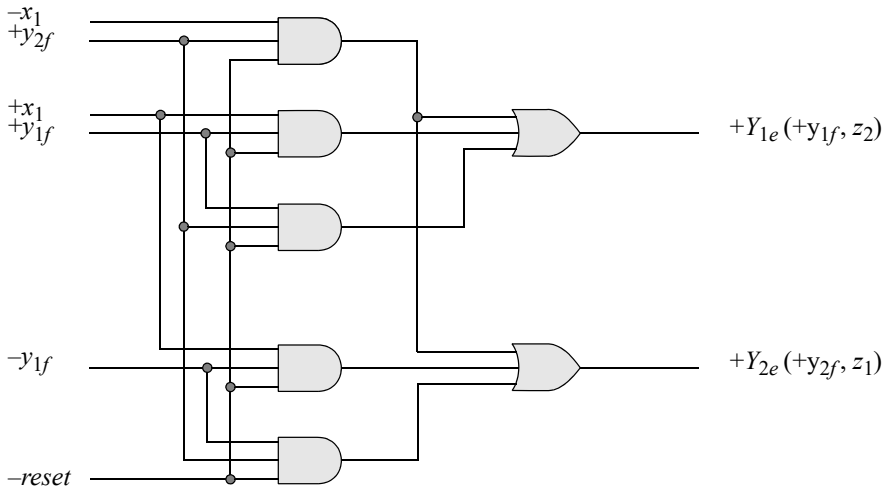


Figure 7.128 Logic diagram for Example 7.14.

Example 7.15 This example will concentrate on constructing a transition diagram. Methods will now be presented for assigning values to the feedback variables so that each state transition sequence will involve a change of only one excitation variable between logically contiguous rows in the cycle. Two changes of excitation variables will still be allowed, however, provided that the resulting race condition does not generate a critical race. Each transition in the merged flow table must be examined to ensure that the assigned feedback values differ by a change of only one variable between the row containing the beginning stable state and each successive pair of rows in the cycle, including the row containing the destination stable state. That is, the feedback variables must be assigned adjacent p-tuples for each successive row in the cycle.

Consider the merged flow table of Figure 7.129 consisting of three rows labeled 1, 2, and 3. Inspection of column $x_1x_2 = 00$ indicates that the feedback values assigned to row 1 must be adjacent to those assigned to row 2, to provide a race-free cycle from state ⓑ or ⓖ through unstable state a to state ⓐ. The same adjacency requirement is observed in column $x_1x_2 = 11$.

	x_1x_2 00	01	11	10
1	ⓐ	b	ⓔ	c
2	a	ⓑ	e	ⓖ
3	ⓕ	b	ⓓ	ⓒ

Figure 7.129 Merged flow table for an asynchronous sequential machine for Example 7.15.

Examination of column $x_1x_2 = 10$ identifies an adjacency requirement between rows 1 and 3. This requirement accommodates a transition from either state ⓐ or ⓔ through unstable state c to state ⓒ.

Similarly, rows 2 and 3 must have adjacent feedback values to realize a transition from state ⓓ or ⓕ through unstable state b to state ⓑ. The observation of the above adjacency requirements for race-free operation are summarized as follows:

Column $x_1x_2 = 00$: Rows 1 and 2 must be adjacent.
Column $x_1x_2 = 01$: Rows 1 and 2 must be adjacent.
Rows 2 and 3 must be adjacent.
Column $x_1x_2 = 11$: Rows 1 and 2 must be adjacent.
Column $x_1x_2 = 10$: Rows 1 and 3 must be adjacent.

In column $x_1x_2 = 01$, only one stable state is specified; therefore, a critical race condition is impossible. A noncritical race, however, may occur from row 1 or 3 to row 2,

depending on the assigned feedback values. Since noncritical races do not present a problem in the deterministic operation of an asynchronous sequential machine, the assignment of values for the feedback variables is not crucial.

The preceding requirements listed for each column specify the adjacencies that are needed to establish race-free operation for the indicated transitions. The same information is portrayed graphically in the *transition diagram* of Figure 7.130. For a merged flow table containing three rows, the rows are listed in a triangular arrangement. Each row is represented by a vertex. Each pair of rows, for which adjacency is required, is connected by a line. The connecting line indicates a requisite transition between the pair of rows.

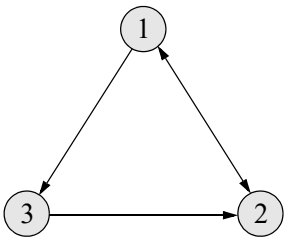


Figure 7.130 Transition diagram for the merged flow table of Figure 7.129.

There is no need to specify the direction of the lines, since adjacency is the only information that is relevant. However, directed lines in a transition diagram may be advantageous in visualizing the sequence of transitions.

The next step is to assign values to the vertices which will represent the values of the feedback variables. Each pair of logically adjacent rows must be assigned adjacent state codes. The transition diagram of Figure 7.130 illustrates that all three rows of the merged flow table must be adjacent. It is obviously not possible to assign 2-tuples to the three rows so that each row is adjacent to all other rows.

To illustrate this impracticability, observe the transition diagram of Figure 7.131. The codes assigned to the state variables in Figure 7.131(a) produce a noncritical race condition for a transition between rows 2 and 3. Figure 7.131(b) generates a critical race for a transition between rows 1 and 3. If the transition diagram contained four row vertices, then a 2-tuple Gray code assignment would realize a race-free operation.

The vertices of triangular or other polygons with an odd number of sides cannot be encoded with adjacent p-tuples for every row. The transition diagram must be altered so that triangular polygons do not appear. Modifying the transition diagram in this way may require more than a minimal number of state variables.

Since a merged flow table containing three rows requires two feedback variables, the addition of a fourth row will not increase the number of feedback variables. The number of rows in a table satisfies the expression of Equation 7.44, where r is the number of rows and p is the number of states, or feedback variables.

$$r = 2^p \tag{7.44}$$

Whether $r = 3$ or 4, the number of feedback variables remains the same. Therefore, a fourth row consisting of unspecified entries will be appended to the bottom row of the merged flow table. The additional row is not associated with any stable state. The unspecified entries will be used, where applicable, to establish intermediate unstable states which will direct the machine to the appropriate destination stable state. Two state variables are required to encode four rows.

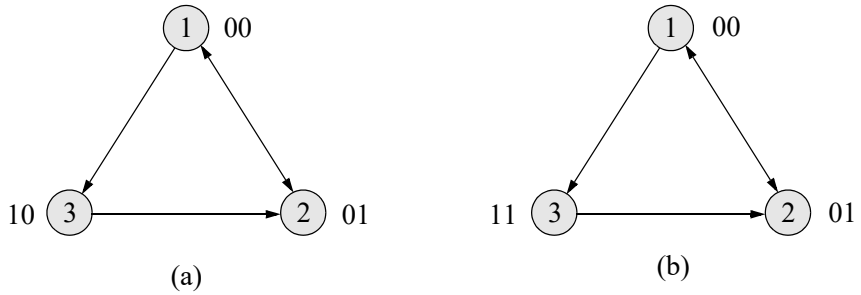


Figure 7.131 Transition diagrams illustrating state code assignments for a 3-row merged flow table containing noncritical and critical races: (a) a noncritical race between rows 2 and 3 and (b) a critical race between rows 1 and 3.

Figure 7.132 depicts the augmented merged flow table containing the original three rows plus a fourth row of unspecified entries. The transition diagram for the augmented flow table is shown in Figure 7.133. The transition from row 1 to row 3 is replaced by an equivalent sequence from row 1 to row 4 and then to row 3, as shown by the arrows in Figure 7.132 and Figure 7.133. This sequence represents a transition from stable state ⓐ or ⓔ through transient state c in column $x_1x_2 = 10$, then to the unspecified entry in row 4, then to state ⓒ in row 3. All other transitions involve a change of only one feedback variable.

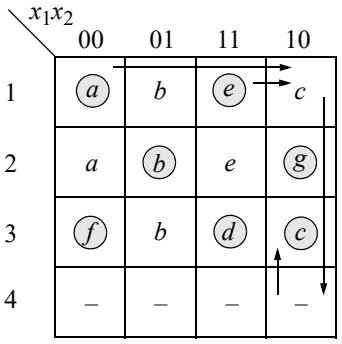


Figure 7.132 Augmented merged flow table for Example 7.15.

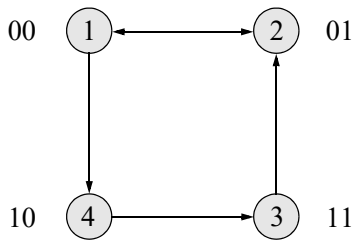


Figure 7.133 Transition diagram for the augmented merged flow table of Figure 7.132.

The codes next to each row vertex indicate the assigned values for the feedback variables. The choice of state codes is arbitrary in this context. Any assignment of sequential Gray code 2-tuples is a suitable choice. Each state name is now replaced by its corresponding assigned 2-tuple.

The excitation map for the augmented merged flow table is shown in Figure 7.134. This is a combined excitation map for excitation variables Y_{1e} and Y_{2e}. The feedback variables are $y_{1f}y_{2f}$. The stable states in each row are assigned excitation values that are equal to the feedback values of the corresponding row. The unstable states are assigned excitation values that direct the machine to the destination stable state.

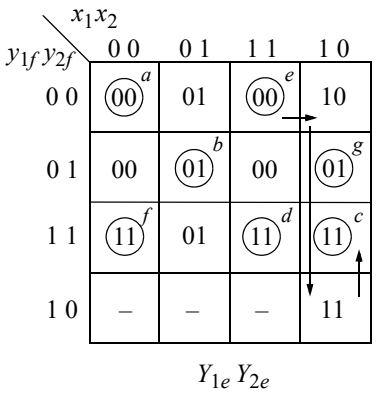


Figure 7.134 Combined excitation map for the augmented merged flow table of Figure 7.132.

For example, the transition from state ⓐ to state ⓑ is specified by ⓐ $\rightarrow b \rightarrow$ ⓑ. Thus, $Y_{1e}Y_{2e} =$ ⓪⓪ $\rightarrow 01 \rightarrow$ ⓪①. The entry $Y_{1e}Y_{2e} = 01$ for unstable state b in row $y_{1f}y_{2f} = 00$, column $x_1x_2 = 01$, specifies the excitation values which

will become the values of the feedback variables after a delay of Δt, directing the machine to state ⓑ in row $y_{1f}y_{2f} = 01$, column $x_1x_2 = 01$.

Likewise, the transition from state ⓖ to state ⓔ requires excitation values of $Y_{1e}Y_{2e} = 00$ to be entered in unstable state e in row $y_{1f}y_{2f} = 01$, column $x_1x_2 = 11$. Thus, after a delay of Δt, the feedback variables become equal to the excitation variables and the machine enters state ⓔ. The "don't care" entries in row 4 can be used as intermediate transient states to introduce cycles which direct the machine to the desired stable state. Code assignments must be avoided that would cause the machine to cycle continuously between unstable states.

The transition from state ⓔ to state ⓒ necessitates the values for $Y_{1e}Y_{2e}$ as shown in Figure 7.135. The sequence from state ⓔ to state ⓒ is graphically illustrated by the arrows in Figure 7.134. A similar path is realized for a transition from state ⓐ to state ⓒ. Thus, the addition of the fourth row in the excitation map resolves the adjacency problem and thus, the possible critical race condition of the original merged flow table of Figure 7.129. Other arrangements are possible for the four rows of the augmented merged flow table.

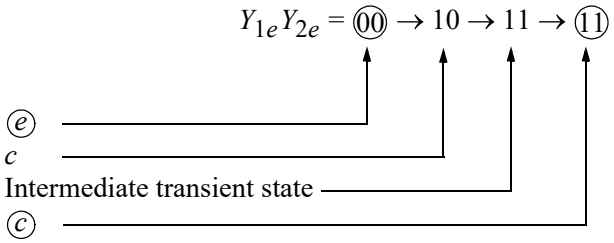


Figure 7.135 State transition sequence depicting the transition from stable state ⓔ to stable state ⓒ.

The combined excitation map of Figure 7.134 is now separated into its constituent parts to obtain individual excitation maps for Y_{1e} and Y_{2e}, as shown in Figure 7.136. To obtain the excitation map for Y_{1e}, simply transfer the values for Y_{1e} from the minterm locations in the combined map to the same squares in the individual map. Repeat the process to obtain the excitation map for Y_{2e}.

The equations for the excitation variables are derived directly from the excitation maps. The equations can be specified in either a sum-of-products form or a product-of-sums form. Regardless of the form used, the Boolean equations must be free of static-1 and static-0 hazards. The sum-of-products notation is shown in Equation 7.45 for Y_{1e} and Y_{2e}; the product-of-sums form is shown in Equation 7.46 for each excitation variable. All equations are free of static hazards. In some instances, the equations may be manipulated — using the laws of Boolean algebra — to obtain a network with fewer logic gates.

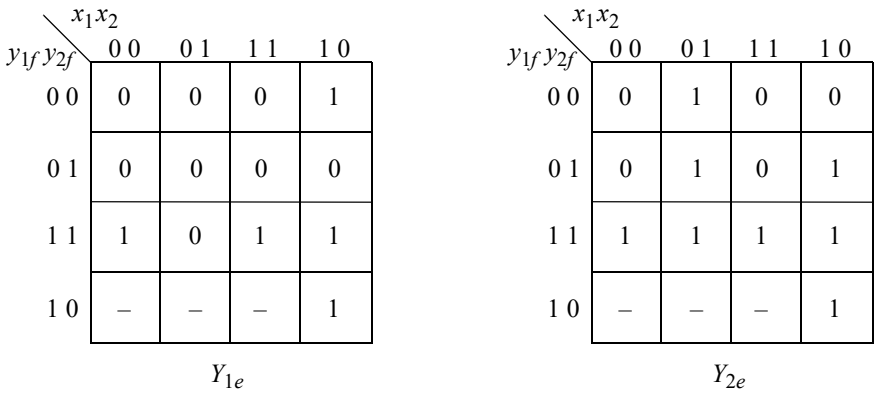


Figure 7.136 Individual excitation maps obtained from Figure 7.134 for the asynchronous sequential machine of Example 7.15.

$$Y_{1e} = y_{1f}x_1 + y_{1f}x_2' + y_{2f}'x_1x_2'$$
$$Y_{2e} = y_{1f} + x_1'x_2 + y_{2f}x_1x_2' \tag{7.45}$$

$$Y_{1e} = (x_1 + x_2')\,(y_{1f} + y_{2f}')\,(y_{1f} + x_1)\,(y_{1f} + x_2')$$
$$Y_{2e} = (y_{1f} + x_1 + x_2)\,(y_{1f} + x_1' + x_2')\,(y_{1f} + y_{2f} + x_1')\,(y_{1f} + y_{2f} + x_2) \tag{7.46}$$

Hazard cover

Example 7.16 A merged flow table is shown in Figure 7.137 in which multiple adjacencies are required. The state transitions can be determined with reference to individual rows in the merged flow table. This is accomplished by means of a transition diagram, which clearly identifies all race conditions. The four rows are listed as shown in the transition diagram of Figure 7.138.

Row 1 proceeds to row 2 by the following two transitions: $ⓐ \rightarrow f \rightarrow ⓕ$ and $ⓑ \rightarrow c \rightarrow ⓒ$. This is illustrated by the directed line connecting vertices 1 and 2. A transition occurs from row 2 to rows 3 and 4 by state transitions $ⓒ \rightarrow d \rightarrow ⓓ$ and $ⓕ \rightarrow g \rightarrow ⓖ$, respectively, as indicated by the lines connecting row 2 to row 3 and row 4.

			x_1x_2 00	01	11	10
1	(a)	(b)	(a)	f	c	(b)
2	(c)	(f)	g	(f)	(c)	d
3	(d)	(h)	a	(h)	e	(d)
4	(e)	(g)	(g)	h	(e)	b

Figure 7.137 Merged flow table for Example 7.16.

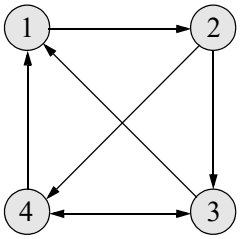


Figure 7.138 Transition diagram obtained from the merged flow table of Figure 7.137.

A transition is realized between rows 3 and 1 from state (h) when the input vector changes from $x_1x_2 = 01$ to 00 or from state (d) when the input vector changes from $x_1x_2 = 10$ to 00. Also, a transition can emanate from row 3 to row 4 from stable states (h) or (d) when the inputs change from $x_1x_2 = 01$ to 11 or from $x_1x_2 = 10$ to 11, respectively. Finally, a transition can occur from row 4 to row 3 by the sequence (g) $\rightarrow$ $h \rightarrow$ (h) and from row 4 to row 1 by the sequence (e) $\rightarrow b \rightarrow$ (b).

Since it is impossible to assign two-tuple state variable codes so that all four rows are adjacent, it is necessary to introduce an additional state variable. Adding a state variable produces an augmented merged flow table containing eight rows as shown in Figure 7.139. The top four rows are unchanged from the original table. The bottom four rows contain unspecified entries in every column. The lower rows will be used to insert intermediate transient states, where applicable. An intermediate state will be inserted in the cycle whenever a state transition sequence causes two or more excitation variables to change state simultaneously. The values for the feedback variables will be obtained from a transition diagram.

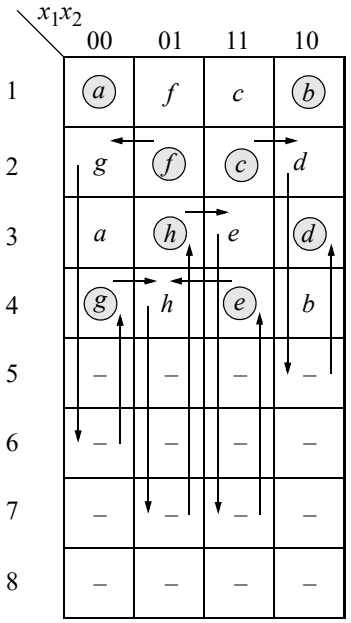


Figure 7.139 Augmented merged flow table for the asynchronous sequential machine of Example 7.16.

The transition diagram for eight rows is shown in Figure 7.140. This is a cube in which the top surface is assigned the 3-tuple Gray code 000, 001, 011, 010 for row vertices 1, 2, 6, 4, respectively. The bottom surface is assigned Gray code values of 100, 101, 111, 110 for row vertices 3, 5, 8, 7, respectively.

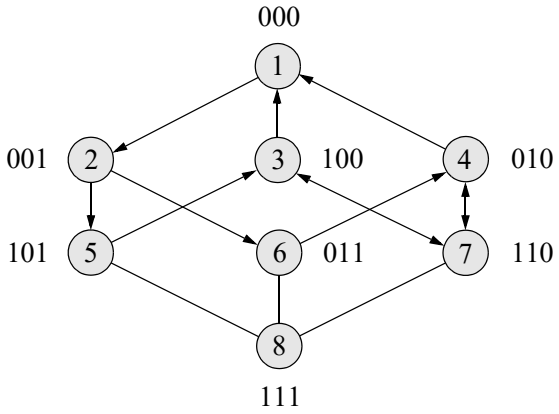


Figure 7.140 Transition diagram for the augmented merged flow table of Figure 7.139.

Any state transition that results in two or more state variables changing values simultaneously can be rerouted through unspecified entries in rows 5 through 8, such that each individual transition in the cycle takes place between adjacent state variable codes. For example, the transition $\textcircled{f} \rightarrow g \rightarrow \textcircled{g}$ from row 2 to row 4 can be directed first to row 6 and then to row 4. Each step of the cycle proceeds through contiguous vertices that involve a change of only one state variable. The transitions in the merged flow table of Figure 7.137 in which race conditions exist are shown redirected in the augmented merged flow table of Figure 7.139.

The transition from state $\textcircled{f}$ (row 2) to state $\textcircled{g}$ (row 4) passes through an intermediate state in row 6, as shown by the arrows. The transition from state $\textcircled{c}$ (row 2) to state $\textcircled{d}$ (row 3) causes two state variables to change (from 001 to 100), as shown in the transition diagram. To avoid this race condition, the transition from row 2 is directed first to row 5 and then to row 3. This modified state transition is

$$2\ (001) \rightarrow 5\ (101) \rightarrow 3\ (100); \qquad \textcircled{c} \rightarrow d \rightarrow \textcircled{d}$$

In a similar manner, the remaining transitions are modified in which race conditions are present. The arrows in Figure 7.139 illustrate the redirected transitions, in which each contiguous vertex (row) in the cycle involves a change of only one state variable. The modified row transitions are listed in Figure 7.141 together with their corresponding state transitions.

$$\begin{aligned}
&2\ (001) \rightarrow 6\ (011) \rightarrow 4\ (010); && \textcircled{f} \rightarrow g \rightarrow \textcircled{g}\\
&2\ (001) \rightarrow 5\ (101) \rightarrow 3\ (100); && \textcircled{c} \rightarrow d \rightarrow \textcircled{d}\\
&3\ (100) \rightarrow 7\ (110) \rightarrow 4\ (010); && \textcircled{h} \rightarrow e \rightarrow \textcircled{e}\\
&3\ (100) \rightarrow 7\ (110) \rightarrow 4\ (010); && \textcircled{d} \rightarrow e \rightarrow \textcircled{e}\\
&4\ (010) \rightarrow 7\ (110) \rightarrow 3\ (100); && \textcircled{g} \rightarrow h \rightarrow \textcircled{h}\\
&4\ (010) \rightarrow 7\ (110) \rightarrow 3\ (100); && \textcircled{e} \rightarrow h \rightarrow \textcircled{h}
\end{aligned}$$

7.5 Analysis of Pulse-Mode Asynchronous Sequential Machines

Many situations are encountered in digital engineering where the input signals occur as pulses and in which there is no periodic clock signal to synchronize the operation of the sequential machine. Typical examples which use the principles of *pulse-mode* techniques are vending machines, demand-access road intersections, and automatic toll booths.

In the presentation of synchronous sequential machines, the data input signals were asserted as voltage levels. A periodic clock input was also required, such that state changes occurred on the active clock transition. In the synthesis of asynchronous sequential machines, the input variables were also considered as voltage levels; however, there was no machine clock to synchronize state changes. The operation of

pulse-mode machines is similar, in some respects, to both synchronous and asynchronous sequential machines. State changes occur on the application of input pulses which trigger the storage elements, rather than on a clock signal. The input pulses, however, occur randomly in an asynchronous manner.

In pulse-mode sequential machines, each variable of the input alphabet X is active in the form of a pulse. The duration of the pulse is less than the propagation delay of the storage elements and associated logic gates. Thus, an input pulse will initiate a state change, but the completion of the change will not take place until after the corresponding input has been deasserted. Multiple inputs cannot be active simultaneously. There is no separate clock input in pulse-mode machines.

The storage elements in pulse-mode machines are usually level-sensitive rather than edge-triggered devices. Thus, *Set/Reset* (*SR*) latches using NAND or NOR logic are typically used in the implementation of pulse-mode machines. In order for the operation of the machine to be deterministic, some restrictions apply to the input pulses:

1. Input pulses must be of sufficient duration to trigger the storage elements.

2. The time duration of the pulses must be shorter than the minimal propagation delay through the combinational input logic and the storage elements, so that the pulses are deasserted before the storage elements can again change state.

3. The time duration between successive input pulses must be sufficient to allow the machine to stabilize before application of the next pulse.

4. Only one input pulse can be active at a time.

If the input pulse is of insufficient duration, then the storage elements may not be triggered and the machine will not sequence to the next state. If the pulse duration is too long, then the pulse will still be active when the machine changes from the present state $Y_{j(t)}$ to the next state $Y_{k(t+1)}$. The storage elements may then be triggered again and sequence the machine to an incorrect next state. If the time between consecutive pulses is too short, then the machine will be triggered while in an unstable condition, resulting in unpredictable behavior.

Since pulse inputs cannot occur simultaneously, a pulse-mode machine with n input signals can have only $n+1$ combinations of the input alphabet, instead of 2^n combinations as in synchronous sequential machines and asynchronous sequential machines that are not inherently characterized by pulse-mode operation. For example, for a pulse-mode machine with two inputs x_1 and x_2, three possible valid combinations can occur: $x_1x_2 = 00$, 10, or 01. However, since no changes are initiated by an input vector of $x_1x_2 = 00$, it is necessary to consider only the vectors $x_1x_2 = 10$ and 01 when analyzing or designing a pulse-mode asynchronous sequential machine.

Similarly, for a machine with three inputs x_1, x_2, and x_3, only the following input vectors need be considered: $x_1x_2x_3 = 100$, 010, and 001. The absence of a clock signal implies that state transitions occur only when an input is asserted.

7.5.1 Analysis Procedure

Pulse-mode machines respond immediately to the assertion of an input signal without waiting for a clock signal. Analysis of pulse-mode sequential machines — as for synchronous sequential machines — consists of deriving a next-state table, input maps and equations, output maps and equations, and a state diagram for a given logic diagram. A timing diagram may also prove useful in analysis.

***SR* latches as storage elements** The analysis proceeds in a manner analogous to that described for synchronous sequential machines. The predominant differences are the absence of a clock signal and the input restrictions mentioned previously. A Moore pulse-mode asynchronous sequential machine is shown in Figure 7.141, and will be analyzed with respect to a next-state table together with input and output maps.

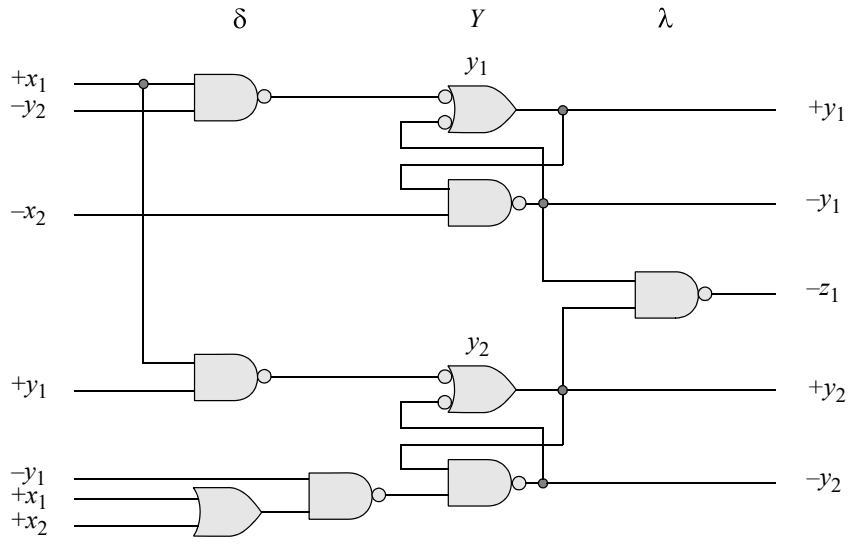


Figure 7.141 Pulse-mode Moore machine using latches for storage elements.

Assume that the machine is reset initially such that, $y_1y_2 = 00$. Inputs x_1 and x_2 are assigned values of $x_1x_2 = 10$ and 01 for each specified state. The equations to set and reset y_1 and y_2 are derived from the logic diagram and are shown in Equation 7.47.

If x_1 is pulsed when the machine is in an initial reset condition, then latch y_1 will set. Input pulse x_1 must be deasserted, however, before latch y_1 stabilizes in a set condition, otherwise the conditions to set latch y_2 would be established. If y_2 sets before x_1 is deasserted, then an erroneous next state will be generated because the input pulse must be deasserted before the state change is received at the input logic. Assuming

that the input restrictions are met, latch y_2 will remain in a reset state. Thus, from an initial condition of $y_1y_2 = 00$, a pulse on input x_1 causes the machine to proceed to state $y_1y_2 = 10$, as shown in the first row of Table 7.21. Output z_1 remains inactive.

If input x_2 is pulsed in state $y_1y_2 = 00$, then both latches will receive a reset pulse coincident with the assertion of x_2; the pulse will be active for the duration of x_2. Therefore, latches y_1 and y_2 will remain reset, as shown in the second row of Table 7.21. Output z_1 remains inactive.

$$
\begin{aligned}
Sy_1 &= y_2' x_1 \\
Ry_1 &= x_2 \\
\\
Sy_2 &= y_1 x_1 \\
Ry_2 &= y_1'(x_1 + x_2)
\end{aligned}
\tag{7.47}
$$

Table 7.21 Next-State Table for the Pulse-Mode Moore Machine of Figure 7.141

	Present State	Inputs	Next State	Output
State Name	y_1 y_2	x_1 x_2	y_1 y_2	z_1
a	0 0	1 0	1 0	0
	0 0	0 1	0 0	0
b	1 0	1 0	1 1	0
	1 0	0 1	0 0	0
c	1 1	1 0	1 1	0
	1 1	0 1	0 1	0
d	0 1	1 0	0 0	1
	0 1	0 1	0 0	1

Assume that the machine is now in state $y_1y_2 = 10$ and that the input vectors $x_1x_2 = 10$ and 01 are applied in sequence. Using either Equation 7.47 or the logic diagram, it is evident that a pulse on x_1 will cause y_1 to remain set and y_2 to be set. If x_2 is pulsed, then y_1 will be reset and y_2 will remain reset.

Let the present state be $y_1y_2 = 11$ and the input vectors $x_1x_2 = 10$ and 01 be applied in sequence. When x_1 is pulsed, no change occurs to latch y_1 because

$Sy_1 = y_2'x_1 = 01 = 0$. Thus, y_1 remains set. Latch y_2 will also remain set because $Sy_2 = y_1x_1 = 11 = 1$. When x_2 is asserted, y_1 will be reset and y_2 will remain unchanged at $y_2 = 1$ because $Ry_2 = y_1'(x_1 + x_2) = 0(0 + 1) = 0$.

Finally, when the machine is in state $y_1y_2 = 01$, output z_1 is asserted unconditionally, due to the Moore characteristics of the output. If input x_1 is pulsed, then no change occurs to latch y_1 because $Sy_1 = y_2'x_1 = 01 = 0$. There is also no set pulse to latch y_2 because $Sy_2 = y_1x_1 = 01 = 0$. However, a reset pulse is generated for y_2 when x_1 is asserted, since $Ry_2 = y_1'(x_1 + x_2) = 1(1 + 0) = 1$. A pulse on x_2 provides a reset pulse to y_1; thus, y_1 remains reset. Latch y_2 is also reset by an x_2 pulse because $Ry_2 = y_1'(x_1 + x_2) = 1(0 + 1) = 1$. Therefore, the assertion of x_1 or x_2 in state $y_1y_2 = 01$ returns the machine to the initial state of $y_1y_2 = 00$. Table 7.21 lists all possible states of two storage elements with the associated input vectors and the corresponding next states. Column z_1 lists the output values for the present state.

The input maps contain the same information as the next-state table, but in a different format. The maps for pulse-mode machines are slightly different than those for synchronous sequential machines because the input pulses are exclusive. Since the inputs cannot be asserted simultaneously, each latch requires n input maps, one map for each input $x_1, x_2, \cdots, x_n$. In this example, each latch requires two input maps, one each for x_1 and x_2, as shown in Figure 7.142. The maps for each latch are in the same row. Each column of maps corresponds to a separate input. The map entries are defined as follows:

S indicates that the latch will be set.
s indicates that the latch will remain set.
R indicates that the latch will be reset.
r indicates that the latch will remain reset.

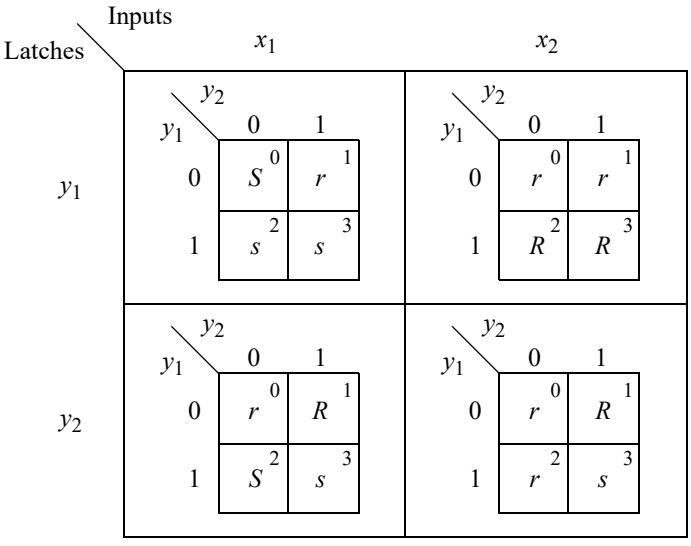


Figure 7.142 Input maps for the pulse-mode sequential machine of Figure 7.141.

In the presentation which follows, the set and reset equations of Equation 7.47 in conjunction with the next-state table of Table 7.21 will be utilized in deriving the entries for the input maps. The four maps will be constructed in parallel by considering each stable state in turn for each input vector.

Consider the input map in row y_1, column x_1 for an initial condition of $y_1y_2 = 00$ when x_1 is pulsed. The next state for latch y_1 will be 1 for an input vector of $x_1x_2 = 10$. That is, y_1 will change from $y_1 = 0$ to 1, which represents a set condition. Thus, the letter *S* is inserted in minterm location 0 of the map in row y_1, column x_1. The map in row y_2, column x_1 specifies the next states for latch y_2 when x_1 is pulsed. In state $y_1y_2 = 00$, the set equation for latch y_2 is $Sy_2 = y_1x_1 = 01 = 0$. Thus, there will be no change to the state of y_2, which remains reset, as indicated by the letter *r*.

Consider the map for y_1 in row y_1, column x_2 for an initial condition of $y_1y_2 = 00$. The assertion of x_2 provides a reset pulse to latch y_1. Since y_1 remains in a reset state, the letter *r* is inserted in minterm location 0. Similarly, the letter *r* is entered in minterm location 0 of the map for y_2 in row y_2, column x_2, because $Ry_2 = y_1'(x_1 + x_2) = 1(0 + 1) = 1$.

Consider row y_1, column x_1, for a present state of $y_1y_2 = 01$. When x_1 is asserted, there will be no change to latch y_1, because $Sy_1 = y_2'x_1 = 01 = 0$; that is, x_1 will not produce a set pulse to y_1. Thus, y_1 remains in a reset state, as indicated by the letter *r*. There is also no set pulse for y_2 when x_1 is asserted, because $Sy_2 = y_1x_1 = 01 = 0$. Latch y_2, however, receives a reset pulse when x_1 is asserted because $Ry_2 = y_1'(x_1 + x_2) = 1(1 + 0) = 1$. Since y_2 changes from $y_2 = 1$ to 0; therefore, the letter *R* is entered in minterm location 1.

For a present state of $y_1y_2 = 01$ in column x_2, the letters *r* and *R* are inserted in minterm location 1 for y_1 and y_2, respectively. Input x_2 provides a reset pulse to latch y_1, which remains reset. Input x_2 also provides a reset pulse for y_2 as specified by $Ry_2 = y_1'(x_1 + x_2) = 1(0 + 1) = 1$. Since the state of y_2 changes from 1 to 0, the letter *R* is entered in minterm location 1 of row y_2, column x_2.

In a similar manner, the remaining entries are derived. In column x_1, a pulse on input x_1 will provide either a set or reset pulse or leave the latches unchanged. In column x_2, a pulse on input x_2 will either reset the latches or leave them unchanged. Comparison of the entries in Table 7.21 with the entries in the input maps show a one-to-one correspondence for each state for identical input vectors.

When deriving the equations from the input maps, only the uppercase letters need be considered. The lowercase letters are treated as "don't care" entries and are used only if they aid in minimizing the equation. The equations derived from the input maps are identical to those shown in Equation 7.47. For example, consider the map in row y_1, column x_1. Minterms 0 and 2 can be combined, since both represent a set condition. Minterms 0 and 2 have common variables y_2' and x_1. Therefore, $Sy_1 = y_2'x_1$. The map in row y_1, column x_2, contains a reset entry in every location, indicating that x_2 will either reset latch y_1 or leave the latch in a reset state. Therefore, $Ry_1 = x_2$.

The map in row y_2, column x_1 contains the letters *S* and *s* in minterm locations 2 and 3, respectively, where *s* is considered a "don't care" entry. Since both squares possess the common variables y_1 and x_1, the set condition for latch y_2 is $Sy_2 = y_1x_1$. The map also contains entries of *r* and *R* in minterm locations 0 and 1, respectively, where *r* is considered a "don't care" entry. The same values exist in the same locations

for the map in row y_2, column x_2. Therefore, the reset equation for latch y_2 is $Ry_2 = y_1'x_1 + y_1'x_2 = y_1'(x_1 + x_2)$.

Sequential machines that operate in pulse mode will not have race conditions — either noncritical or critical — because only one input is active at a time and the machine is stable even in the absence of input pulses. Each product term in the set and reset equations contains at least one input variable. Also, no input variable appears in a complemented form because the complement of a variable corresponds to an inactive input signal.

***SR* latches with *D* flip-flops as storage elements** A pulse-mode machine is shown in the logic diagram of Figure 7.143. The machine has two inputs x_1 and x_2 and one output z_1. The storage elements consist of *SR* latches and *D* flip-flops. The output of each latch connects to the *D* input of the associated flip-flop forming a master-slave relationship. Since the *D* flip-flops are clocked on the trailing edge of the positive input pulses, state changes are not fed back to the δ next-state logic until the active input has been deasserted. Clocking the flip-flops on the negative edge of the positive input pulses delays the next state from affecting the input logic while an input pulse is still active. Thus, the machine operates in a deterministic manner.

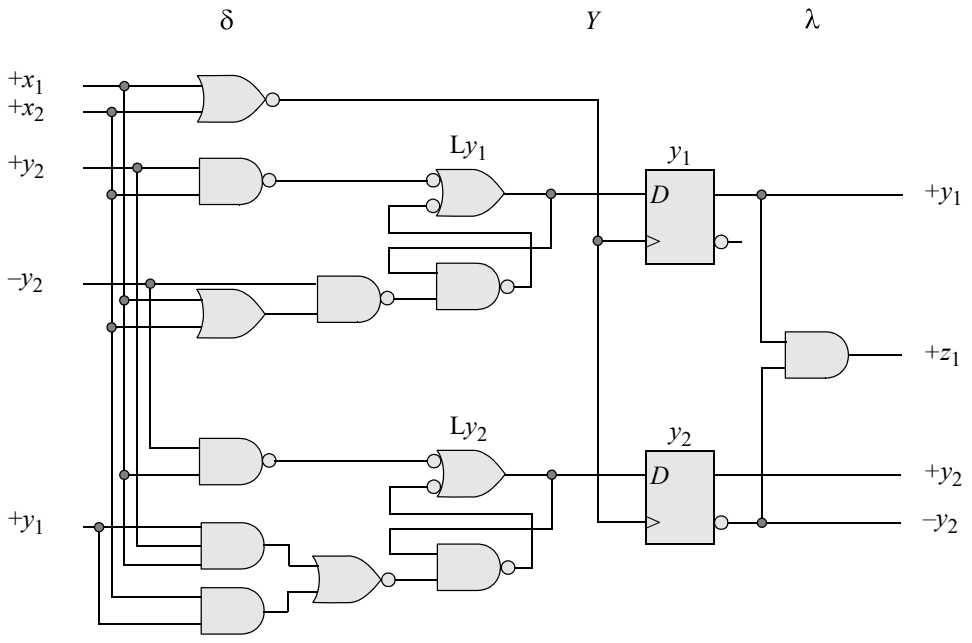


Figure 7.143 Logic diagram for a pulse-mode machine using latches and *D* flip-flops in a master-slave relationship.

The input equations are obtained directly from the logic diagram and are shown in Equation 7.48. Latch Ly_1 will be set if flip-flop y_2 is set and x_2 is pulsed. Latch Ly_1

will be reset if y_2 is reset and either x_1 or x_2 is pulsed. Thus, the set and reset conditions for latch Ly_1 are $SLy_1 = y_2x_2$ and $RLy_1 = y_2'(x_1 + x_2)$, respectively.

$$SLy_1 = y_2x_2$$
$$RLy_1 = y_2'x_1 + y_2'x_2 = y_2'(x_1 + x_2)$$

$$SLy_2 = y_2'x_1$$
$$RLy_2 = y_1y_2x_1 + y_1x_2 \tag{7.48}$$

Similarly, the set and reset conditions for latch Ly_2 are obtained. Latch Ly_2 will be set if flip-flop y_2 is reset and x_1 is pulsed. Latch Ly_2 will be reset if flip-flops y_1 and y_2 are both set and x_1 is pulsed, or if flip-flop y_1 is set and x_2 is pulsed. These conditions yield the set and reset equations $SLy_2 = y_2'x_1$ and $RLy_2 = y_1y_2x_1 + y_1x_2$, respectively.

The input equations for Dy_1 and Dy_2 are not required, since the state of each latch is transferred to its associated flip-flop on the negative transition of the active input variable.

Using the equations of Equation 7.48, the next-state table can now be constructed. If the machine is in the initial reset state of $y_1y_2 = 00$ and input x_1 is pulsed, then latch Ly_1 remains reset and latch Ly_2 is set, as shown in the first row of the next-state table of Table 7.22. If x_2 is pulsed in the initial reset condition, then both latches remain reset. Output z_1 remains inactive.

Table 7.22 Next-State Table for the Pulse-Mode Moore Machine of Figure 7.143

	Present State	Inputs	Next State	Output
State Name	y_1 y_2	x_1 x_2	y_1 y_2	z_1
a	0 0	1 0	0 1	0
	0 0	0 1	0 0	0
b	0 1	1 0	0 1	0
	0 1	0 1	1 1	0
c	1 1	1 0	1 0	0
	1 1	0 1	1 0	0
d	1 0	1 0	0 1	1
	1 0	0 1	0 0	1

In state $y_1y_2 = 01$, if x_1 is asserted, then latches Ly_1 and Ly_2 will remain reset and set, respectively, and the machine will not change state. If x_2 is asserted, however, then latch Ly_1 will set and latch Ly_2 will remain set, because $SLy_1 = y_2x_2 = 11 = 1$ and $Ry_2 = y_1y_2x_1 + y_1x_2 = 010 + 01 = 0$, resulting in a transition from state $y_1y_2 = 01$ to state $y_1y_2 = 11$. Output z_1 remains inactive.

In state $y_1y_2 = 11$, if x_1 is pulsed, then latch Ly_1 remains set while latch Ly_2 resets. The same set and reset conditions apply if x_2 is pulsed. Thus, a state transition will occur from $y_1y_2 = 11$ to 10 if either x_1 or x_2 is pulsed.

Finally, in state $y_1y_2 = 10$, output z_1 is asserted unconditionally, due to the Moore characteristics of the output variable. If x_1 is pulsed in state $y_1y_2 = 10$, then latch Ly_1 is reset, whereas latch Ly_2 is set and the machine proceeds to state $y_1y_2 = 01$. If x_2 is pulsed, then latch Ly_1 is reset and latch Ly_2 remains reset, causing a transition to the initial state of $y_1y_2 = 00$.

The input maps can be derived directly from the next-state table or from the input equations. The input maps are shown in Figure 7.144, where the letters *S* and *s* indicate that the latch will be set or remain set, respectively, and the letters *R* and *r* indicate that the latch will be reset or remain reset, respectively.

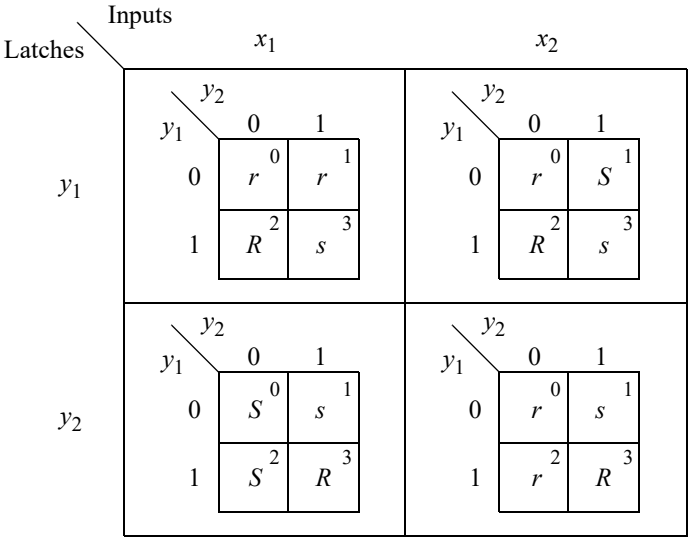


Figure 7.144 Input maps for the pulse-mode sequential machine of Figure 7.143.

The map entries correlate directly to the entries in the next-state table. For example, in state $y_1y_2 = 10$, latch Ly_1 will be reset if x_1 is pulsed, as indicated by the letter *R* in minterm location 2 of the map in row y_1, column x_1. Also, in state $y_1y_2 = 10$, latch Ly_2 will be set if x_1 is pulsed, as indicated by the letter *S* in minterm location 2 of the map in row y_2, column x_1.

Now consider the effect when input x_2 is asserted in state $y_1y_2 = 10$. When x_2 is activated, a reset pulse is applied to latch Ly_1, as shown in the logic diagram and the next-state table. Since latch Ly_1 was set, the letter *R* is entered in minterm location 2 of the map for latch Ly_1 in row y_1, column x_2. The assertion of x_2 will also generate a reset pulse to latch Ly_2. However, since latch Ly_2 is already reset, an x_2 pulse causes latch Ly_2 to remain in a reset state, as specified by the letter *r* in minterm location 2 of the map for latch Ly_2 in row y_2, column x_2.

7.6 Synthesis of Pulse-Mode Asynchronous Sequential Machines

Due to the stringent requirements of input pulse characteristics, pulse-mode machines are less preferred than synchronous sequential machines, especially when the machine is implemented with subnanosecond logic. The synthesis concepts, however, are of sufficient importance to warrant a detailed presentation of the synthesis procedure.

Reliability of pulse-mode machines can be increased by inserting delay circuits of an appropriate duration in the output networks of the storage elements. The aggregate delay of the storage elements and the delay circuit must be of sufficient duration so that the input pulse will be deasserted before the storage element output signals arrive at the δ next-state logic.

7.6.1 Synthesis Procedure

Three techniques are commonly used to insert delays in the storage element outputs: An even number of inverters are connected in series with each latch output; a linear delay circuit is connected in series with each latch output; or an edge-triggered *D* flip-flop is connected in series with each latch output. The flip-flops are set to the state of the latches, but are triggered on the trailing edge of the input pulses. Thus, the flip-flop outputs — and therefore the state of the machine as represented by the *SR* latch outputs — are received at the δ next-state logic only when the active input pulse has been deasserted. The *SR* latches and the *D* flip-flops constitute a master-slave relationship and will be the primary means to implement pulse-mode asynchronous sequential machines in this section.

***SR* latches with *D* flip-flops as storage elements** The pulse width restrictions that are dominant in pulse-mode sequential machines can be eliminated by including *D* flip-flops in the feedback path from the *SR* latches to the δ next-state logic. Providing edge-triggered *D* flip-flops as a constituent part of the implementation negates the requirement of precisely-controlled input pulse durations. This is by far the most reliable means of synthesizing pulse-mode machines. The *SR* latches, in conjunction with the *D* flip-flops, form a master-slave configuration as shown in Figure 7.145.

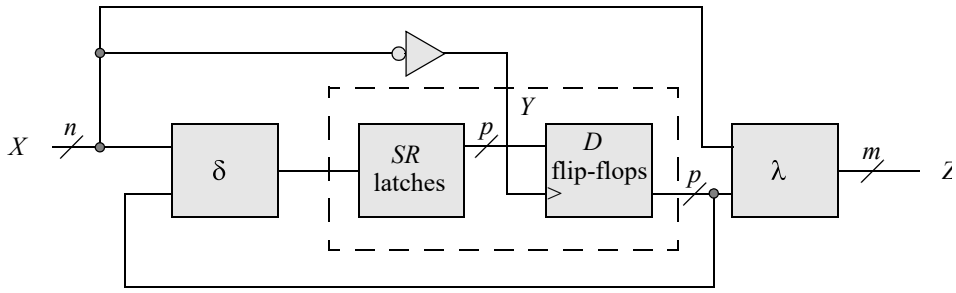


Figure 7.145 General block diagram of a pulse-mode asynchronous sequential machine using *SR* latches and *D* flip-flops in a master-slave configuration.

The output of each latch connects to the *D* input of its associated flip-flop which in turn connects to the δ next-state logic. The flip-flops are clocked on the complemented trailing edge of the active input variable. For example, if the input pulses are active high, then the *D* flip-flops are triggered on the inverted negative transition of the active input pulse; that is, the flip-flops are triggered on a positive transition, as shown in Figure 7.145. *JK* flip-flops may also be used as the slave storage elements. The output alphabet *Z* of pulse-mode machines can be generated as either levels for Moore machines or as pulses for Mealy machines.

Example 7.17 A Mealy pulse-mode asynchronous sequential machine will be synthesized which has two pulse input variables x_1 and x_2 and one output z_1. Output z_1 is asserted coincident with every second x_2 pulse, if and only if the pair of x_2 pulses is immediately preceded by an x_1 pulse. The machine will be implemented using NOR logic and inverters for the logic primitives. The storage elements will consist of *SR* latches and positive-edge-triggered *D* flip-flops in a master-slave configuration. A representative timing diagram is shown in Figure 7.146.

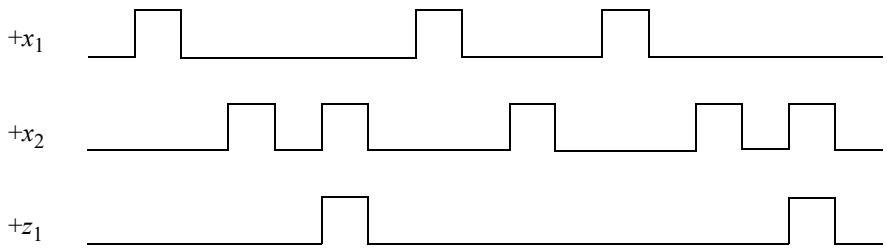


Figure 7.146 Representative timing diagram for the Mealy pulse-mode asynchronous sequential machine of Example 7.17.

The state diagram for the Mealy machine is shown in Figure 7.147; notice that only one input is asserted for any state transition. The input maps are shown in Figure 7.148. There is one unused state $y_1y_2 = 11$. Beginning in state ⓐ ($y_1y_2 = 00$) of the state diagram, if x_1 is asserted, then flip-flop y_1 remains reset; therefore, an entry of r is inserted in minterm location 0 of the input map for flip-flop y_1 in column x_1.

In state ⓒ ($y_1y_2 = 10$), if x_1 is asserted, then flip-flop y_1 is reset; therefore, an entry of R is inserted in minterm location 2 of the map for flip-flop y_1 in column x_1. In state ⓑ ($y_1y_2 = 01$), if x_2 is asserted, then flip-flop y_2 is reset; therefore, an entry of R is inserted in minterm location 1 of the map for flip-flop y_2 in column x_2. The remaining entries are inserted in a similar manner.

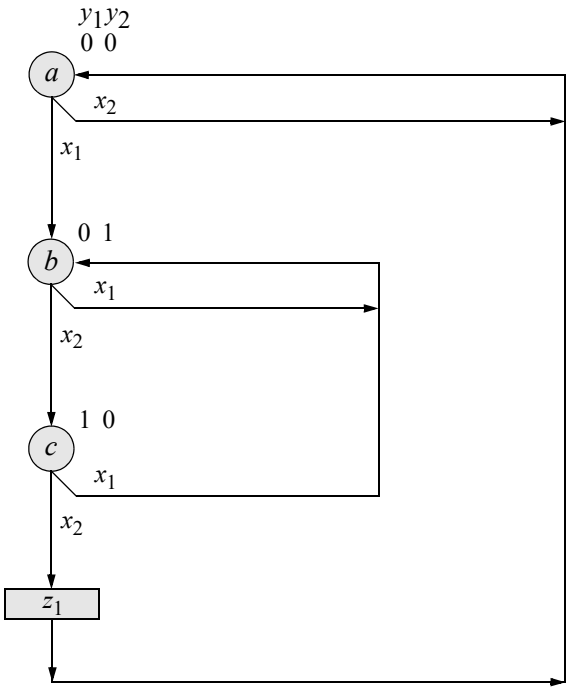


Figure 7.147 State diagram for the Mealy pulse-mode asynchronous sequential machine of Example 7.17.

The input equations are shown in Equation 7.49. The logic diagram is shown in Figure 7.149 using NOR logic for the logic primitives and the latches. The output from the latches connects directly to the D inputs of the clocked storage elements which are positive-edge-triggered D flip-flops. Output z_1 is asserted in state ⓒ if $x_2 = 1$. The output map for z_1 is shown in Figure 7.150 and the output equation is shown in Equation 7.50.

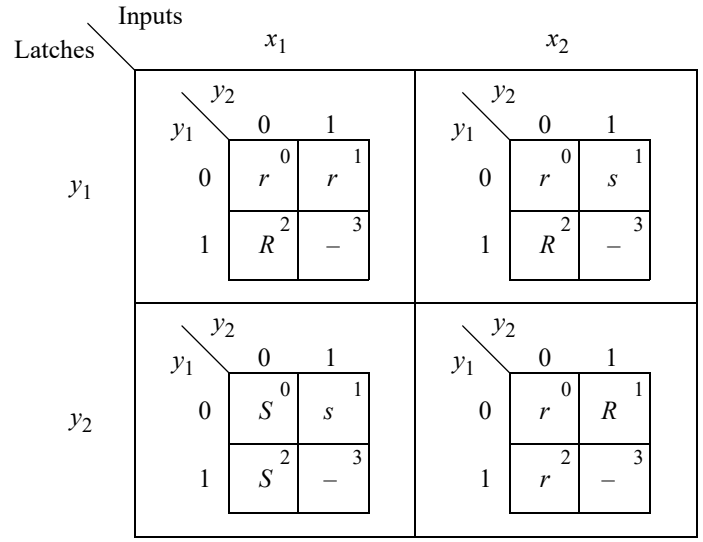


Figure 7.148 Input maps for the Mealy pulse-mode asynchronous sequential machine of Example 7.17.

$$SLy_1 = y_2x_2 \qquad RLy_1 = x_1 + y_1x_2 \qquad SLy_2 = x_1 \qquad RLy_2 = x_2 \tag{7.49}$$

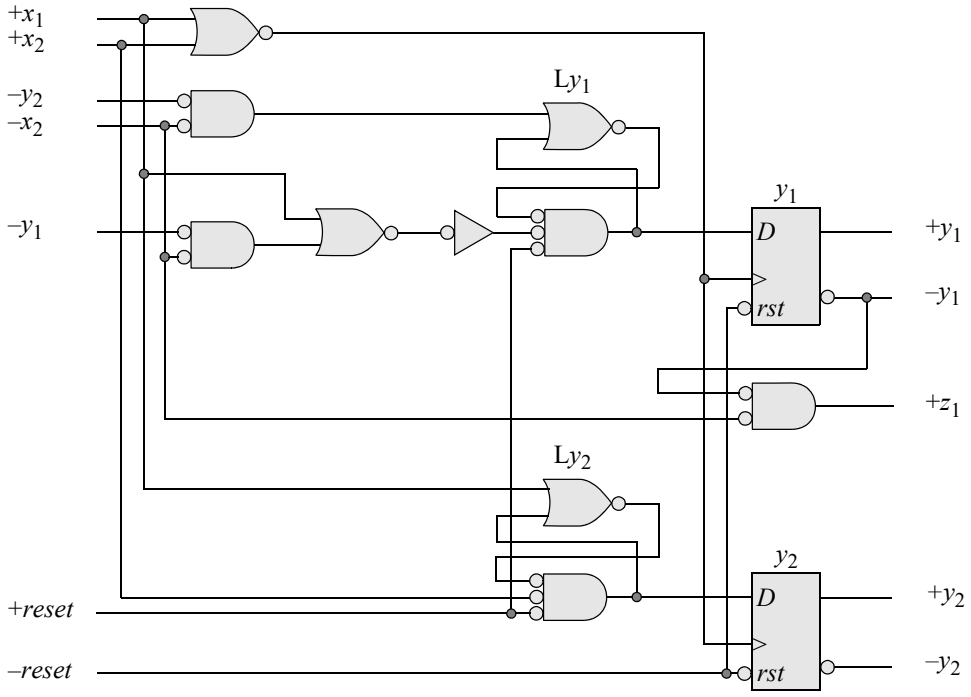


Figure 7.149 Logic diagram for the Mealy machine of Example 7.17.

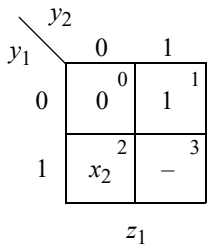


Figure 7.150 Output map for z_1 for the Mealy machine of Example 7.17.

$$z_1 = y_1 x_2 \tag{7.50}$$

Example 7.18 A Moore pulse-mode asynchronous sequential machine will be designed that has two inputs x_1 and x_2 and one output z_1. The negative transition of every second consecutive x_1 pulse will assert output z_1 as a level. The output will remain set for all following contiguous x_1 pulses. The output will be deasserted at the negative transition at the second of two consecutive x_2 pulses. The machine will be implemented with NAND logic for the δ next-state logic. The storage elements will be NAND *SR* latches and positive-edge-triggered *D* flip-flops in a master-slave configuration.

A representative timing diagram is shown in Figure 7.151 and the state diagram is shown in Figure 7.152 depicting all possible state transition sequences that conform to the functional specifications. The input maps are shown in Figure 7.153 using *S* (set), *s* (remains set), *R* (reset), and *r* (remains reset) entries. The corresponding set and reset equations for the *SR* latches are listed in Equation 7.51.

The output map for z_1 is shown in Figure 7.154 and the output equation appears in Equation 7.52. The logic diagram is shown in Figure 7.155 using only NAND logic — no inverters — for the δ next-state logic and *SR* latches together with positive-edge-triggered *D* flip-flops.

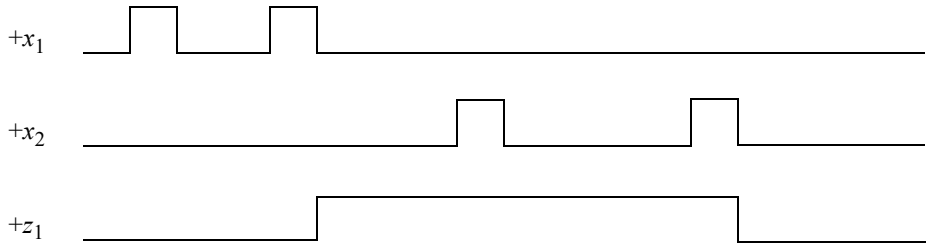


Figure 7.151 Representative timing diagram for the Moore pulse-mode asynchronous sequential machine of Example 7.18.

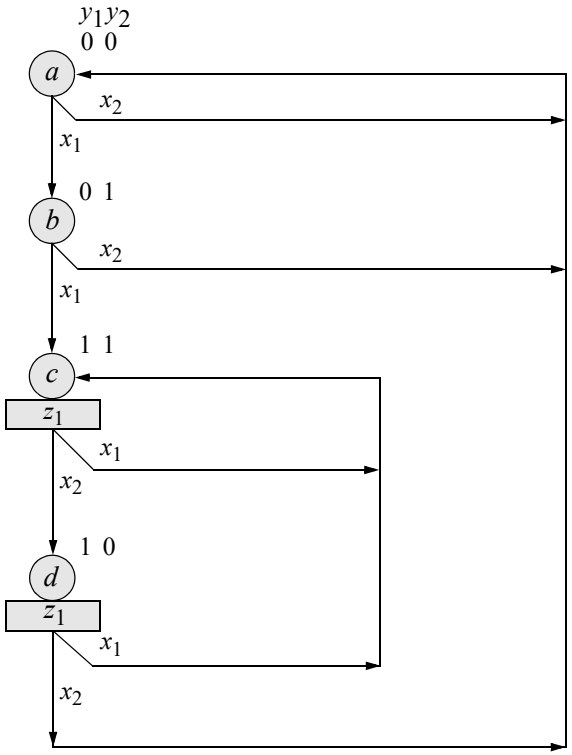


Figure 7.152 State diagram for the Moore pulse-mode asynchronous sequential machine of Example 7.18.

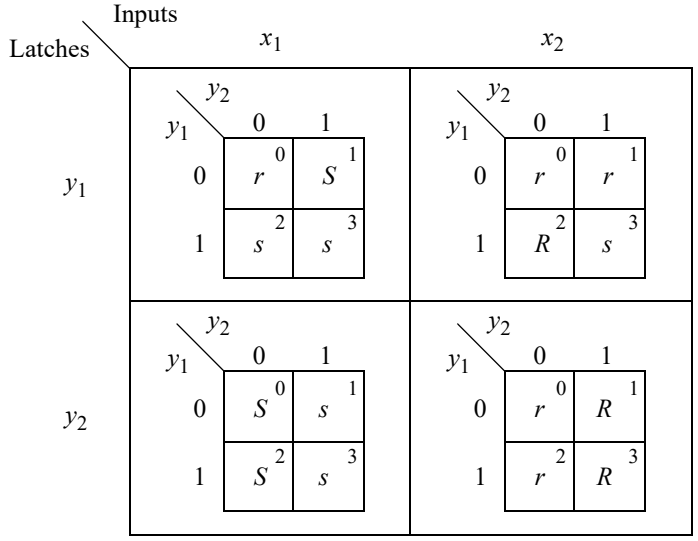


Figure 7.153 Input maps for the Moore pulse-mode asynchronous sequential machine of Example 7.18.

$$SLy_1 = y_2x_1 \qquad RLy_1 = y_2'x_2$$
$$SLy_2 = x_1 \qquad RLy_2 = x_2 \tag{7.51}$$

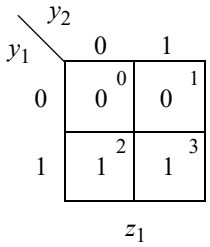


Figure 7.154 Output map for z_1 for the Moore pulse-mode asynchronous sequential machine of Example 7.18.

$$z_1 = y_1 \tag{7.52}$$

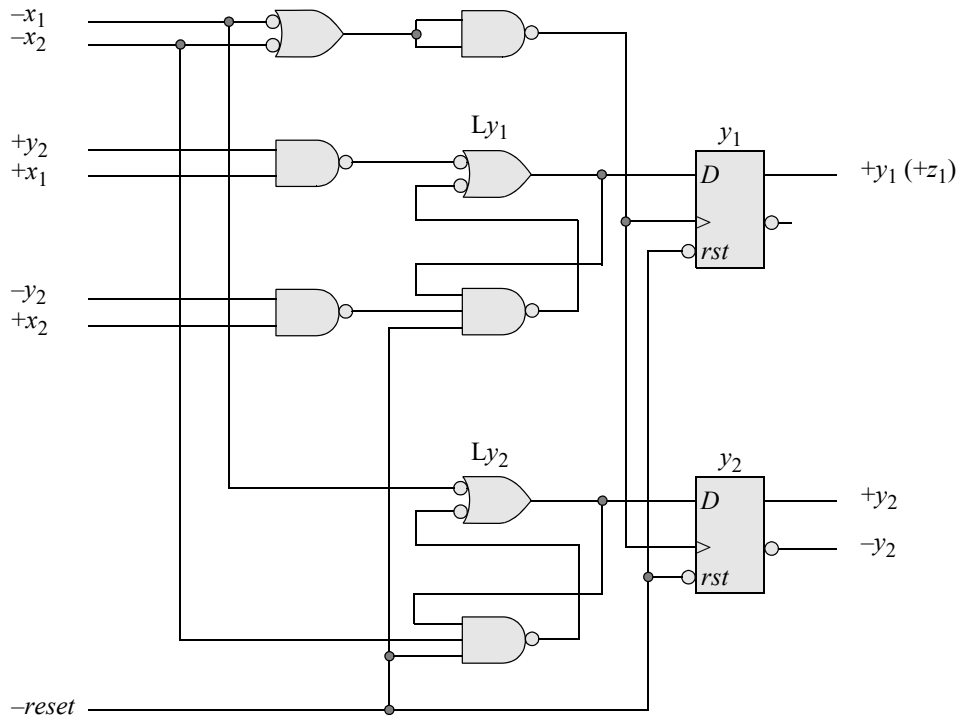


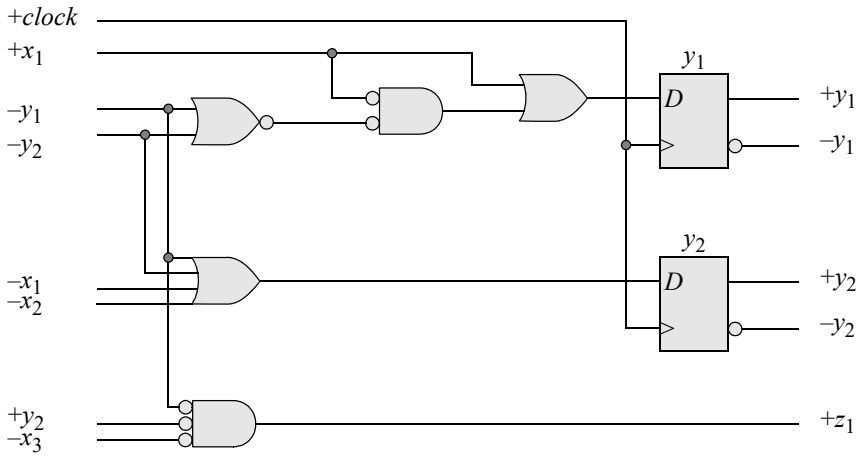
Figure 7.155 Logic diagram for the Moore pulse-mode asynchronous sequential machine of Example 7.18.

The logic diagram is implemented directly from Equation 7.51 and Equation 7.52. The state flip-flops y_1 and y_2 are clocked on the trailing transition of input x_1 or x_2 as shown in the timing diagram and the logic diagram. Because this is a Moore machine, the outputs are a function of the inputs only; therefore, when flip-flop y_1 is set, output z_1 is asserted, as shown in the state diagram.

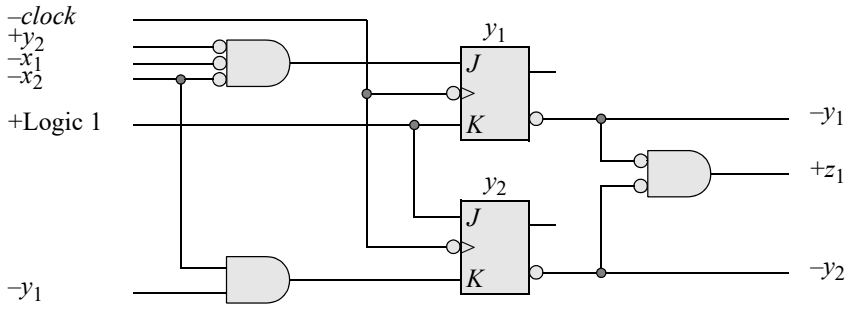
The concept of pulse-mode asynchronous sequential machines is frequently encountered in a variety of common applications such as vending machines and traffic control for demand-access intersections. Because of the stringent timing requirements on input pulse duration associated with pulse-mode machines, only the *SR* latches with edge-triggered *D* flip-flops offer a high degree of reliability.

7.7 Problems

7.1 Obtain the state diagram for the Mealy synchronous sequential machine shown below. The machine is reset to $y_1y_2 = 11$.

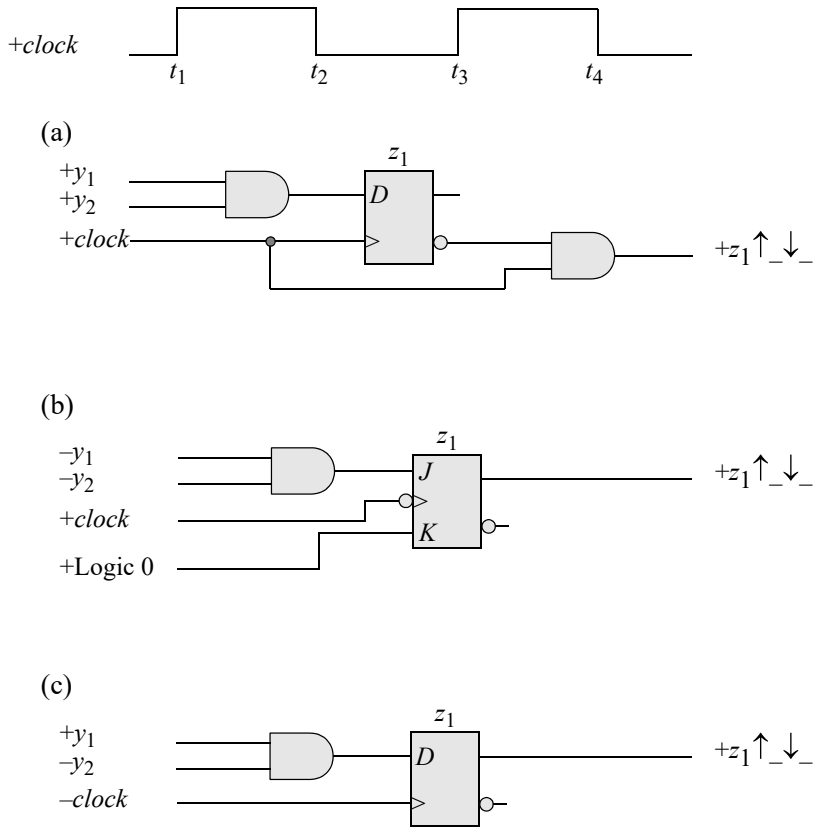


7.2 Obtain the state diagram for the Moore synchronous sequential machine shown below. The machine is reset to $y_1y_2 = 00$.

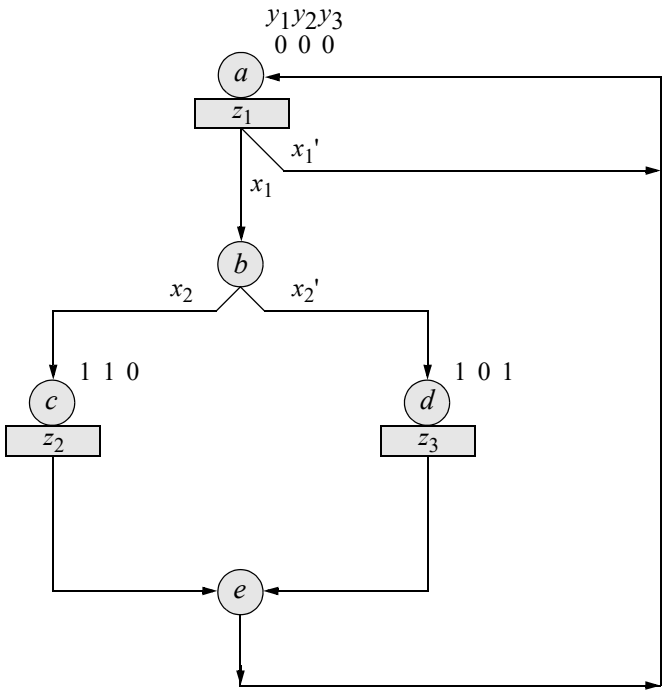


7.3 Obtain the state diagram for a Moore synchronous sequential machine that receives 4-bit words over a serial data line x_1 then generates a fifth bit z_1, which maintains odd parity over all 5 bits. There is 1 bit space words.

7.4 The λ output logic for a synchronous sequential machine is shown below for three separate cases. Specify the assertion/deassertion statement for each of the three cases. The state flip-flops are clocked on the positive transition of the clock.



7.5 Select state codes for states *b* and *e* to eliminate all output glitches for the state diagram shown below. The Moore outputs z_1, z_2, and z_3 are asserted for the entire state time; that is, $\uparrow t_1 \downarrow t_3$.



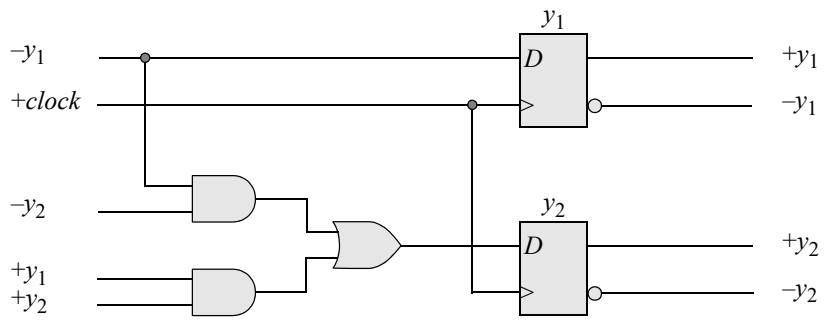
7.6 Find all equivalent states for the next-state table shown below.

Present State	Input x_1	Next State	Output z_1
a	0	b	0
	1	g	0
b	0	g	0
	1	a	1
c	0	h	0
	1	g	1
d	0	c	0
	1	a	1
e	0	h	0
	1	c	0

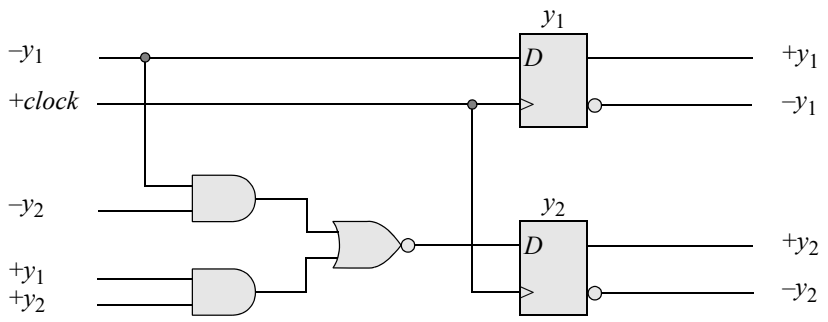
(Continued on next page)

Present State	Input x_1	Next State	Output z_1
f	0	c	0
	1	e	1
g	0	d	0
	1	g	1
h	0	c	0
	1	a	1

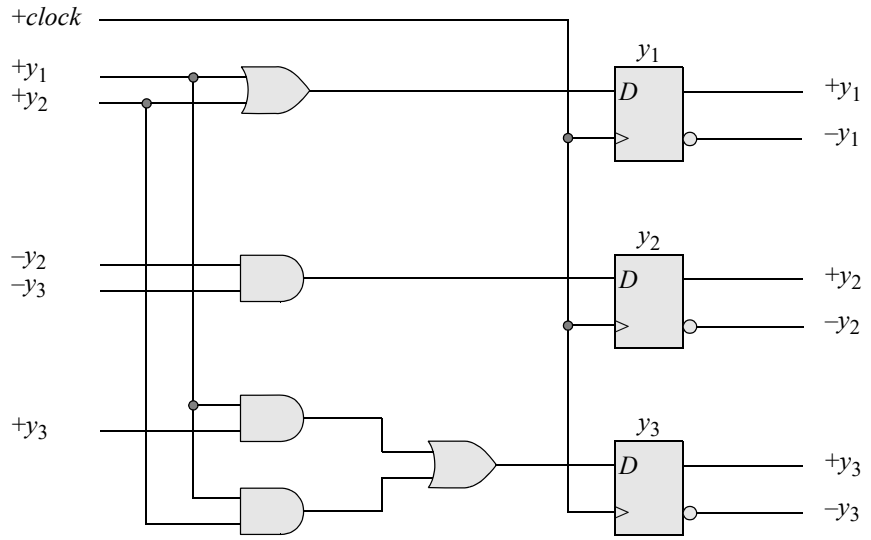
7.7 Determine the counting sequence for the counter shown below. The counter is reset initially to $y_1y_2 = 00$, where y_2 is the low-order flip-flop.



7.8 Determine the counting sequence for the counter shown below. The counter is reset initially to $y_1y_2 = 00$, where y_2 is the low-order flip-flop.



7.9 Determine the counting sequence for the counter shown below. The counter is reset initially to $y_1y_2y_3 = 000$, where y_3 is the low-order flip-flop.



7.10 Design a synchronous modulo-11 counter with no self-starting state using *JK* flip-flops.

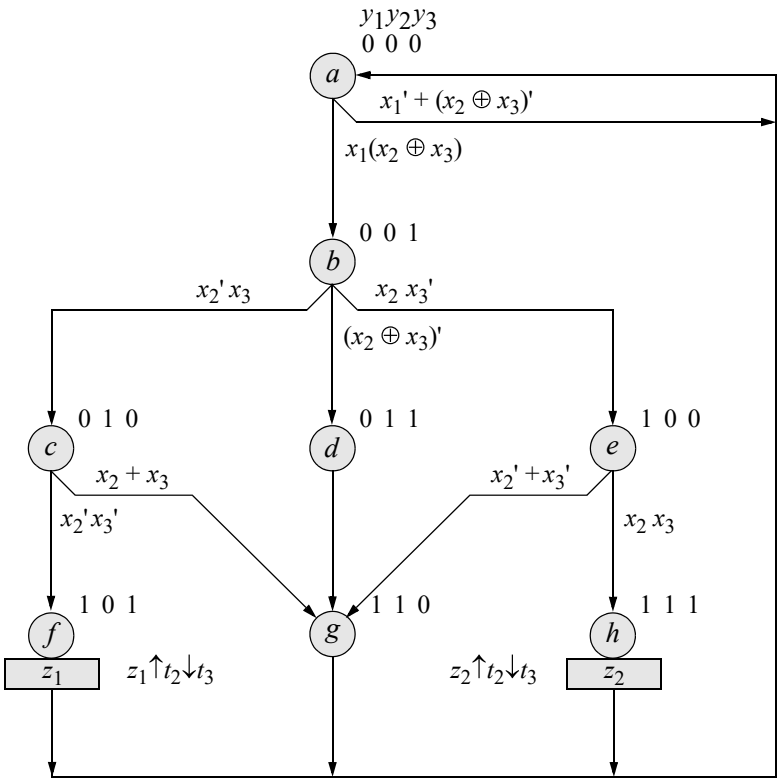
7.11 Design a counter using *D* flip-flops that counts in the following sequence:

000, 111, 001, 110, 010, 101, 011, 100, 000, . . .

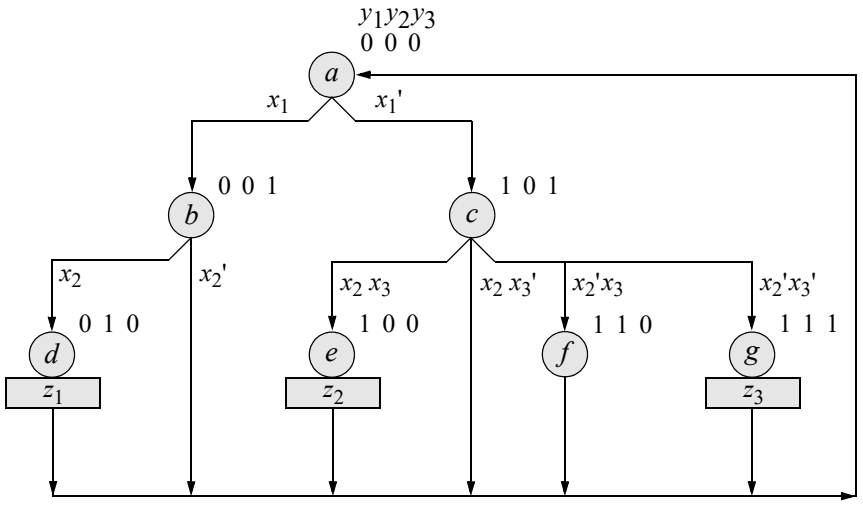
7.12 Obtain the input equations for flip-flops y_1 and y_4 only, for a BCD counter that counts in the sequence shown below. The equations are to be in minimum form. Use *JK* flip-flops. There is no self-starting state.

y_1	y_2	y_3	y_4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
0	0	0	0

7.13 Given the state diagram shown below for a Moore synchronous sequential machine with three inputs x_1, x_2, and x_3, derive the input maps for flip-flops y_1, y_2, and y_3. Then design the machine using linear-select multiplexers and logic primitives for the δ next-state logic. Use *D* flip-flops as the storage elements. Outputs $z_1$and z_2 have the following assertion/deassertion times: $\uparrow t_2 \downarrow t_3$.



7.14 Given the state diagram shown below for a Moore synchronous sequential machine, implement the design using nonlinear-select multiplexers for the δ next-state logic together with logic primitives. Let flip-flops $y_1y_3 = s_1s_0$. Use positive-edge-triggered *D* flip-flops for the storage elements and a decoder for the λ output logic. Since the state codes may cause glitches on the outputs for some state transition sequences, z_1, z_2, and z_3 should be asserted at time t_2 and deasserted at time t_3.



7.15 Design a Moore machine that has four parallel inputs x_1, x_2, x_3, and x_4 and two outputs z_1 and z_2. Output $z_1 \uparrow t_1 \downarrow t_3$; output $z_2 \uparrow t_2 \downarrow t_3$. The inputs constitute a 4-bit word. There is 1 bit space between words. Use positive-edge-triggered *D* flip-flops as the storage elements. The machine operates as follows:

(a) Output $z_1 = 1$ if the 4-bit word contains an odd number of 1s.
(b) Output $z_2 = 1$ if the 4-bit word contains an even number of 1s, but not zero 1s.

7.16 Given the Karnaugh shown below for function z_1, analyze all transitions resulting from a single change to the input vector and identify all static-1 and static-0 hazards. The inputs are active high. Obtain the sum-of-products equation containing the hazard cover and the product-of-sums equation containing the hazard cover.

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	1	1	0	0
0 1	1	1	0	1
1 1	0	0	0	1
1 0	0	0	0	0

z_1

7.17 Given the equation for z_1 shown below, identify all static-1 hazards and determine the hazard covers.

$$z_1 = x_1'x_3'x_5' + x_2x_3'x_4x_5' + x_2x_3x_4x_5 + x_1x_3'x_4x_5$$

7.18 Given the excitation map shown below for excitation variables Y_{1e} and Y_{2e}, obtain all possible state transition sequences that result in one or more oscillations. Indicate the starting stable state and the path that is taken when a single input changes value; for example, $\textcircled{q} \rightarrow r \leftrightarrow s$.

$y_{1f}y_{2f}$ \ x_1x_2	0 0	0 1	1 1	1 0
0 0	*a* 10	*b* 01	*c* 10	*d* (00)
0 1	*e* (01)	*f* 00	*g* 11	*h* 11
1 1	*i* 10	*j* 01	*k* (11)	*l* 01
1 0	*m* (10)	*n* 11	*o* 11	*p* 11

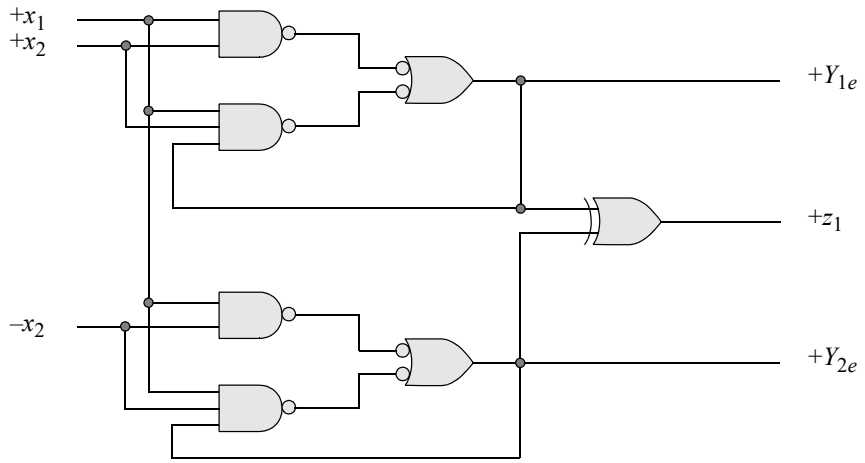
$Y_{1e}Y_{2e}$

7.19 Given the excitation map shown below for an asynchronous sequential machine, list all possible transitions (paths) that could occur for noncritical races, critical races, and oscillations. Indicate the starting stable state and the path that is taken when a single input changes.

$y_{1f}y_{2f}$ \ x_1x_2	0 0	0 1	1 1	1 0
0 0	*a* (00)	*b* (00)	*c* 10	*d* (00)
0 1	*e* 10	*f* 00	*g* (01)	*h* (01)
1 1	*i* (11)	*j* 10	*k* 01	*l* (11)
1 0	*m* 11	*n* 11	*o* (10)	*p* (10)

$Y_{1e}Y_{2e}$

7.20 Analyze the Moore asynchronous sequential machine shown below by generating an excitation map and a flow table. Let the machine be initially reset to $y_{1f}y_{2f}x_1x_2 = 0000$. Determine the shortest input sequence which will assert output z_1.

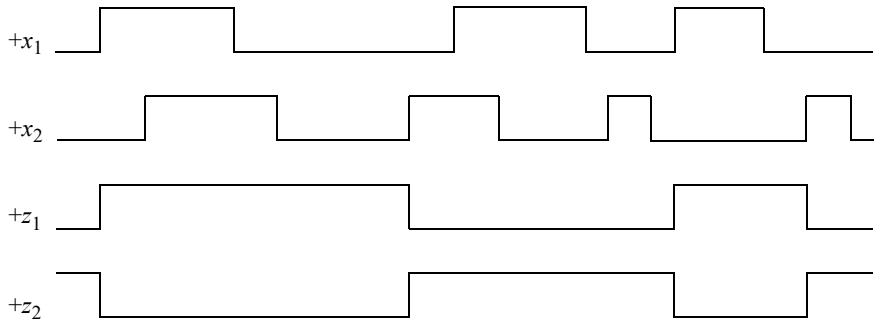


7.21 The equations for excitation variables Y_{1e}, Y_{2e}, and output z_1 are shown below for a Mealy asynchronous sequential machine.

$$Y_{1e} = x_1x_2' + y_{1f}y_{2f}'x_2 + y_{2f}x_1'x_2$$
$$Y_{2e} = x_1'x_2 + y_{2f}x_2$$
$$z_1 = y_{1f}y_{2f}x_2 + y_{1f}'y_{2f}'x_2$$

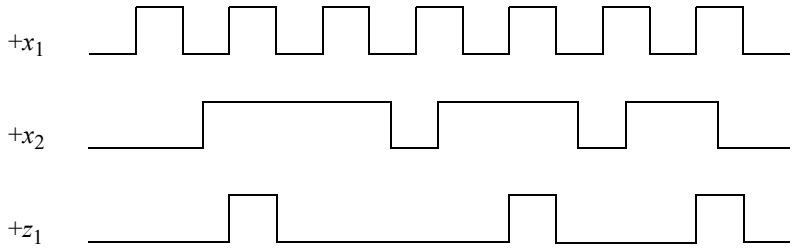
(a) Obtain the excitation maps for the individual excitation variables.
(b) Obtain the combined excitation map and indicate the stable states.
(c) Obtain the output map for z_1.

7.22 Obtain the primitive flow table — not reduced — for an asynchronous sequential machine that has two inputs x_1 and x_2 and two outputs z_1 and z_2. If x_1 is asserted before x_2, then z_1 is asserted and remains active until z_2 is asserted. If x_2 is asserted before x_1, then z_2 is asserted and remains active until z_1 is asserted. Except for a reset condition, the outputs are mutually exclusive. The timing diagram shown below represents some typical input sequences.



7.23 Obtain the primitive flow table for an asynchronous sequential machine that has two inputs x_1 and x_2 and one output z_1. Input x_1 is a periodic clock signal for a synchronous sequential machine under control of an asynchronous sequential machine. Input x_2 is used to control a single-step operation; that is, when x_2 is asserted, a single full width x_1 pulse — represented by output z_1 — is transferred to the synchronous sequential machine.

Input x_2 must be asserted before x_1 in order to generate a corresponding pulse on output z_1. If x_2 is deasserted before x_1 is deasserted, then output z_1 maintains its active state for the duration of x_1. A representative timing diagram is shown below.



7.24 Identify equivalent states in the primitive flow table shown below. Then eliminate redundant states and obtain a reduced primitive flow table.

x_1x_2 00	01	11	10	z_1
(a)	b	–	c	0
a	(b)	d	–	1
f	–	e	(c)	1
–	h	(d)	g	0
–	h	(e)	g	0
(f)	b	–	c	0
a	–	d	(g)	0
f	(h)	e	–	0

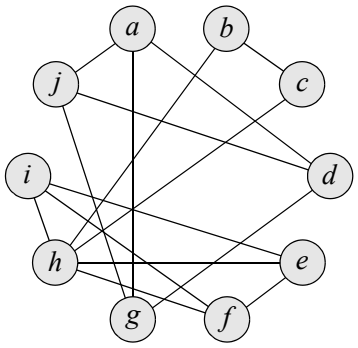
7.25 Identify equivalent states in the primitive flow table shown below. Then eliminate redundant states and obtain a reduced primitive flow table.

x_1x_2 00	01	11	10	z_1	z_2
(a)	b	–	f	0	0
a	(b)	e	–	1	0
(c)	d	–	f	0	0
a	(d)	e	–	1	0
–	g	(e)	f	0	1
c	–	e	(f)	1	1
c	(g)	e	–	1	1

7.26 Derive the merger diagram from the reduced primitive flow table below.

x_1x_2 00	01	11	10	z_1	z_2
(a)	b	–	d	0	0
a	(b)	c	–	1	1
–	j	(c)	d	1	0
a	–	c	(d)	0	1
(i)	j	–	d	0	1
i	(j)	c	–	1	0

7.27 Given the merger diagram shown below, obtain a partition which contains sets of maximal compatible rows.



7.28 Derive the merger diagram from the reduced primitive flow table shown below.

x_1x_2 00	01	11	10	z_1	z_2
(a)	b	–	d	0	0
a	(b)	c	–	0	1
–	–	(c)	d	0	0
a	–	–	(d)	1	0
–	f	(e)	d	1	0
g	(f)	c	–	1	0
(g)	h	–	d	0	1
g	(h)	e	–	1	1

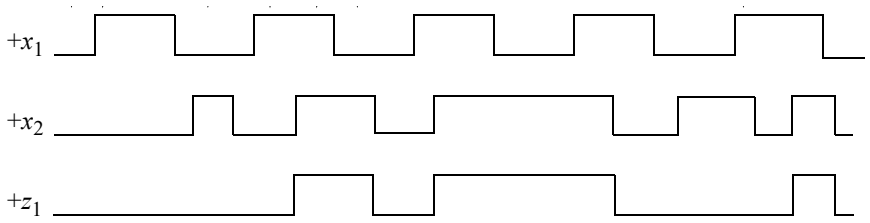
7.29 Derive the transition diagram for the merged flow table shown below. Resolve all adjacency conflicts, then obtain the excitation maps and equations in a sum-of-products notation with no static-1 hazards.

	x_1x_2 00	01	11	10
1	(a)	(b)	g	f
2	–	d	(c)	e
3	a	(d)	c	(e)
4	a	d	(g)	(f)

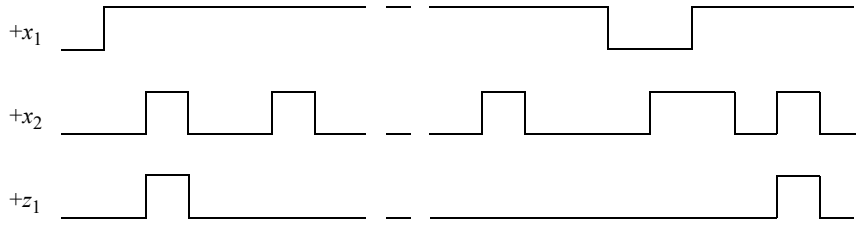
7.30 Derive the transition diagram for the merged flow table shown below. Resolve all adjacency conflicts, then obtain the excitation maps and equations in a sum-of-products notation with no static-1 hazards.

	x_1x_2 00	01	11	10
1	(a)	(b)	c	(g)
2	–	e	(c)	g
3	a	–	f	(d)
4	a	(e)	(f)	d

7.31 Synthesize an asynchronous sequential machine which has two inputs x_1 and x_2 and one output z_1. Output z_1 is asserted for the duration of x_2 if and only if x_1 is already asserted. Assume that the initial state of the machine is $x_1x_2z_1 = 000$. A representative timing diagram is shown below. Obtain the excitation equations in both a sum-of-products and a product-of-sums form with no static-1 or static-0 hazards.



7.32 Synthesize an asynchronous sequential machine which has two inputs x_1 and x_2 and one output z_1. Output z_1 will be asserted coincident with the assertion of the first x_2 pulse and will remain active for the duration of the first x_2 pulse. The output will be asserted only if the assertion of x_1 precedes the assertion of x_2. Input x_1 will not become deasserted while x_2 is asserted. The λ output logic must have a minimal number of logic gates. A representative timing diagram is shown below.

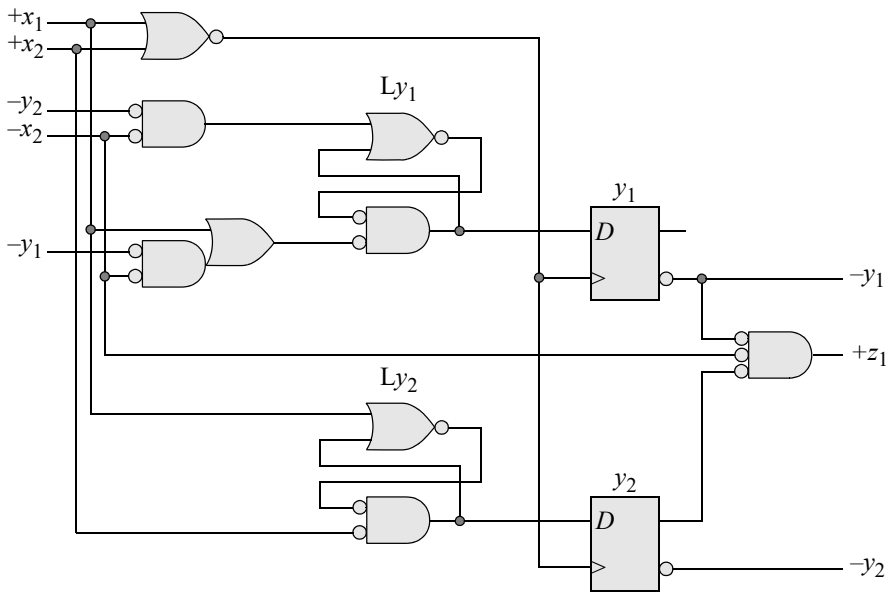


7.33 An asynchronous sequential machine has two inputs x_1 and x_2 and one output z_1. The machine operates according to the following specifications:

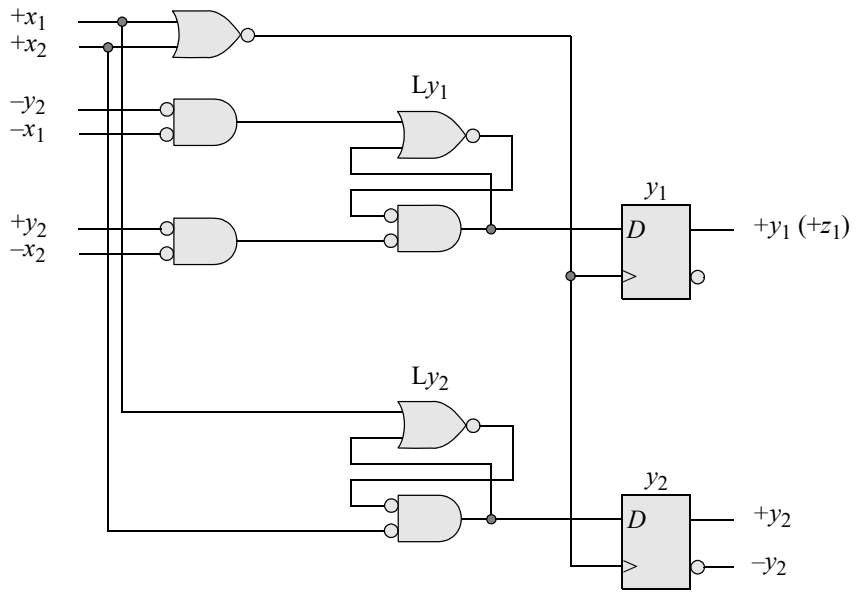
If $x_1x_2 = 00$, then the state of z_1 is unchanged.
If $x_1x_2 = 01$, then z_1 is deasserted.
If $x_1x_2 = 10$, then z_1 is asserted.
If $x_1x_2 = 11$, then z_1 changes state.

Derive the logic diagram using AND gates, OR gates, and inverters. The inputs are available in both high and low assertion. Assume that the initial conditions are $x_1x_2z_1 = 000$. The output must change as fast as possible. There must be no output glitches.

7.34 Analyze the Mealy pulse-mode sequential machine shown below. Obtain the next-state table, the input maps and equations, the output map and equation, and the state diagram.

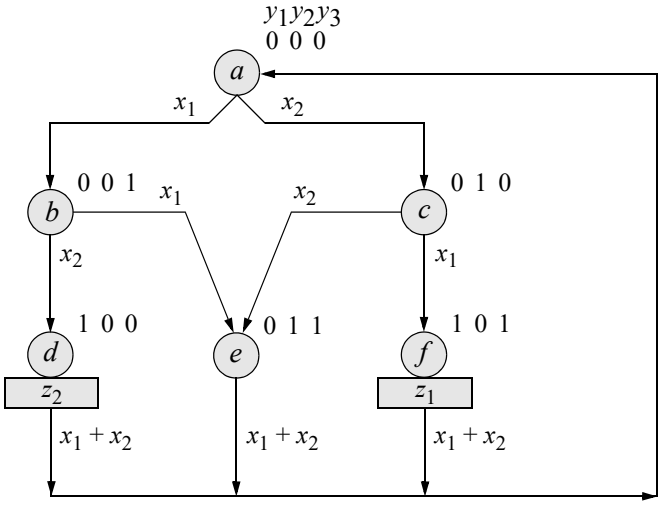


7.35 Analyze the Moore pulse-mode sequential machine shown below. Obtain the next-state table, the input maps and equations, the output map and equation, and the state diagram.



7.36 Synthesize a Moore pulse-mode sequential machine which has three inputs x_1, x_2, and x_3 and one output z_1. Output z_1 will be asserted coincident with the assertion of the x_3 pulse if and only if the x_3 pulse was preceded by an x_1 pulse followed by an x_2 pulse. That is, the input vector must be $x_1x_2x_3 = 100$, 000, 010, 000, 001 to assert z_1. Output z_1 will be deasserted at the next active x_1 pulse or x_2 pulse. Use NOR *SR* latches and positive-edge-triggered *D* flip-flops as the storage elements.

7.37 Given the state diagram shown below for a Moore pulse-mode asynchronous sequential machine, implement the machine using NAND gates for the *SR* latches and *D* flip-flops as the storage elements in a master-slave configuration. Use any type of gates for the logic primitives.


$y_1y_2y_3$
0 0 0
a
x_1
x_2
0 0 1
b
x_1
x_2
0 1 0
c
x_2
x_1
1 0 0
d
0 1 1
e
1 0 1
f
z_2
z_1
$x_1 + x_2$
$x_1 + x_2$
$x_1 + x_2$

8

Sequential Logic Design Using Verilog HDL

This chapter includes numerous examples, including some of the design examples from Chapter 7 for a comparative study of the design methodologies used in Chapter 7 with the design methodologies used in the modeling constructs of Verilog HDL. Behavioral and structural modeling constructs will be used to design synchronous sequential machines, asynchronous sequential machines, and pulse-mode asynchronous sequential machines. Dataflow modeling will be used to design asynchronous sequential machines and pulse-mode asynchronous sequential machines in conjunction with structural modeling. Some of the examples will use all three modeling constructs.

Dataflow modeling uses the *continuous assignment* statement to design combinational logic without using gates and interconnecting nets. Continuous assignment statements provide a Boolean correspondence between the right-hand side expression and the left-hand side target. The continuous assignment statement uses the keyword **assign** and has the following syntax with optional drive strength and delay:

assign [drive_strength] [delay] left-hand side target = right-hand side expression

The continuous assignment statement assigns a value to a net (**wire**) that has been previously declared — it cannot be used to assign a value to a register. Therefore, the left-hand target must be a scalar or vector net or a concatenation of scalar and vector nets. The operands on the right-hand side can be registers, nets, or function calls. The registers and nets can be declared as either scalars or vectors.

Behavioral modeling describes the *behavior* of a digital system and is not concerned with the direct implementation of logic gates but more with the architecture of the system. This is an algorithmic approach to hardware implementation and represents a higher level of abstraction than other modeling constructs — the logic details and organization are left to the synthesis tool.

Verilog contains two structured procedure statements or behaviors: **initial** and **always**. A behavior may consist of a single statement or a block of statements delimited by the keywords **begin** . . . **end**. A module may contain multiple **initial** and **always** statements. These statements are the basic statements used in behavioral modeling and execute concurrently starting at time zero in which the order of execution is not important. All other behavioral statements are contained inside these structured procedure statements.

Structural modeling consists of instantiation of one or more of the following design objects:

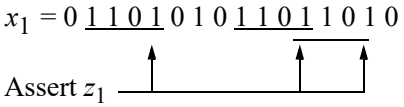
- Built-in primitives
- User-defined primitives (UDPs)
- Design modules

Instantiation means to use one or more lower-level modules — including logic primitives — that are interconnected in the construction of a higher-level structural module. A module can be a logic gate, an adder, a multiplexer, a counter, or some other logical function. The objects that are instantiated are called *instances*. Structural modeling is described by the interconnection of these lower-level logic primitives or modules. The interconnections are made by wires that connect primitive terminals or module ports.

8.1 Synchronous Sequential Machines

Several examples of Moore and Mealy finite-state synchronous sequential machines will be designed using Verilog HDL in this section. The designs will utilize behavioral and structural modeling constructs and consist of sequence detectors and counters of various moduli.

Example 8.1 A Moore machine has one input x_1 and one output z_1. Output z_1 is asserted whenever the input contains a sequence $x_1 = 1101$; overlapping sequences are allowed. An example of correct sequences is shown below. The assertion/deassertion statement for z_1 is $\uparrow t_2 \downarrow t_3$.



The design will use behavioral modeling with the **case** statement together with the **assign** statement for output z_1; therefore, a logic diagram is not necessary. The state diagram is shown in Figure 8.1 indicating the sequences to assert output z_1 and overlapping sequences. The module is shown in Figure 8.2 using behavioral modeling. The test bench is shown in Figure 8.3 and the outputs in Figure 8.4. The waveforms of Figure 8.5 show the input sequences to assert output z_1 and follow the state diagram exactly.

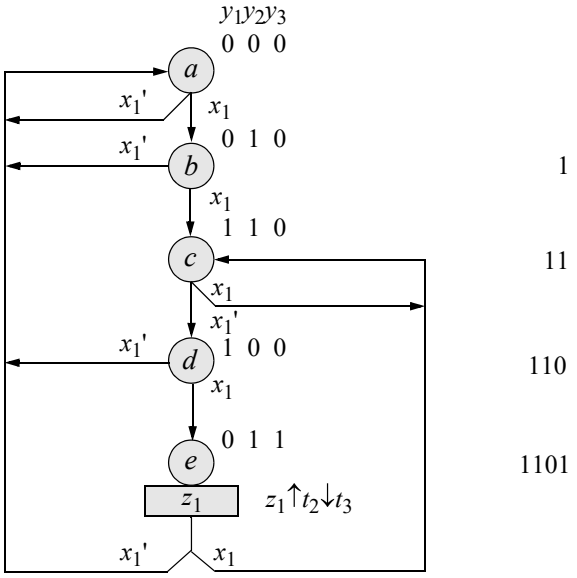


Figure 8.1 State diagram for the Moore machine of Example 8.1.

```
//behavioral moore synchronous sequential machine
module moore_ssm20 (rst_n, clk, x1, y, z1);

input rst_n, clk, x1;
output [1:3] y;
output z1;

reg [1:3] y, next_state;
wire z1;

//assign state codes
parameter   state_a = 3'b000,
            state_b = 3'b010,
            state_c = 3'b110,
            state_d = 3'b100,
            state_e = 3'b011;       //continued on next page
```

Figure 8.2 Behavioral module for the Moore machine of Example 8.1.

```
always @ (posedge clk)          //set next state
begin
   if (~rst_n)
      y <= state_a;
   else
      y <= next_state;
end

assign   z1 = y[3] & ~clk;     //define output z1

always @ (y or x1)              //determine next state
begin
   case (y)
      state_a:
         if (x1==0)
            next_state = state_a;
         else
            next_state = state_b;

      state_b:
         if (x1==0)
            next_state = state_a;
         else
            next_state = state_c;

      state_c:
         if (x1==0)
            next_state = state_d;
         else
            next_state = state_c;

      state_d:
         if (x1==0)
            next_state = state_a;
         else
            next_state = state_e;

      state_e:
         if (x1==0)
            next_state = state_a;
         else
            next_state = state_c;

      default:next_state = state_a;
   endcase
end
endmodule
```

Figure 8.2 (Continued)

```
//test bench for moore synchronous sequential machine
module moore_ssm20_tb;

reg rst_n, clk, x1;
wire [1:3] y;
wire z1;

//display variables
initial
$monitor ("x1 = %b, state = %b, z1 = %b",
            x1, y, z1);

//define clock
initial
begin
   clk = 1'b0;
   forever
      #10clk = ~clk;
end

//define input sequence
initial
begin
   #0 rst_n = 1'b0;
      x1 = 1'b0;

   #5 rst_n = 1'b1;

   x1 = 1'b0;@ (posedge clk)  //remain in state_a (000)
   x1 = 1'b1;@ (posedge clk)  //go to state_b (010)
   x1 = 1'b0;@ (posedge clk)  //go to state_a (000)
   x1 = 1'b1;@ (posedge clk)  //go to state_b (010)
   x1 = 1'b1;@ (posedge clk)  //go to state_c (110)
   x1 = 1'b0;@ (posedge clk)  //go to state_d (100)
   x1 = 1'b1;@ (posedge clk)  //go to state_e (011); assert z1
   x1 = 1'b0;@ (posedge clk)  //go to state_a (000)
   x1 = 1'b1;@ (posedge clk)  //go to state_b (010)
   x1 = 1'b1;@ (posedge clk)  //go to state_c (110)
   x1 = 1'b0;@ (posedge clk)  //go to state_d (100)
   x1 = 1'b1;@ (posedge clk)  //go to state_e (011); assert z1
   x1 = 1'b1;@ (posedge clk)  //go to state_c (110); assert z1
   x1 = 1'b0;@ (posedge clk)  //go to state_d (100); assert z1
   x1 = 1'b1;@ (posedge clk)  //go to state_e (011); assert z1
   x1 = 1'b0;@ (posedge clk)  //go to state_a (000)

   #10   $stop;
end                           //continued on next page
```

Figure 8.3 Test bench for the Moore machine of Example 8.1.

```
//instantiate the module into the test bench
moore_ssm20 inst1 (
   .rst_n(rst_n),
   .clk(clk),
   .x1(x1),
   .y(y),
   .z1(z1)
   );

endmodule
```

Figure 8.3 (Continued)

```
x1 = 0, state = xxx, z1 = x    x1 = 1, state = 010, z1 = 0
x1 = 1, state = 000, z1 = 0    x1 = 0, state = 110, z1 = 0
x1 = 0, state = 010, z1 = 0    x1 = 1, state = 100, z1 = 0
x1 = 1, state = 000, z1 = 0    x1 = 1, state = 011, z1 = 0
x1 = 1, state = 010, z1 = 0    x1 = 1, state = 011, z1 = 1
x1 = 0, state = 110, z1 = 0    x1 = 0, state = 110, z1 = 0
x1 = 1, state = 100, z1 = 0    x1 = 1, state = 100, z1 = 0
x1 = 0, state = 011, z1 = 0    x1 = 0, state = 011, z1 = 0
x1 = 0, state = 011, z1 = 1    x1 = 0, state = 011, z1 = 1
x1 = 1, state = 000, z1 = 0    x1 = 0, state = 000, z1 = 0
```

Figure 8.4 Outputs for the Moore machine of Example 8.1.

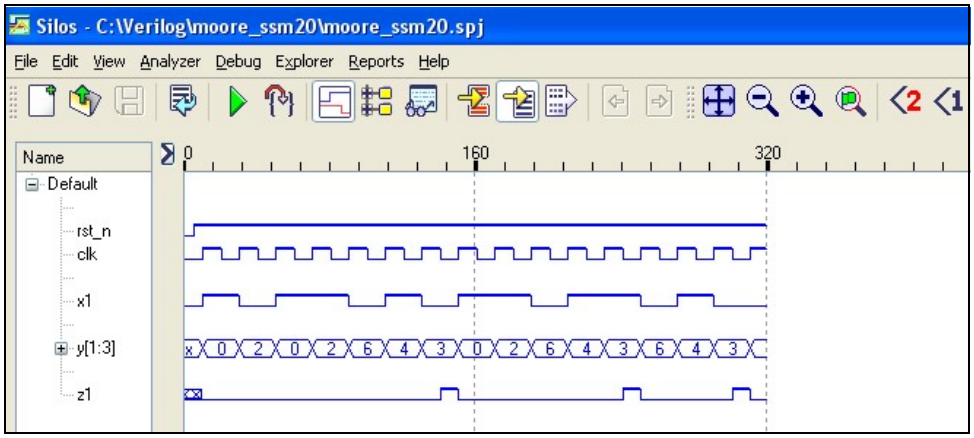


Figure 8.5 Waveforms for the Moore machine of Example 8.1.

Example 8.2 The same Moore synchronous sequential machine that was designed in Example 8.1 using behavioral modeling will now be designed using structural modeling. This will provide a comparison between behavioral modeling and structural modeling for the same machine — the results should be identical. Structural modeling usually requires algorithmic equations or a logic diagram as the basis for the module. The structural module will use built-in primitives and instantiated *D* flip-flops. The test bench that was utilized in Example 8.1 will also be used in this example.

For convenience, the state diagram is reproduced in Figure 8.6. The input maps are obtained directly from the state diagram and are shown in Figure 8.7; the *D* input equations are shown in Equation 8.1. The logic diagram is shown in Figure 8.8 and depicts the instantiation names for the built-in primitives, the net names, and the *D* flip-flop instantiations.

The structural module, test bench, outputs, and waveforms are shown in Figure 8.9, Figure 8.10, Figure 8.11, and Figure 8.12, respectively.

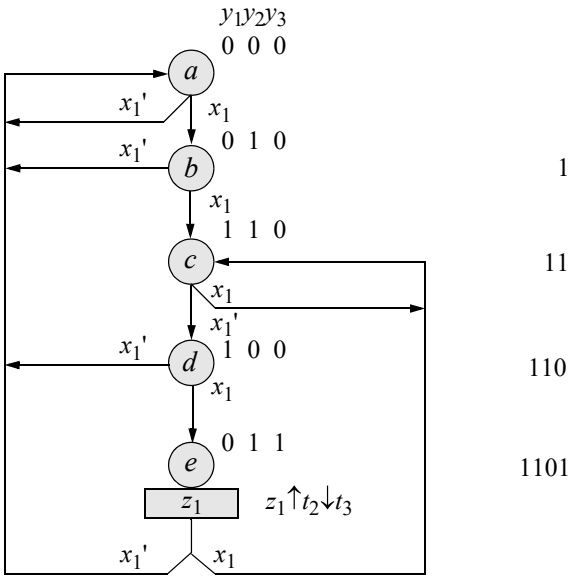
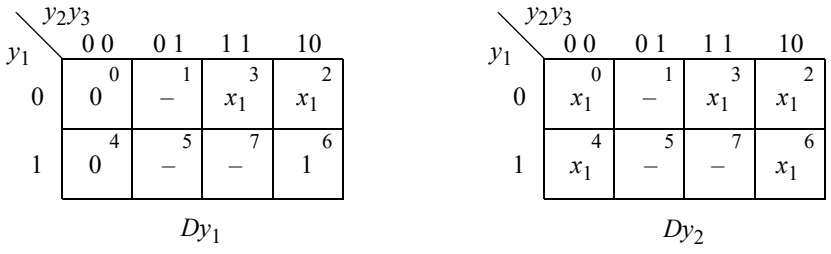


Figure 8.6 State diagram for the Moore machine of Example 8.2.



(Continued on next page)

Figure 8.7 Input maps for the *D* flip-flops of Example 8.2.

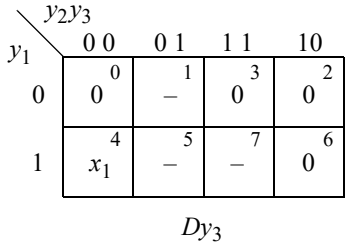


Figure 8.7 (Continued)

$$Dy_1 = y_2x_1 + y_1y_2 = y_2(x_1 + y_1)$$
$$Dy_2 = x_1$$
$$Dy_3 = y_1y_2'x_1 \tag{8.1}$$

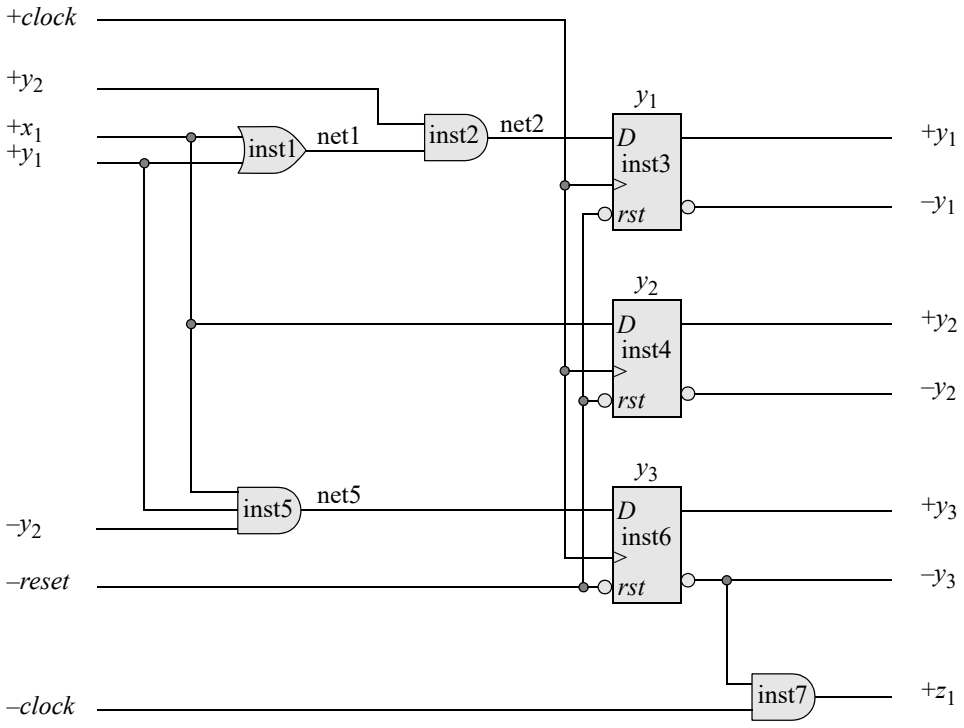


Figure 8.8 Logic diagram for the Moore machine of Example 8.2.

```
//structural moore synchronous sequential machine
module moore_ssm20a (rst_n, clk, x1, y, z1);

input rst_n, clk, x1;
output [1:3] y;
output z1;

wire rst_n, clk, x1;
wire [1:3] y;
wire z1;

//define internal wires
wire net1, net2, net5;

//instantiate the logic for flip-flop y[1]
or    inst1 (net1, x1, y[1]);
and   inst2 (net2, y[2], net1);

//instantiate the D flip-flop for y[1]
d_ff_bh inst3 (
   .rst_n(rst_n),
   .clk(clk),
   .d(net2),
   .q(y[1])
   );

//instantiate the D flip-flop for y[2]
d_ff_bh inst4 (
   .rst_n(rst_n),
   .clk(clk),
   .d(x1),
   .q(y[2])
   );

//instantiate the logic for flip-flop y[3]
and   inst5 (net5, x1, y[1], ~y[2]);

//instantiate the D flip-flop for y[3]
d_ff_bh inst6 (
   .rst_n(rst_n),
   .clk(clk),
   .d(net5),
   .q(y[3])
   );

and   inst7 (z1, y[3], ~clk); //instantiate the logic for z1
endmodule
```

Figure 8.9 Structural module for the Moore machine of Example 8.2.

```
//test bench for moore synchronous sequential machine
module moore_ssm20a_tb;

reg rst_n, clk, x1;
wire [1:3] y;
wire z1;

//display variables
initial
$monitor ("x1 = %b, state = %b, z1 = %b",
            x1, y, z1);

//define clock
initial
begin
   clk = 1'b0;
   forever
      #10clk = ~clk;
end

//define input sequence
initial
begin
   #0 rst_n = 1'b0;
      x1 = 1'b0;

   #5 rst_n = 1'b1;

   x1 = 1'b0;@ (posedge clk)  //remain in state_a (000)
   x1 = 1'b1;@ (posedge clk)  //go to state_b (010)
   x1 = 1'b0;@ (posedge clk)  //go to state_a (000)
   x1 = 1'b1;@ (posedge clk)  //go to state_b (010)
   x1 = 1'b1;@ (posedge clk)  //go to state_c (110)
   x1 = 1'b0;@ (posedge clk)  //go to state_d (100)
   x1 = 1'b1;@ (posedge clk)  //go to state_e (011); assert z1
   x1 = 1'b0;@ (posedge clk)  //go to state_a (000)
   x1 = 1'b1;@ (posedge clk)  //go to state_b (010)
   x1 = 1'b1;@ (posedge clk)  //go to state_c (110)
   x1 = 1'b0;@ (posedge clk)  //go to state_d (100)
   x1 = 1'b1;@ (posedge clk)  //go to state_e (011); assert z1
   x1 = 1'b1;@ (posedge clk)  //go to state_c (110); assert z1
   x1 = 1'b0;@ (posedge clk)  //go to state_d (100); assert z1
   x1 = 1'b1;@ (posedge clk)  //go to state_e (011); assert z1
   x1 = 1'b0;@ (posedge clk)  //go to state_a (000)

   #10   $stop;
end                           //continued on next page
```

Figure 8.10 Test bench for the Moore machine of Example 8.2.

```
//instantiate the module into the test bench
moore_ssm20a inst1 (
   .rst_n(rst_n),
   .clk(clk),
   .x1(x1),
   .y(y),
   .z1(z1)
   );

endmodule
```

Figure 8.10 (Continued)

```
x1 = 0, state = 000, z1 = 0     x1 = 1, state = 010, z1 = 0
x1 = 1, state = 000, z1 = 0     x1 = 0, state = 110, z1 = 0
x1 = 0, state = 010, z1 = 0     x1 = 1, state = 100, z1 = 0
x1 = 1, state = 000, z1 = 0     x1 = 1, state = 011, z1 = 0
x1 = 1, state = 010, z1 = 0     x1 = 1, state = 011, z1 = 1
x1 = 0, state = 110, z1 = 0     x1 = 0, state = 110, z1 = 0
x1 = 1, state = 100, z1 = 0     x1 = 1, state = 100, z1 = 0
x1 = 0, state = 011, z1 = 0     x1 = 0, state = 011, z1 = 0
x1 = 0, state = 011, z1 = 1     x1 = 0, state = 011, z1 = 1
x1 = 1, state = 000, z1 = 0     x1 = 0, state = 000, z1 = 0
```

Figure 8.11 Outputs for the Moore machine of Example 8.2.

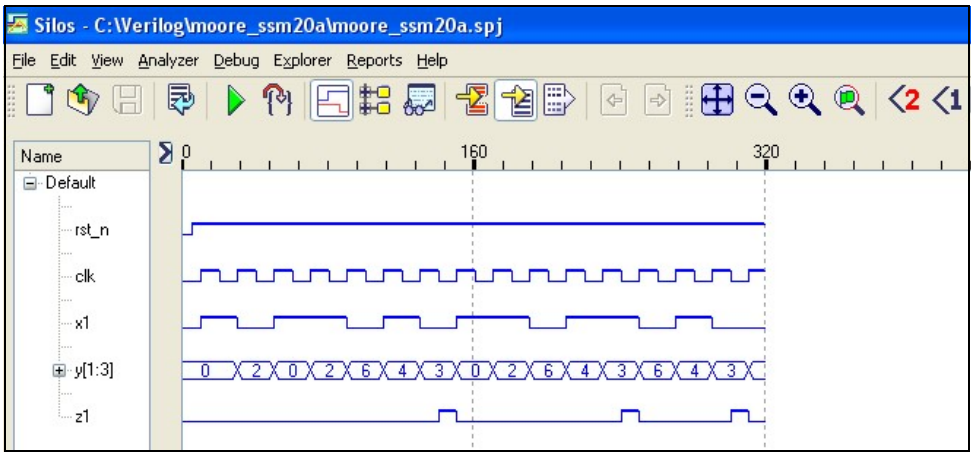


Figure 8.12 Waveforms for the Moore machine of Example 8.2.

Example 8.3 The state diagram for a Moore synchronous sequential machine is shown in Figure 8.13 which has two inputs x_1 and x_2 together with two outputs z_1 and z_2, both of which have the following assertion/deassertion: $\uparrow t_2 \downarrow t_3$. Behavioral modeling will be used in the implementation; therefore, the machine can be designed directly from the state diagram.

The behavioral module using the **case** statement is shown in Figure 8.14 — outputs z_1 and z_2 are defined using the continuous assignment statement of dataflow modeling. The test bench module is shown in Figure 8.15 using the **$random** system task to generate a random value for certain inputs when their value can be considered a "don't care" — either 0 or 1. The outputs and waveforms are shown in Figure 8.16 and Figure 8.17, respectively.

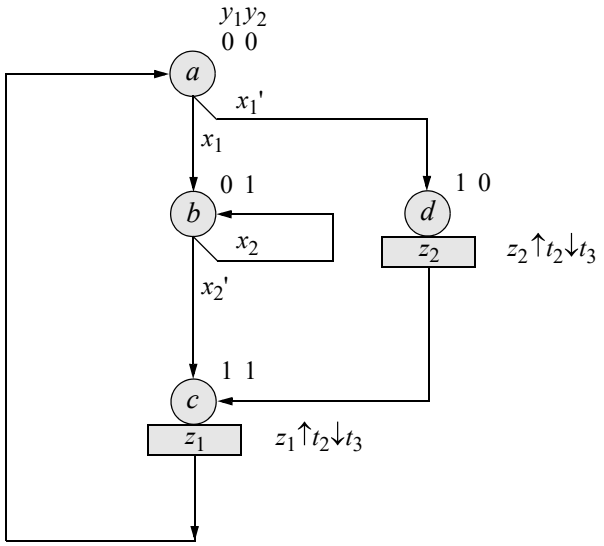


Figure 8.13 State diagram for the Moore machine of Example 8.3.

```
//behavioral moore synchronous sequential machine
module moore_ssm14 (clk, rst_n, x1, x2, y, z1, z2);

input clk, rst_n, x1, x2;
output [1:2] y;
output z1, z2;

reg [1:2] y, next_state;
wire z1, z2;

//continued on next page
```

Figure 8.14 Behavioral module for the Moore machine of Example 8.3.

```
//assign state codes
parameter   state_a = 2'b00,
            state_b = 2'b01,
            state_c = 2'b11,
            state_d = 2'b10;

//set next state
always @ (posedge clk)
begin
   if (~rst_n)
      y <= state_a;
   else
      y <= next_state;
end

//define outputs
assign   z1 = (y[1] & y[2] & ~clk);
assign   z2 = (y[1] & ~y[2] & ~clk);

//determine next state
always @ (y or x1 or x2)
begin
   case (y)
      state_a:
         if (x1==0)
            next_state = state_d;
         else
            next_state = state_b;

      state_b:
         if (x2==0)
            next_state = state_c;
         else
            next_state = state_b;

      state_c: next_state = state_a;

      state_d: next_state = state_c;

      default: next_state = state_a;

   endcase
end

endmodule
```

Figure 8.14 (Continued)

```
//test bench for moore synchronous sequential machine
module moore_ssm14_tb;

reg clk, rst_n, x1, x2;
wire [1:2] y;
wire z1, z2;

//display variables
initial
$monitor ("x1x2=%b, state=%b, z1z2=%b",
         {x1, x2}, y, {z1, z2});

//define clock
initial
begin
   clk = 1'b0;
   forever
      #10clk = ~clk;
end

//define input sequence
initial
begin
   #0 rst_n = 1'b0;     //reset to state_a
      x1 = 1'b0;
      x2 = 1'b0;
   #5 rst_n = 1'b1;

      x1 = 1'b1;  x2 = $random;
      @ (posedge clk)

      x1 = 1'b1;  x2 = $random;
      @ (posedge clk)   //go to state_b (01)

      x1 = $random;  x2 = 1'b1;
      @ (posedge clk)   //remain in state_b (01)

      x1 = $random;  x2 = 1'b0;
      @ (posedge clk)   //go to state_c (11)

      x1 = $random;  x2 = $random;
      @ (posedge clk)   //go to state_a (00)

      x1 = 1'b0;  x2 = $random;
      @ (posedge clk)   //go to state_d (10)

//continued on next page
```

Figure 8.15 Test bench for the Moore machine of Example 8.3.

```
      x1 = $random;  x2 = $random;
      @ (posedge clk)    //go to state_c (11)

      x1 = $random;  x2 = $random;
      @ (posedge clk)    //go to state_a (00)
   #10   $stop;
end

moore_ssm14 inst1 (  //instantiate the module
   .clk(clk),
   .rst_n(rst_n),
   .x1(x1),
   .x2(x2),
   .y(y),
   .z1(z1),
   .z2(z2)
   );
endmodule
```

Figure 8.15 (Continued)

```
x1x2=00, state=xx, z1z2=xx
x1x2=10, state=xx, z1z2=xx
x1x2=11, state=00, z1z2=00
x1x2=11, state=01, z1z2=00
x1x2=10, state=01, z1z2=00
x1x2=11, state=11, z1z2=00
```

```
x1x2=11, state=11, z1z2=10
x1x2=01, state=00, z1z2=00
x1x2=01, state=10, z1z2=00
x1x2=01, state=10, z1z2=01
x1x2=10, state=11, z1z2=00
x1x2=10, state=11, z1z2=10
x1x2=10, state=00, z1z2=00
```

Figure 8.16 Outputs for the Moore machine of Example 8.3.

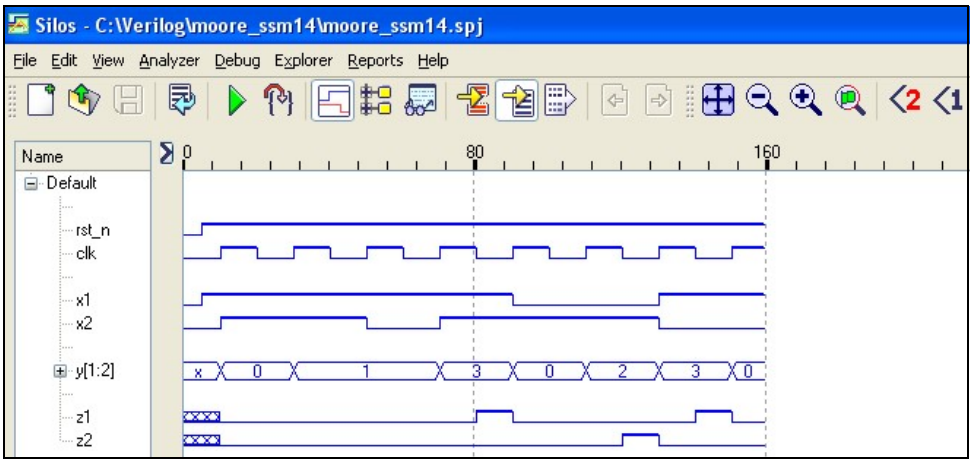


Figure 8.17 Waveforms for the Moore machine of Example 8.3.

Example 8.4 A Mealy synchronous sequential machine will be designed using behavioral modeling with the **case** statement and the **parameter** keyword. Output z_1 is asserted whenever a serial input data line contains a 3-bit word with an odd number of 1s. The format for the serial data line is shown below,

$$x_1 = \left| b_1 \;\; b_2 \;\; b_3 \right| b_1 \;\; b_2 \;\; b_3 \left| b_1 \;\; b_2 \;\; b_3 \right| \ldots$$

where $b_i = 0$ or 1. There is no space between words. Output z_1 is asserted during the first half of the third bit time b_3; that is, $z_1 \uparrow t_1 \downarrow t_2$.

The state diagram is shown in Figure 8.18 in which the state codes have been selected so that there will be no glitches on output z_1 for any state transition sequence. There are three unused states: $y_1y_2y_3$ = 101, 110, and 111. If two or more flip-flops change value for a state transition sequence, then the machine may momentarily pass through either an unused state or a state in which there is no output — in both cases there will be no glitch on output z_1.

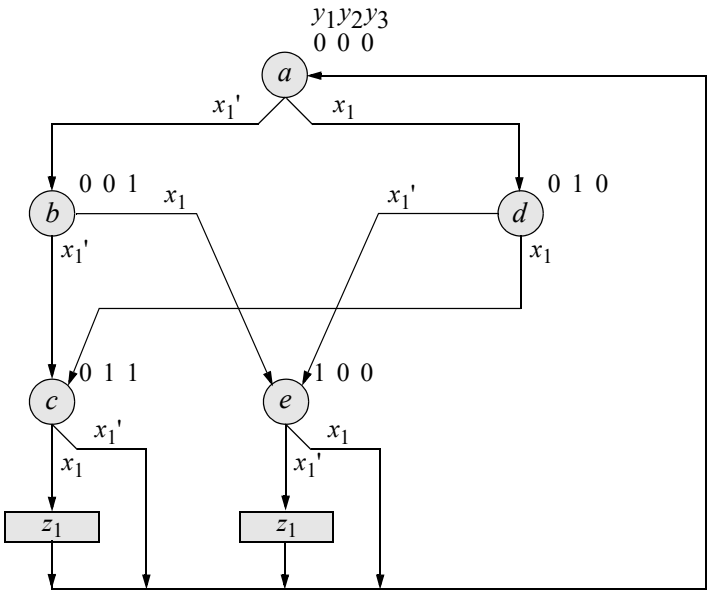


Figure 8.18 State diagram for the Mealy machine of Example 8.4.

The behavioral module is shown in Figure 8.19. The **default** keyword should be inserted at the end of a block containing a **case** statement. The test bench module is shown in Figure 8.20. Every state transition sequence requires a specific value for input x_1; therefore, the **$random** system task cannot be used. The outputs and waveforms are shown in Figure 8.21 and Figure 8.22, respectively.

```
//behavioral mealy synchronous sequential machine
module mealy_ssm7 (rst_n, clk, x1, y, z1);

input rst_n, clk, x1;
output [1:3] y;
output z1;

reg [1:3] y, next_state;
wire z1;

//assign state codes
parameter   state_a = 3'b000,
            state_b = 3'b001,
            state_c = 3'b011,
            state_d = 3'b010,
            state_e = 3'b100;

//set next state
always @ (posedge clk)
begin
   if (~rst_n)
      y <= state_a;
   else
      y <= next_state;
end

//define output
assign z1 = (y[1] & ~x1 & clk) | (y[2] & y[3] & x1 & clk);

//determine next state
always @ (y or x1)
begin
   case (y)
      state_a:
         if (x1==0)
            next_state = state_b;
         else
            next_state = state_d;

      state_b:
         if (x1==0)
            next_state = state_c;
         else
            next_state = state_e;

//continued on next page
```

Figure 8.19 Behavioral module for the Mealy machine of Example 8.4.

```
      state_c: next_state = state_a;

      state_d:
         if (x1==0)
            next_state = state_e;
         else
            next_state = state_c;

      state_e: next_state = state_a;

      default: next_state = state_a;
   endcase
end
endmodule
```

Figure 8.19 (Continued)

```
//test bench for mealy synchronous sequential machine
module mealy_ssm7_tb;

reg rst_n, clk, x1;
wire [1:3] y;
wire z1;

//display variables
initial
$monitor ("x1=%b, state= %b, z1=%b",
            x1, y, z1);

//define clock
initial
begin
   clk = 1'b0;
   forever
      #10   clk = ~clk;
end

//define input sequence
initial
begin
   #0 rst_n = 1'b0;
      x1 = 1'b0;

   #5 rst_n = 1'b1;
//continued on next page
```

Figure 8.20 Test bench for the Mealy machine of Example 8.4.

```
      x1 = 1'b0; @ (posedge clk) //go to state_b (001)
      x1 = 1'b0; @ (posedge clk) //go to state_c (011)
      x1 = 1'b1; @ (posedge clk) //go to state_a (000)
      x1 = 1'b0; @ (posedge clk) //go to state_b (001)
      x1 = 1'b1; @ (posedge clk) //go to state_e (100); set z1
      x1 = 1'b0; @ (posedge clk) //go to state_a (000)
      x1 = 1'b1; @ (posedge clk) //go to state_d (010)
      x1 = 1'b0; @ (posedge clk) //go to state_e (100)
      x1 = 1'b1; @ (posedge clk) //go to state_a (000)
      x1 = 1'b1; @ (posedge clk) //go to state_d (010)
      x1 = 1'b1; @ (posedge clk) //go to state_c (011); set z1
      x1 = 1'b1; @ (posedge clk) //go to state_a (000)
      x1 = 1'b1; @ (posedge clk) //go to state_b (001)

   #10   $stop;
end

//instantiate the module into the test bench
mealy_ssm7 inst1 (
   .rst_n(rst_n),
   .clk(clk),
   .x1(x1),
   .y(y),
   .z1(z1)
   );
endmodule
```

Figure 8.20 (Continued)

```
x1=0, state= xxx, z1=0
x1=0, state= 000, z1=0
x1=1, state= 001, z1=0
x1=0, state= 100, z1=1
x1=0, state= 100, z1=0
x1=1, state= 000, z1=0
x1=0, state= 010, z1=0
x1=1, state= 100, z1=0
x1=0, state= 000, z1=0
x1=1, state= 001, z1=0
x1=1, state= 100, z1=0
x1=1, state= 000, z1=0
x1=1, state= 010, z1=0
x1=1, state= 011, z1=1
x1=1, state= 011, z1=0
x1=1, state= 000, z1=0
```

Figure 8.21 Outputs for the Mealy machine of Example 8.4.

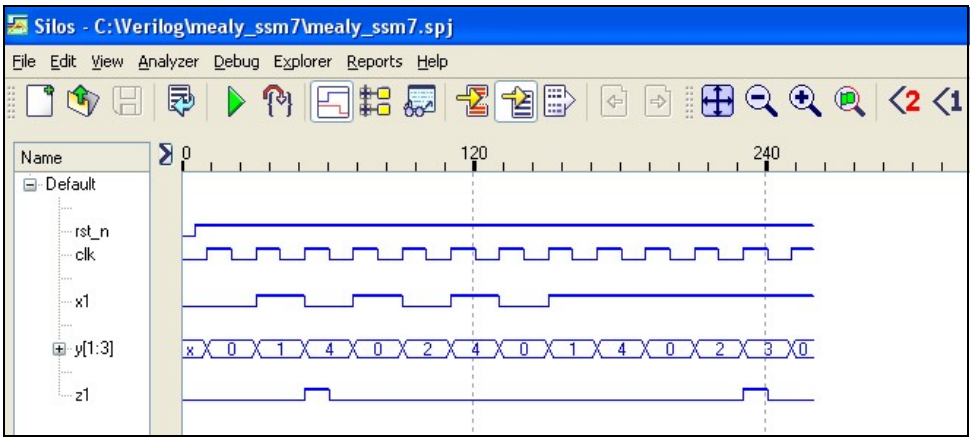


Figure 8.22 Waveforms for the Mealy machine of Example 8.4.

Example 8.5 A counter will be designed using structural modeling that counts in the sequence shown in the chart below. The storage elements will be positive-edge-triggered *D* flip-flops. The input maps are shown in Figure 8.23 and the input equations are shown in Equation 8.4. The logic diagram is shown in Figure 8.24.

y_1	y_2	y_3
0	0	0
1	1	1
0	0	1
1	1	0
0	1	0
1	0	1
0	1	1
1	0	0
0	0	0

The structural module is shown in Figure 8.25 using module instantiation of logic primitives that were previously designed and a positive-edge-triggered *D* flip-flop. The test bench is shown in Figure 8.26. The only inputs are the clock and reset variables. The duration of simulation is 160 time units, which is sufficient to show the complete counting sequence. The outputs are shown in Figure 8.27 which displays the counting sequence. The waveforms are shown in Figure 8.28.

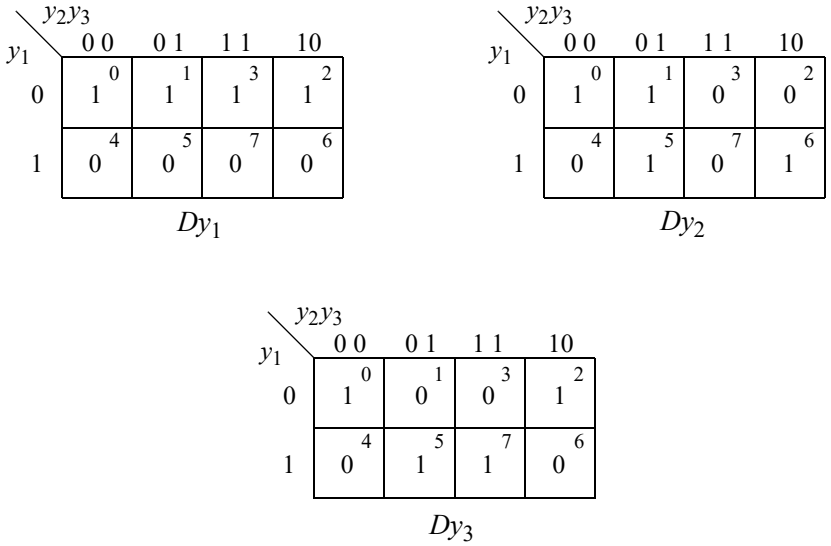


Figure 8.23 Input maps for the nonsequential counter of Example 8.5.

$$
\begin{aligned}
Dy_1 &= y_1' \\
Dy_2 &= y_1'y_2' + y_2'y_3 + y_1y_2y_3' \\
Dy_3 &= y_1'y_3' + y_1y_3 = (y_1 \oplus y_3)'
\end{aligned} \tag{8.4}
$$

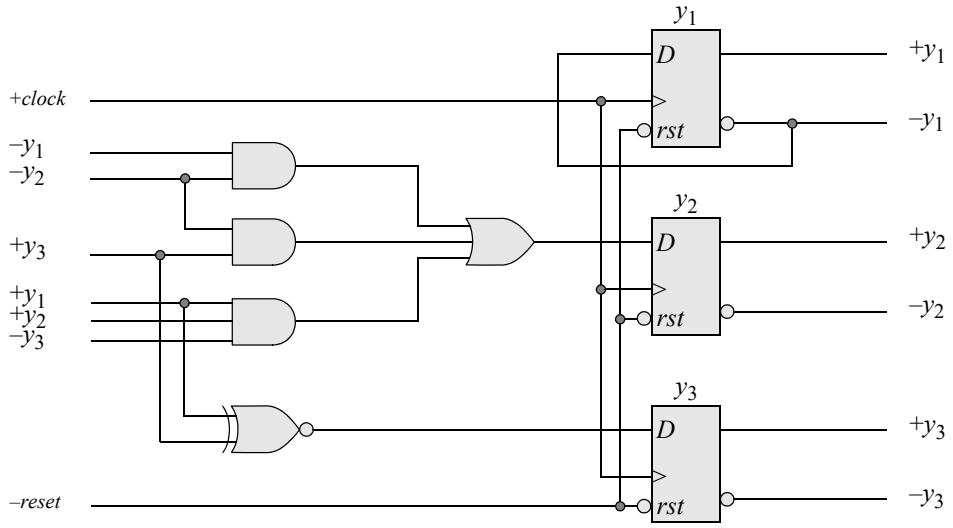


Figure 8.24 Logic diagram for the nonsequential counter of Example 8.5.

```
//structural nonsequential counter
module ctr_non_seq4 (rst_n, clk, y);

input rst_n, clk;
output [1:3] y;

wire [1:3] y;

//define internal nets
wire net2, net3, net4, net5, net7;

//instantiate the flip-flop for y[1]
d_ff_bh inst1 (
   .rst_n(rst_n),
   .clk(clk),
   .d(~y[1]),
   .q(y[1])
   );

//instantiate the logic and D flip-flop for y[2]
and2_df inst2 (
   .x1(~y[1]),
   .x2(~y[2]),
   .z1(net2)
   );

and2_df inst3 (
   .x1(~y[2]),
   .x2(y[3]),
   .z1(net3)
   );

and3_df inst4 (
   .x1(y[1]),
   .x2(y[2]),
   .x3(~y[3]),
   .z1(net4)
   );

or3_df inst5 (
   .x1(net2),
   .x2(net3),
   .x3(net4),
   .z1(net5)
   );

//continued on next page
```

Figure 8.25 Structural module for the nonsequential counter of Example 8.5.

```
d_ff_bh inst6 (
   .rst_n(rst_n),
   .clk(clk),
   .d(net5),
   .q(y[2])
   );

//instantiate the logic and D flip-flop for y[3]
xnor2_df inst7 (
   .x1(y[1]),
   .x2(y[3]),
   .z1(net7)
   );

d_ff_bh inst8 (
   .rst_n(rst_n),
   .clk(clk),
   .d(net7),
   .q(y[3])
   );

endmodule
```

Figure 8.25 (Continued)

```
//test bench for nonsequential counter
module ctr_non_seq4_tb;

reg rst_n, clk;
wire [1:3] y;

//display count
initial
$monitor ("count = %b", y);

//define reset
initial
begin
   #0 rst_n = 1'b0;
   #5 rst_n = 1'b1;
end

//continued on next page
```

Figure 8.26 Test bench for the nonsequential counter of Example 8.5.

```
//define clock
initial
begin
   clk = 1'b0;
   forever
      #10clk = ~clk;
end

//define simulation time
initial
begin
   #160 $finish;
end

ctr_non_seq4 inst1 (       //instantiate the module
   .rst_n(rst_n),
   .clk(clk),
   .y(y)
   );
endmodule
```

Figure 8.26 (Continued)

```
count = 000
count = 111
count = 001
count = 110
count = 010
count = 101
count = 011
count = 100
count = 000
count = 111
```

Figure 8.27 Outputs for the nonsequential counter of Example 8.5.

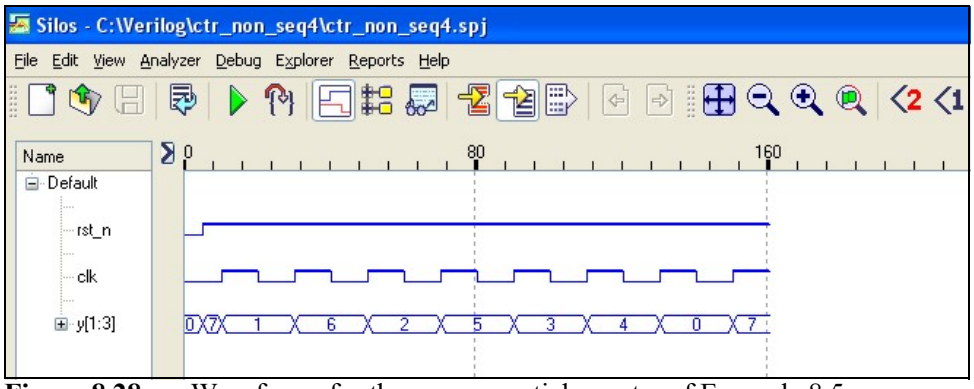
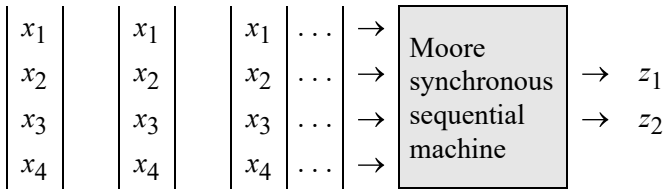


Figure 8.28 Waveforms for the nonsequential counter of Example 8.5.

Example 8.6 A Moore synchronous sequential machine will be designed using structural modeling that has four parallel inputs x_1, x_2, x_3, and x_4 and two outputs z_1 and z_2, as shown below.



The inputs constitute a 4-bit word. There is one bit space between words. The machines operates as follows:

(a) Output $z_1 = 1$ if the 4-bit word contains an odd number of 1s.
(b) Output $z_2 = 1$ if the 4-bit word contains an even number of 1s, but not zero 1s.

The state diagram is shown in Figure 8.29 depicting one path for output z_1 and one path for output z_2. The assertion/deassertion statements for the outputs are: $z_1 \uparrow t_1 \downarrow t_3$ and $z_2 \uparrow t_2 \downarrow t_3$. The truth table for the machine is shown in Table 8.1, indicating the words that contain an even or odd number of 1s. The equation for an odd number of 1s — obtained from the truth table — is shown in Equation 8.5; the equation for an even number of 1s — but not zero 1s — is shown in Equation 8.6.

The logic diagram is shown in Figure 8.30 as obtained from Equation 8.5 and Equation 8.6, where flip-flop y_1 represents an even number of 1s and flip-flop y_2 represents an odd number of 1s. The output logic is obtained from the state diagram. The structural design module and the test bench module are shown in Figure 8.31 and Figure 8.32, respectively. The outputs and waveforms are shown in Figure 8.33 and Figure 8.34, respectively.

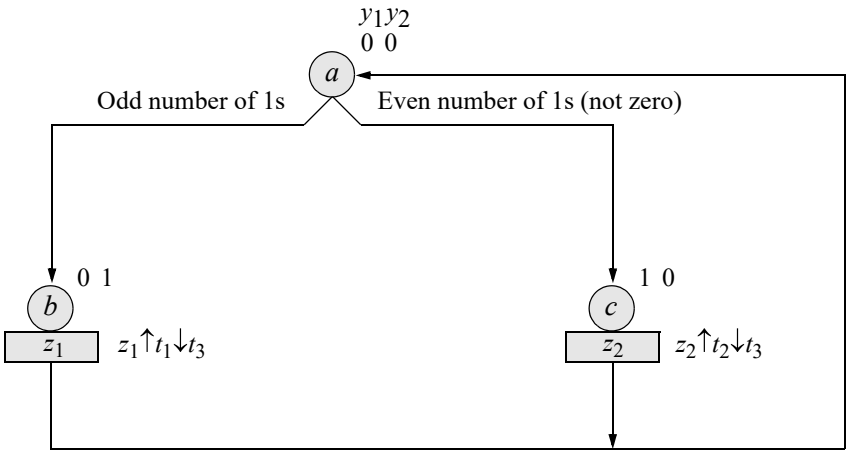


Figure 8.29 State diagram for the Moore machine of Example 8.6.

Table 8.1 Table Showing Words Containing an Odd or Even Number of 1s

x_1	x_2	x_3	x_4	
0	0	0	0	
	0	0	1	Odd number of 1s
0	0	1	0	
0	0	1	1	
	1	0	0	
0	1	0	1	
0	1	1	0	
	1	1	1	
	0	0	0	
1	0	0	1	
1	0	1	0	
	0	1	1	
1	1	0	0	
	1	0	1	
	1	1	0	
1	1	1	1	

$$\begin{aligned}\text{Odd} &= x_1'x_2'x_3'x_4 + x_1'x_2'x_3x_4' + x_1'x_2x_3'x_4' + x_1'x_2x_3x_4 + \\ &\quad x_1x_2'x_3'x_4' + x_1x_2'x_3x_4 + x_1x_2x_3'x_4 + x_1x_2x_3x_4' \\ &= x_1'x_2'(x_3 \oplus x_4) + x_1'x_2(x_3 \oplus x_4)' + x_1x_2(x_3 \oplus x_4) + x_1x_2'(x_3 \oplus x_4)' \\ &= (x_3 \oplus x_4)(x_1 \oplus x_2)' + (x_3 \oplus x_4)'(x_1 \oplus x_2) \\ &= (x_1 \oplus x_2) \oplus (x_3 \oplus x_4) \end{aligned} \tag{8.5}$$

$$\begin{aligned}\text{Even} &= x_1'x_2'x_3x_4 + x_1'x_2x_3'x_4 + x_1'x_2x_3x_4' + x_1x_2'x_3'x_4 + \\ &\quad x_1x_2'x_3x_4' + x_1x_2x_3'x_4' + x_1x_2x_3x_4 \\ &= x_1'x_2(x_3 \oplus x_4) + x_1x_2(x_3 \oplus x_4)' + x_1x_2'(x_3 \oplus x_4) + x_1'x_2'x_3x_4 \\ &= (x_1 \oplus x_2)(x_3 \oplus x_4) + x_1x_2(x_3 \oplus x_4)' + x_1'x_2'x_3x_4 \end{aligned} \tag{8.6}$$

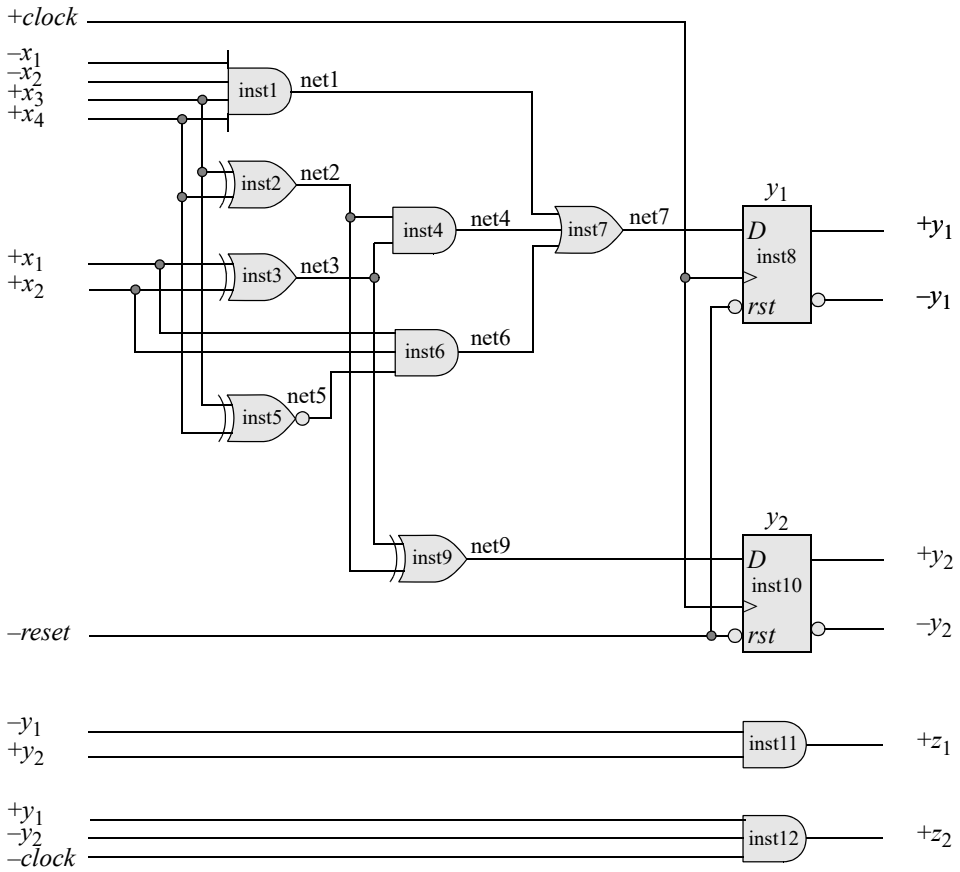


Figure 8.30 Logic diagram for the Moore machine of Example 8.6.

```
//structural moore synchronous sequential machine
module moore_ssm19a (rst_n, clk, x, y, z1, z2);

input rst_n, clk;
input [1:4] x;
output [1:2] y;
output z1, z2;

wire rst_n, clk;
wire [1:4] x;
wire [1:2] y;
wire z1, z2;

//continued on next page
```

Figure 8.31 Structural module for the Moore machine of Example 8.6.

```
//define internal wires
wire  net1, net2, net3, net4, net5,
      net6, net7, net9;

//instantiate the logic for flip-flop y[1]
and4_df inst1 (
   .x1(~x[1]),
   .x2(~x[2]),
   .x3(x[3]),
   .x4(x[4]),
   .z1(net1)
   );

xor2_df inst2 (
   .x1(x[3]),
   .x2(x[4]),
   .z1(net2)
   );

xor2_df inst3 (
   .x1(x[1]),
   .x2(x[2]),
   .z1(net3)
   );

and2_df inst4 (
   .x1(net2),
   .x2(net3),
   .z1(net4)
   );

xnor2_df inst5 (
   .x1(x[3]),
   .x2(x[4]),
   .z1(net5)
   );

and3_df inst6 (
   .x1(x[1]),
   .x2(x[2]),
   .x3(net5),
   .z1(net6)
   );

//continued on next page
```

Figure 8.31 (Continued)

```
or3_df inst7 (
   .x1(net1),
   .x2(net4),
   .x3(net6),
   .z1(net7)
   );

d_ff_bh inst8 (
   .rst_n(rst_n),
   .clk(clk),
   .d(net7),
   .q(y[1])
   );

//instantiate the logic for flip-flop y[2]
xor2_df inst9 (
   .x1(net3),
   .x2(net2),
   .z1(net9)
   );

d_ff_bh inst10 (
   .rst_n(rst_n),
   .clk(clk),
   .d(net9),
   .q(y[2])
   );

//instantiate the logic for outputs z1 and z2
and2_df inst11 (
   .x1(~y[1]),
   .x2(y[2]),
   .z1(z1)
   );

and3_df inst12 (
   .x1(y[1]),
   .x2(~y[2]),
   .x3(~clk),
   .z1(z2)
   );

endmodule
```

Figure 8.31 (Continued)

```
//test bench for moore synchronous sequential machine 19a
module moore_ssm19a_tb;

reg rst_n, clk;
reg [1:4] x;
wire [1:2] y;
wire z1, z2;

//display variables
initial
$monitor ("x = %b, state = %b, z1z2 = %b",
            x, y, {z1, z2});

//define clock
initial
begin
   clk = 1'b0;
   forever
      #10clk = ~clk;
end

//define input sequence
initial
begin
   #0 rst_n = 1'b0;
      x = 4'b0000;

   #5 rst_n = 1'b1;

      x = 4'b0001;
      @ (posedge clk)   //go to state_b (01); assert z1

      x = 4'b0000;
      @ (posedge clk)   //go to state_a (00)

      x = 4'b1010;
      @ (posedge clk)   //go to state_c (10); assert z2

      x = 4'b0000;
      @ (posedge clk)   //go to state_a (00)

      x = 4'b0111;
      @ (posedge clk)   //go to state_b (01); assert z1

      x = 4'b0000;
      @ (posedge clk)   //go to state_a (00)
//continued on next page
```

Figure 8.32 Test bench module for the Moore machine of Example 8.6.

```
      x = 4'b1111;
      @ (posedge clk)    //go to state_c (10); assert z2

      x = 4'b0000;
      @ (posedge clk)    //go to state_a (00)
   #10   $stop;
end

moore_ssm19a inst1 (    //instantiate the module
   .rst_n(rst_n),
   .clk(clk),
   .x(x),
   .y(y),
   .z1(z1),
   .z2(z2)
   );
endmodule
```

Figure 8.32 (Continued)

```
x = 0000, state = 00, z1z2 = 00 | x = 0111, state = 00, z1z2 = 00
x = 0001, state = 00, z1z2 = 00 | x = 0000, state = 01, z1z2 = 10
x = 0000, state = 01, z1z2 = 10 | x = 1111, state = 00, z1z2 = 00
x = 1010, state = 00, z1z2 = 00 | x = 0000, state = 10, z1z2 = 00
x = 0000, state = 10, z1z2 = 00 | x = 0000, state = 10, z1z2 = 01
x = 0000, state = 10, z1z2 = 01 | x = 0000, state = 00, z1z2 = 00
```

Figure 8.33 Outputs for the Moore machine of Example 8.6.

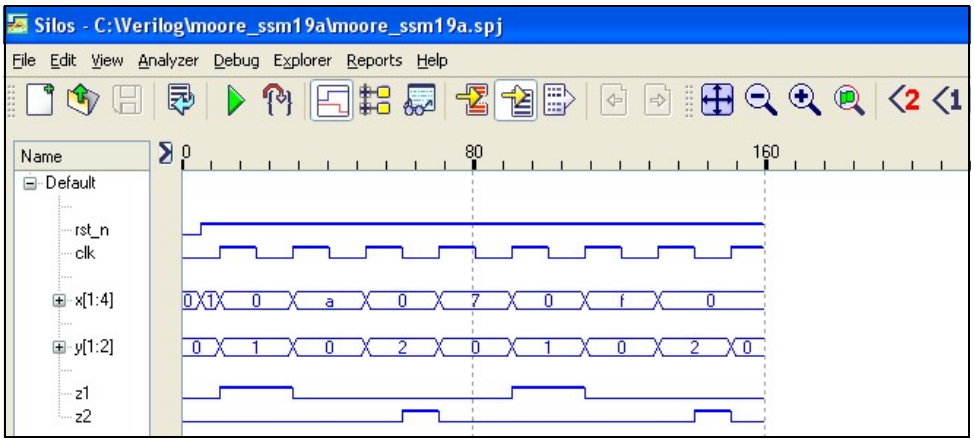


Figure 8.34 Waveforms for the Moore machine of Example 8.6.

8.2 Asynchronous Sequential Machines

Asynchronous sequential machines are another class of finite-state machines, where state changes occur on the application of input signals only — there is no machine clock. Like synchronous machines, the outputs of asynchronous machines are a function of either the present state only, or of the present state and the present inputs, corresponding to Moore and Mealy machines, respectively.

This section presents a variety of asynchronous sequential machines that are designed using Verilog HDL. These state machines were designed in Chapter 7 using traditional design methodologies and are presented in this section as a means to compare the two design techniques.

Example 8.7 An asynchronous sequential machine will be designed that has one input x_1 and two outputs z_1 and z_2. Output z_1 is asserted for the duration of every second x_1 pulse. Output z_2 is asserted for the duration of every second z_1 pulse. The outputs are to respond as fast as possible to changes in the input variable. A representative timing diagram is shown in Figure 8.35.

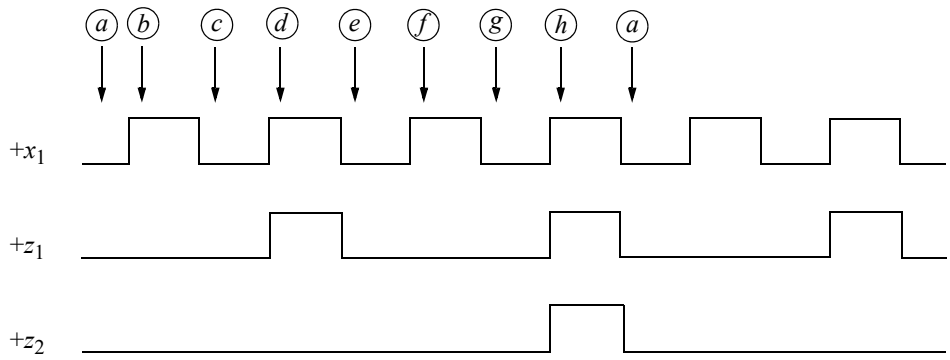


Figure 8.35 Representative timing diagram for the asynchronous sequential machine of Example 8.7.

This example replicates Example 7.13. Since this section emphasizes the design of asynchronous sequential machines using Verilog HDL; therefore, only the excitation equations, the output equations, and the logic diagram are presented. Refer to Example 7.13 for a detailed presentation of the steps required to design this asynchronous machine.

The excitation equations for Y_{1e}, Y_{2e}, and Y_{3e} are shown in Equation 8.7; the output equations for z_1 and z_2 are shown in Equation 8.8 and represent the fastest possible output changes. The logic diagram is shown in Figure 8.36, which depicts the logic primitives and associated net names. The machine will be designed using the continuous assignment statement **assign** for dataflow modeling.

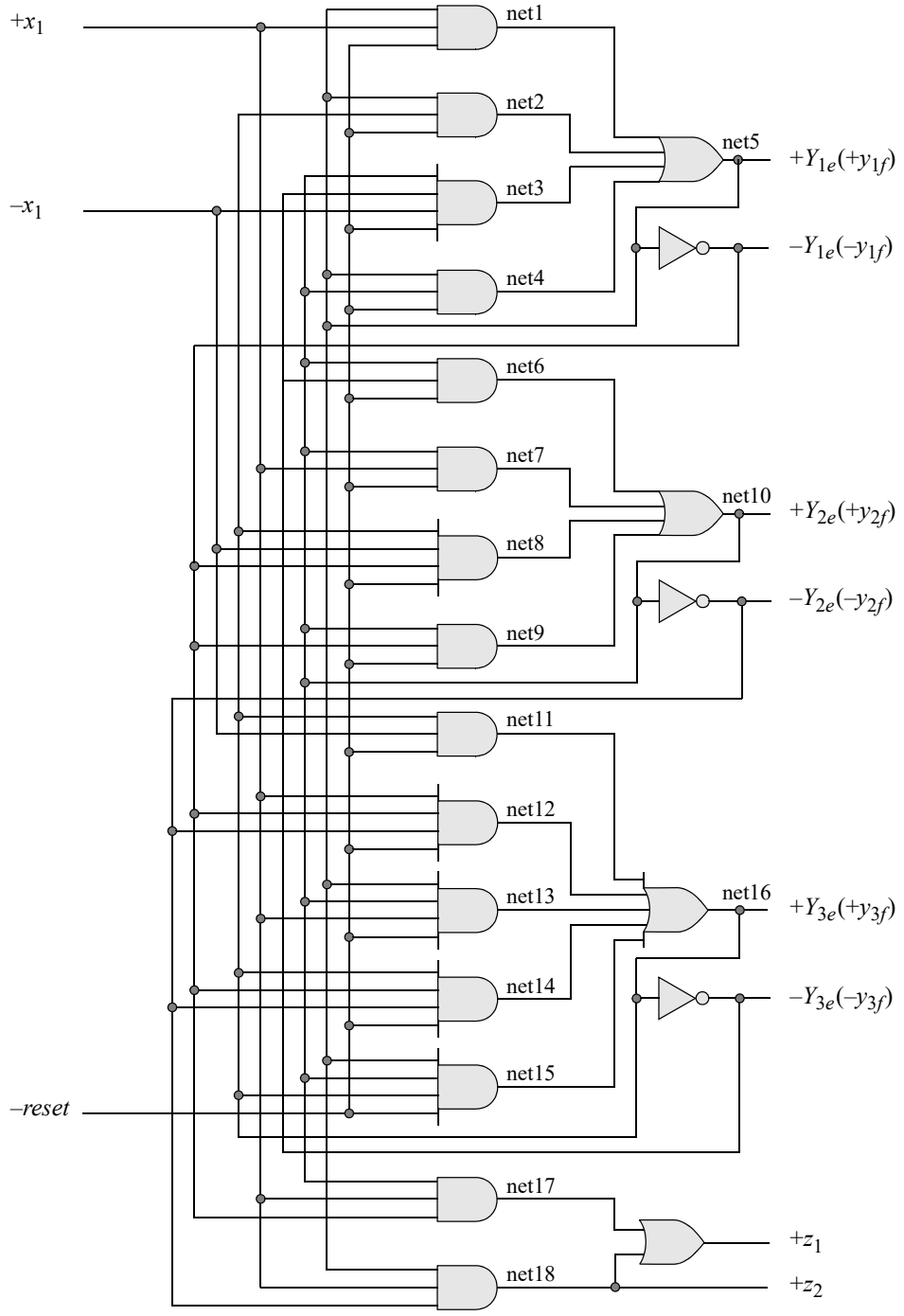


Figure 8.36 Logic diagram for the asynchronous sequential machine of Example 8.7.

The dataflow module and test bench are shown in Figure 8.37 and Figure 8.38, respectively. The test bench sequences the machine through the paths illustrated in the primitive flow table. The outputs and waveforms are shown in Figure 8.39 and Figure 8.40, respectively.

```
//dataflow asynchronous sequential machine
module asm13 (rst_n, x1, z1, z2);

input rst_n, x1;
output z1, z2;

//define internal nets
wire  net1, net2, net3, net4, net5, net6,
      net7, net8, net9, net10, net11, net12,
      net13, net14, net15, net16, net17, net18;

//design the logic for excitation variable Y1e
assign   net1  = rst_n & net5 & x1,
         net2  = rst_n & net5 & net16,
         net3  = rst_n & net10 & ~net16 & ~x1,
         net4  = rst_n & net5 & net10,
         net5  = net1 | net2 | net3 | net4;

//design the logic for excitation variable Y2e
assign   net6  = rst_n & net10 & ~net16,
         net7  = rst_n & net10 & x1,
         net8  = rst_n & net16 & ~x1 & ~net5,
         net9  = rst_n & net10 & ~net5,
         net10 = net6 | net7 | net8 | net9;

//design the logic for excitation variable Y3e
assign   net11 = rst_n & net16 & ~x1,
         net12 = rst_n & x1 & ~net5 & ~net10,
         net13 = rst_n & net5 & net10 & x1,
         net14 = rst_n & net16 & ~net5 & ~net10,
         net15 = rst_n & net5 & net10 & net16,
         net16 = net11 | net12 | net13 | net14 | net15;

//design the logic for output z1
assign   net17 = net10 & x1 & ~net5,
         net18 = net5 & x1 & ~net10,
         z1    = net17 | net18;

//design the logic for output z2
assign   z2    = net18;

endmodule
```

Figure 8.37 Dataflow module for the asynchronous sequential machine of Example 8.7.

```
//test bench for asynchronous sequential machine
module asm13_tb;

reg rst_n, x1;
wire z1, z2;

//display variables
initial
$monitor ("x1 = %b, z1 = %b, z2 = %b", x1, z1, z2);

//apply input values
initial
begin
   #0    rst_n = 1'b0;
         x1 = 1'b0;  //reset to state_a
   #5    rst_n = 1'b1;

   #10   x1 = 1'b1;  //go to state_b
   #10   x1 = 1'b0;  //go to state_c
   #10   x1 = 1'b1;  //go to state_d; assert z1
   #10   x1 = 1'b0;  //go to state_e
   #10   x1 = 1'b1;  //go to state_f
   #10   x1 = 1'b0;  //go to state_g
   #10   x1 = 1'b1;  //go to state_h; assert z1 and z2
   #10   x1 = 1'b0;  //go to state_a
   #10   x1 = 1'b1;  //go to state_b
   #10   $stop;
end

//instantiate the module into the test bench
asm13 inst1 (
   .rst_n(rst_n),
   .x1(x1),
   .z1(z1),
   .z2(z2)
   );
endmodule
```

Figure 8.38 Test bench for the asynchronous sequential machine of Example 8.7.

```
x1 = 0, z1 = 0, z2 = 0          x1 = 1, z1 = 0, z2 = 0
x1 = 1, z1 = 0, z2 = 0          x1 = 0, z1 = 0, z2 = 0
x1 = 0, z1 = 0, z2 = 0          x1 = 1, z1 = 1, z2 = 1
x1 = 1, z1 = 1, z2 = 0          x1 = 0, z1 = 0, z2 = 0
x1 = 0, z1 = 0, z2 = 0          x1 = 1, z1 = 0, z2 = 0
```

Figure 8.39 Outputs for the asynchronous sequential machine of Example 8.7.

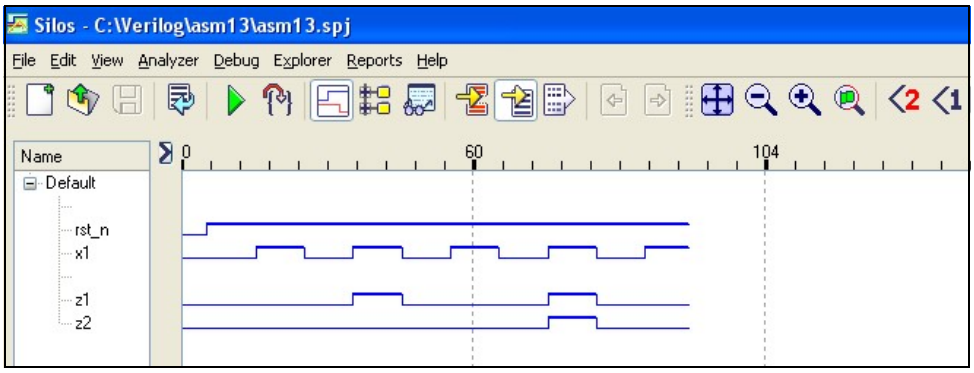


Figure 8.40 Waveforms for the asynchronous sequential machine of Example 8.7.

The next three examples will design the same asynchronous sequential machine using three different modeling techniques: dataflow modeling, behavioral modeling, and structural modeling. These three examples will offer a comparative study of three different modeling constructs for the same machine. The same test bench will be used for all three examples and will produce similar outputs and waveforms.

Example 8.8 An asynchronous sequential machine will be designed using dataflow modeling that has two inputs x_1 and x_2 and one output z_1. Output z_1 will be asserted coincident with the assertion of the first x_2 pulse and will remain active for the duration of the first x_2 pulse. The output will be asserted only if the assertion of x_1 precedes the assertion of x_2. Input x_1 will not become deasserted while x_2 is asserted. The λ output logic must have a minimal number of logic gates. A representative timing diagram is shown in Figure 8.41.

The primitive flow table is shown in Figure 8.42 in which there are no equivalent states. The merger diagram, the merged flow table, and the transition diagram are shown in Figure 8.43, Figure 8.44, and Figure 8.45, respectively.

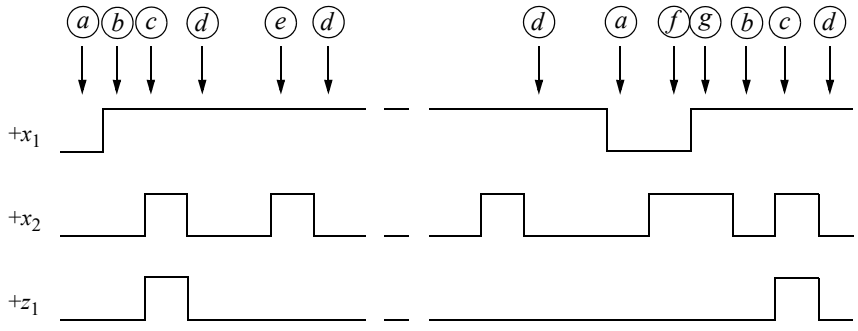


Figure 8.41 An asynchronous sequential machine for Example 8.8.

x_1x_2 00	01	11	10	z_1
(a)	f	–	b	0
a	–	c	(b)	0
–	–	(c)	d	1
a	–	e	(d)	0
–	–	(e)	d	0
a	(f)	g	–	0
–	–	(g)	b	0

Figure 8.42 Primitive flow table for the asynchronous sequential machine of Example 8.8.

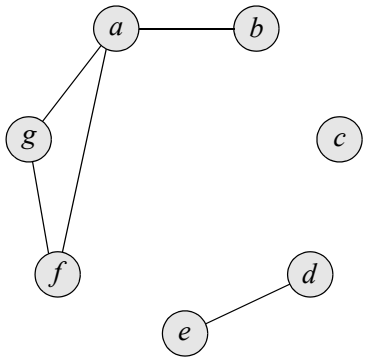


Figure 8.43 Merger diagram for the asynchronous sequential machine of Example 8.8.

				x_1x_2 00	01	11	10
1	(a)	(f)	(g)	(a)	(f)	(g)	b
2			(b)	a	–	c	(b)
3			(c)	–	–	(c)	d
4		(d)	(e)	a	–	(e)	(d)

Figure 8.44 Merged flow table for the asynchronous machine of Example 8.8.

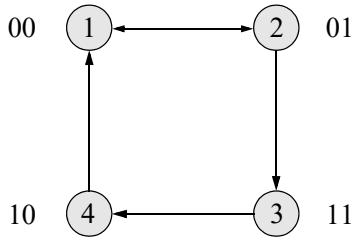


Figure 8.45 Transition diagram for the machine of Example 8.8.

The combined excitation map for Y_{1e} and Y_{2e} is shown in Figure 8.46 and the individual excitation maps are shown in Figure 8.47. The equations for Y_{1e} and Y_{2e} are shown in Equation 8.9. The output map for z_1 is shown in Figure 8.48 and the equation for z_1 is shown in Equation 8.10. The logic diagram — derived from the excitation and output equations — is shown in Figure 8.49.

$y_{1f}\,y_{2f}$ \ x_1x_2	0 0	0 1	1 1	1 0
0 0	(00) *a*	(00) *f*	(00) *g*	01
0 1	00	–	11	(01) *b*
1 1	–	–	(11) *c*	10
1 0	00	–	(10) *e*	(10) *d*

$Y_{1e}\,Y_{2e}$

Figure 8.46 Combined excitation map for the asynchronous sequential machine of Example 8.8.

$y_{1f}\,y_{2f}$ \ x_1x_2	0 0	0 1	1 1	1 0
0 0	0	0	0	0
0 1	0	–	1	0
1 1	–	–	1	1
1 0	0	–	1	1

Y_{1e}

$y_{1f}\,y_{2f}$ \ x_1x_2	0 0	0 1	1 1	1 0
0 0	0	0	0	1
0 1	0	–	1	1
1 1	–	–	1	0
1 0	0	–	0	0

Y_{2e}

Figure 8.47 Individual excitation maps for the machine of Example 8.8.

$$Y_{1e} = y_{1f}x_1 + y_{2f}x_2$$
$$Y_{2e} = y_{2f}x_2 + y_{1f}'x_1x_2' + y_{1f}'y_{2f}x_1 \tag{8.9}$$

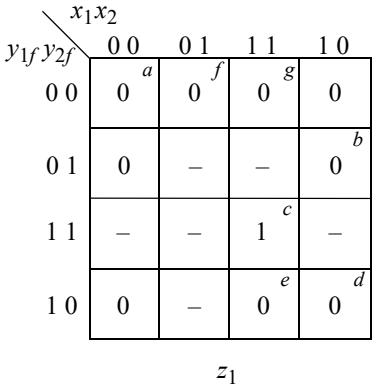


Figure 8.48 Output map for the asynchronous sequential machine of Example 8.8.

$$z_1 = y_{1f}y_{2f} \tag{8.10}$$

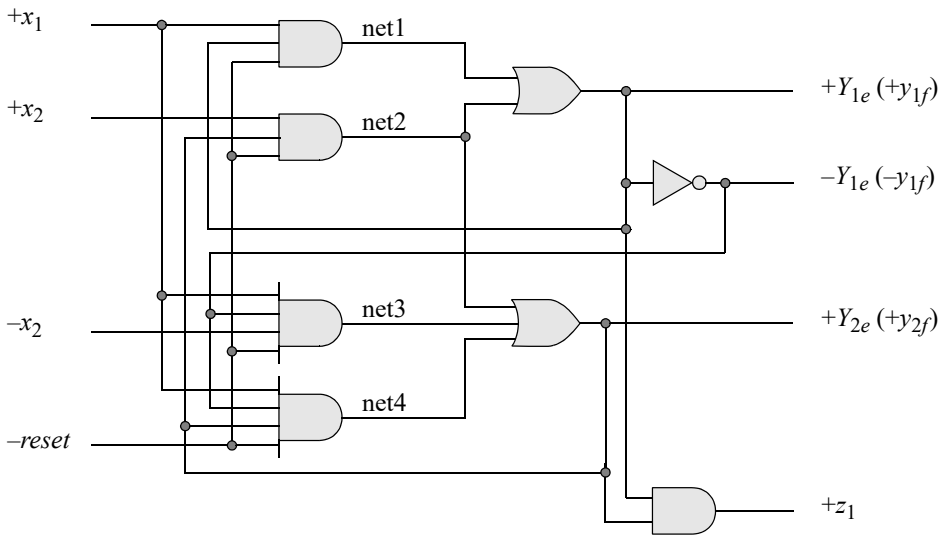


Figure 8.49 Logic diagram for the asynchronous sequential machine of Example 8.8.

The dataflow design module is shown in Figure 8.50 using the **assign** statement for all the logic primitive gates. The test bench — shown in Figure 8.51 — takes the machine through the state transition sequences shown in the primitive flow table and the timing diagram. The outputs are shown in Figure 8.52 and the waveforms in Figure 8.53.

```
//dataflow asynchronous sequential machine 16a
module asm16a (rst_n, x1, x2, y1e, y2e, z1);

input rst_n, x1, x2;
output y1e, y2e, z1;

//define internal nets
wire net1, net2, net3, net4;

//design the logic for excitation variable Y1e
assign   net1 = x1 & y1e & rst_n,
         net2 = x2 & y2e & rst_n,
         y1e = net1 | net2;

//design the logic for excitation variable Y2e
assign   net3 = x1 & ~x2 & ~y1e & rst_n,
         net4 = x1 & y2e & ~y1e & rst_n,
         y2e = net2 | net3 | net4;

//design the logic for output z1
assign   z1 = y1e & y2e;

endmodule
```

Figure 8.50 Dataflow module for the asynchronous sequential machine of Example 8.8.

```
//test bench for asynchronous sequential machine 16a
//use the primitive flow table or the timing diagram to follow
//the input sequence for the output and the waveforms
module asm16a_tb;

reg rst_n, x1, x2;
wire y1e, y2e, z1;

//display variables
initial
$monitor ("x1x2 = %b, state = %b, z1 = %b",  //next page
```

Figure 8.51 Test bench for the asynchronous sequential machine of Example 8.8.

```
//define input vectors
initial
begin
   #0    rst_n = 1'b0;
         x1 = 1'b0;
         x2 = 1'b0;

   #5    rst_n = 1'b1;

   #10   x1=1'b1;    x2=1'b0;    //go to state_b
   #10   x1=1'b1;    x2=1'b1;    //go to state_c; assert z1
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b1;    x2=1'b1;    //go to state_e
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b0;    x2=1'b0;    //go to state_a
   #10   x1=1'b0;    x2=1'b1;    //go to state_f
   #10   x1=1'b1;    x2=1'b1;    //go to state_g
   #10   x1=1'b1;    x2=1'b0;    //go to state_b
   #10   x1=1'b1;    x2=1'b1;    //go to state_c; assert z1
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b0;    x2=1'b0;    //go to state_a

   #10   $stop;
end

//instantiate the module into the test bench
asm16a inst1 (
   .rst_n(rst_n),
   .x1(x1),
   .x2(x2),
   .y1e(y1e),
   .y2e(y2e),
   .z1(z1)
   );
endmodule
```

Figure 8.51 (Continued)

```
x1x2 = 00, state = 00, z1 = 0 | x1x2 = 01, state = 00, z1 = 0
x1x2 = 10, state = 01, z1 = 0 | x1x2 = 11, state = 00, z1 = 0
x1x2 = 11, state = 11, z1 = 1 | x1x2 = 10, state = 01, z1 = 0
x1x2 = 10, state = 10, z1 = 0 | x1x2 = 11, state = 11, z1 = 1
x1x2 = 11, state = 10, z1 = 0 | x1x2 = 10, state = 10, z1 = 0
x1x2 = 10, state = 10, z1 = 0 | x1x2 = 00, state = 00, z1 = 0
x1x2 = 00, state = 00, z1 = 0 |
```

Figure 8.52 Outputs for the asynchronous sequential machine of Example 8.8.

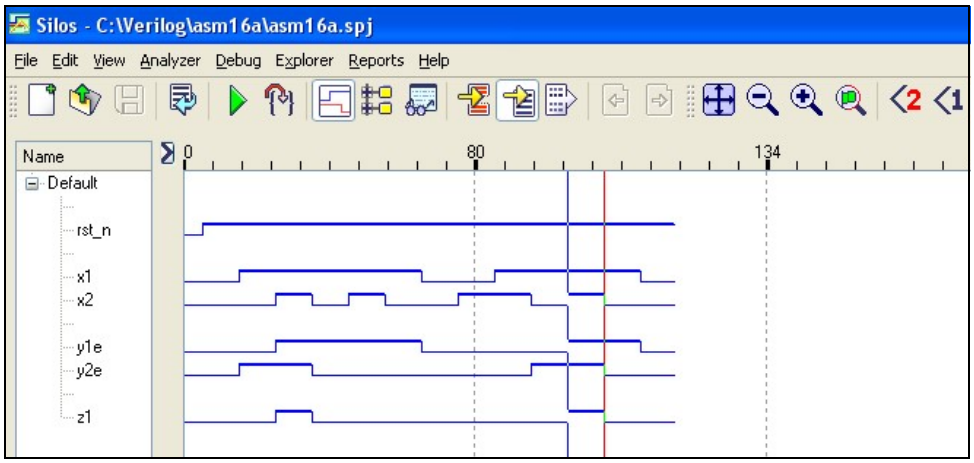


Figure 8.53 Waveforms for the asynchronous sequential machine of Example 8.8.

Example 8.9 This example repeats Example 8.8, but uses behavioral modeling to design the asynchronous sequential machine. The primitive flow table is duplicated in Figure 8.54 and shows the state code assignments that will be used in the behavioral module.

x_1x_2 00	01	11	10	z_1	
(a)	f	–	b	0	(a) = 000
a	–	c	(b)	0	(b) = 001
–	–	(c)	d	1	(c) = 011
a	–	e	(d)	0	(d) = 010
–	–	(e)	d	0	(e) = 110
a	(f)	g	–	0	(f) = 111
–	–	(g)	b	0	(g) = 101

Figure 8.54 Primitive flow table for the asynchronous sequential machine of Example 8.9.

There is no need to obtain the merger diagram, the merged flow table, the transition diagram, the excitation maps, the output map, or the logic diagram. The behavior of the machine is specified and the logic design is accomplished by the synthesis tool.

The behavioral module is shown in Figure 8.55, in which state codes are assigned using the **parameter** keyword. The output is defined for each state and the **case** statement is used to determine the next state. The test bench is shown in Figure 8.56 — the primitive flow table can be used with the test bench to follow the machine through all of the state transition sequences.

The outputs are shown in Figure 8.57 and indicate the 3-bit state codes for the storage elements. The waveforms are shown in Figure 8.58 and display the same information as the waveforms of Figure 8.53, but in a slightly different form.

```
//behavioral asynchronous sequential machine
module asm16 (rst_n, x1, x2, y, z1);

input rst_n, x1, x2;
output [1:3] y;
output z1;

wire rst_n, x1, x2;
reg [1:3] y, next_state;
reg z1;

//assign state codes
parameter   state_a = 3'b000,
            state_b = 3'b001,
            state_c = 3'b011,
            state_d = 3'b010,
            state_e = 3'b110,
            state_f = 3'b111,
            state_g = 3'b101;

//latch next state
always @ (x1 or x2 or rst_n)
begin
   if (~rst_n)
      y <= state_a;
   else
      y <= next_state;
end

//continued on next page
```

Figure 8.55 Behavioral module for the asynchronous sequential machine of Example 8.9.

```
//define output z1
always @ (x1 or x2 or y)
begin
   if (y == state_a)
      z1 = 1'b0;

   if (y == state_b)
      z1 = 1'b0;

   if (y == state_c)
      z1 = 1'b1;

   if (y == state_d)
      z1 = 1'b0;

   if (y == state_e)
      z1 = 1'b0;

   if (y == state_f)
      z1 = 1'b0;

   if (y == state_g)
      z1 = 1'b0;
end

//determine next state
always @ (x1 or x2)
begin
   case (y)
      state_a:    //== is logical equality; && is logical AND
         if ((x1==1'b1) && (x2==1'b0))
            next_state = state_b;
         else if ((x1==1'b0) && (x2==1'b1))
            next_state = state_f;
         else
            next_state = state_a;

      state_b:
         if ((x1==1'b0) && (x2==1'b0))
            next_state = state_a;
         else if ((x1==1'b1) && (x2==1'b1))
            next_state = state_c;
         else
            next_state = state_b;

//continued on next page
```

Figure 8.55 (Continued)

```
      state_c:
         if ((x1==1'b1) && (x2==1'b0))
            next_state = state_d;
         else
            next_state = state_c;


      state_d:
         if ((x1==1'b0) && (x2==1'b0))
            next_state = state_a;
         else if ((x1==1'b1) && (x2==1'b1))
            next_state = state_e;
         else
            next_state = state_d;


      state_e:
         if ((x1==1'b1) && (x2==1'b0))
            next_state = state_d;
         else
            next_state = state_e;


      state_f:
         if ((x1==1'b0) && (x2==1'b0))
            next_state = state_a;
         else if ((x1==1'b1) && (x2==1'b1))
            next_state = state_g;
         else
            next_state = state_f;


      state_g:
         if ((x1==1'b1) && (x2==1'b0))
            next_state = state_b;
         else
            next_state = state_g;

      default: next_state = state_a;

   endcase
end

endmodule
```

Figure 8.55 (Continued)

```
//test bench for asynchronous sequential machine
module asm16_tb;

reg rst_n, x1, x2;
wire [1:3] y;
wire z1;

//display variables
initial
$monitor ("x1x2 = %b, y = %b, z1 = %b",
            {x1, x2}, y, z1);

//define input vectors
initial
begin
   #0    rst_n = 1'b0;
         x1 = 1'b0;
         x2 = 1'b0;

   #5    rst_n = 1'b1;

   #10   x1=1'b1;    x2=1'b0;    //go to state_b
   #10   x1=1'b1;    x2=1'b1;    //go to state_c; assert z1
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b1;    x2=1'b1;    //go to state_e
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b0;    x2=1'b0;    //go to state_a
   #10   x1=1'b0;    x2=1'b1;    //go to state_f
   #10   x1=1'b1;    x2=1'b1;    //go to state_g
   #10   x1=1'b1;    x2=1'b0;    //go to state_b
   #10   x1=1'b1;    x2=1'b1;    //go to state_c; assert z1
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b0;    x2=1'b0;    //go to state_a

   #10   $stop;
end

//instantiate the module into the test bench
asm16 inst1 (
   .rst_n(rst_n),
   .x1(x1),
   .x2(x2),
   .y(y),
   .z1(z1)
   );

endmodule
```

Figure 8.56 Test bench for the asynchronous sequential machine of Example 8.9.

```
x1x2 = 00, y = 000, z1 = 0
x1x2 = 10, y = 001, z1 = 0
x1x2 = 11, y = 011, z1 = 1
x1x2 = 10, y = 010, z1 = 0
x1x2 = 11, y = 110, z1 = 0
x1x2 = 10, y = 010, z1 = 0
x1x2 = 00, y = 000, z1 = 0
x1x2 = 01, y = 111, z1 = 0
x1x2 = 11, y = 101, z1 = 0
x1x2 = 10, y = 001, z1 = 0
x1x2 = 11, y = 011, z1 = 1
x1x2 = 10, y = 010, z1 = 0
x1x2 = 00, y = 000, z1 = 0
```

Figure 8.57 Outputs for the asynchronous sequential machine of Example 8.9.

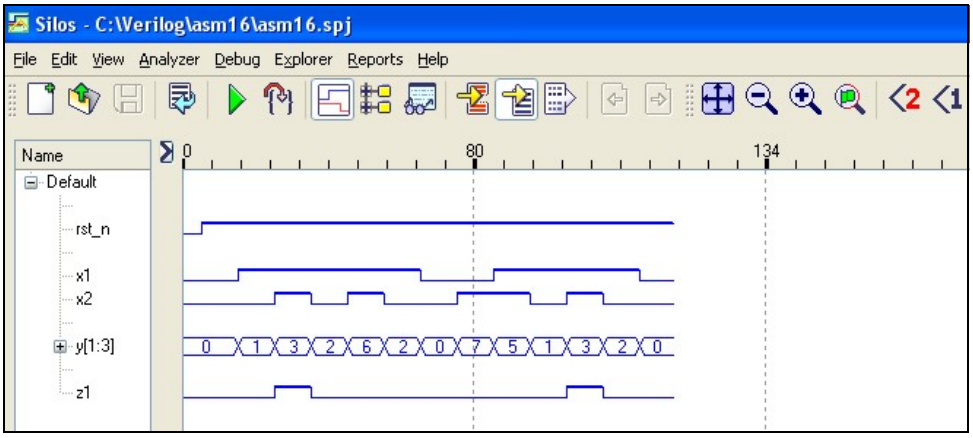


Figure 8.58 Waveforms for the asynchronous sequential machine of Example 8.9.

Example 8.10 This example repeats the asynchronous sequential machine of Example 8.8, but uses structural modeling in the implementation. The primitive flow table is reproduced in Figure 8.59 so that the operation of the machine can be easily followed as it proceeds through the state transition sequences. The various design steps will not be reproduced — they can be examined by referring to Example 8.8.

The excitation equations for Y_{1e} and Y_{2e} and the output equation for z_1 are replicated in Equation 8.9 and Equation 8.10, respectively. The logic diagram is shown in Figure 8.60 and displays the instantiation names and the net names which will be used in the structural module.

x_1x_2 00	01	11	10	z_1
(a)	f	–	b	0
a	–	c	(b)	0
–	–	(c)	d	1
a	–	e	(d)	0
–	–	(e)	d	0
a	(f)	g	–	0
–	–	(g)	b	0

Figure 8.59 Primitive flow table for the asynchronous sequential machine of Example 8.10.

$$Y_{1e} = y_{1f}x_1 + y_{2f}x_2$$
$$Y_{2e} = y_{2f}x_2 + y_{1f}'x_1x_2' + y_{1f}'y_{2f}x_1 \tag{8.11}$$

$$z_1 = y_{1f}y_{2f} \tag{8.12}$$

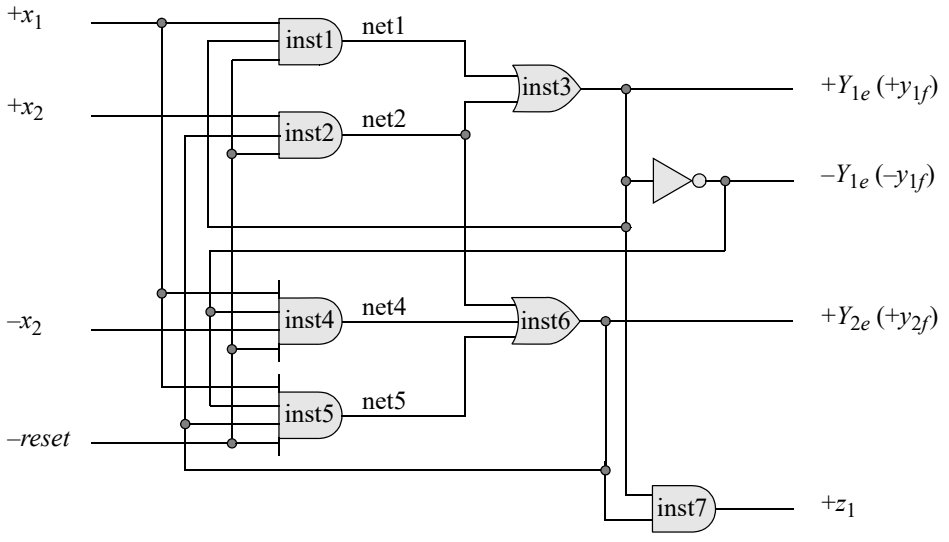


Figure 8.60 Logic diagram for the asynchronous machine of Example 8.10.

The structural module is shown in Figure 8.61 using dataflow logic primitives that were previously designed. The instantiation names and net names correlate to the instantiation names and net names indicated in the logic diagram. The test bench module is shown in Figure 8.62. The input sequence takes the machine through the state transitions shown in the primitive flow table. The outputs and waveforms are shown in Figure 8.63 and Figure 8.64, respectively.

```
//structural asynchronous sequential machine 16b
module asm16b (rst_n, x1, x2, y1e, y2e, z1);

input rst_n, x1, x2;
output y1e, y2e;
output z1;

//define internal nets
wire net1, net2, net4, net5;

//instantiate the logic for latch y1e
and3_df inst1 (
   .x1(y1e),
   .x2(x1),
   .x3(rst_n),
   .z1(net1)
   );

and3_df inst2 (
   .x1(y2e),
   .x2(x2),
   .x3(rst_n),
   .z1(net2)
   );

or2_df inst3 (
   .x1(net1),
   .x2(net2),
   .z1(y1e)
   );

//instantiate the logic for latch y2e
and4_df inst4 (
   .x1(~y1e),
   .x2(x1),
   .x3(~x2),
   .x4(rst_n),
   .z1(net4)
   );             //continued on next page
```

Figure 8.61 Structural module for the asynchronous machine of Example 8.10.

```
and4_df inst5 (
   .x1(~y1e),
   .x2(x1),
   .x3(y2e),
   .x4(rst_n),
   .z1(net5)
   );

or3_df inst6 (
   .x1(net2),
   .x2(net4),
   .x3(net5),
   .z1(y2e)
   );

//instantiate the logic for output z1
and2_df inst7 (
   .x1(y1e),
   .x2(y2e),
   .z1(z1)
   );

endmodule
```

Figure 8.61 (Continued)

```
//test bench for asynchronous sequential machine 16b
//use the primitive flow table to follow the input sequence
module asm16b_tb;

reg rst_n, x1, x2;
wire y1e, y2e;
wire z1;

initial     //display variables
$monitor ("x1x2 = %b, state = %b, z1 = %b",
            {x1, x2}, {y1e, y2e}, z1);

initial     //define input vectors
begin
   #0    rst_n = 1'b0;
         x1 = 1'b0;
         x2 = 1'b0;
   #5    rst_n = 1'b1;
//continued on next page
```

Figure 8.62 Test bench for the asynchronous machine of Example 8.10.

```
   #10   x1=1'b1;    x2=1'b0;    //go to state_b
   #10   x1=1'b1;    x2=1'b1;    //go to state_c; assert z1
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b1;    x2=1'b1;    //go to state_e
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b0;    x2=1'b0;    //go to state_a
   #10   x1=1'b0;    x2=1'b1;    //go to state_f
   #10   x1=1'b1;    x2=1'b1;    //go to state_g
   #10   x1=1'b1;    x2=1'b0;    //go to state_b
   #10   x1=1'b1;    x2=1'b1;    //go to state_c; assert z1
   #10   x1=1'b1;    x2=1'b0;    //go to state_d
   #10   x1=1'b0;    x2=1'b0;    //go to state_a

   #10   $stop;
end

//instantiate the module into the test bench
asm16b inst1 (
   .rst_n(rst_n),
   .x1(x1),
   .x2(x2),
   .y1e(y1e),
   .y2e(y2e),
   .z1(z1)
   );

endmodule
```

Figure 8.62 (Continued)

```
x1x2 = 00, state = 00, z1 = 0
x1x2 = 10, state = 01, z1 = 0
x1x2 = 11, state = 11, z1 = 1
x1x2 = 10, state = 10, z1 = 0
x1x2 = 11, state = 10, z1 = 0
x1x2 = 10, state = 10, z1 = 0
x1x2 = 00, state = 00, z1 = 0
x1x2 = 01, state = 00, z1 = 0
x1x2 = 11, state = 00, z1 = 0
x1x2 = 10, state = 01, z1 = 0
x1x2 = 11, state = 11, z1 = 1
x1x2 = 10, state = 10, z1 = 0
x1x2 = 00, state = 00, z1 = 0
```

Figure 8.63 Outputs for the asynchronous machine of Example 8.10.

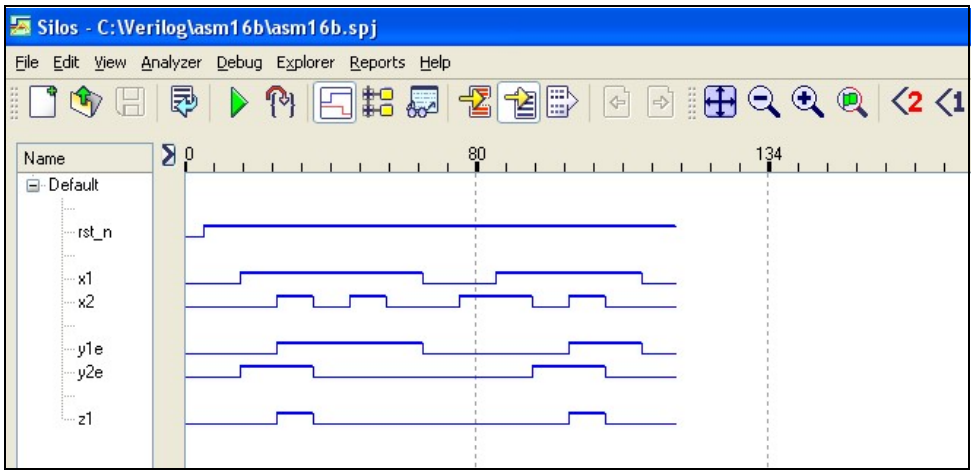


Figure 8.64 Waveforms for the asynchronous machine of Example 8.10.

8.3 Pulse-Mode Asynchronous Sequential Machines

Before beginning the design examples of pulse-mode asynchronous sequential machines, a brief review will be presented. As mentioned previously, each variable of the input alphabet is in the form of a pulse. The duration of each pulse is less than the propagation delay of the storage elements and the associated logic gates. Thus, an input pulse will initiate a state change, but the completion of the change will not take place until after the corresponding input has been deasserted. Multiple inputs cannot be active simultaneously. There is no separate clock input in pulse-mode machines.

Unlike a system clock, which has a specified frequency, the input pulses can occur randomly and more than one input pulse can generate an output. If the input pulse is of insufficient duration, then the storage elements may not be triggered and the machine will not sequence to the next state. If the pulse duration is too long, then the pulse will still be active when the machine changes from the present state $Y_{j(t)}$ to the next state $Y_{k(t+1)}$. The storage elements may then be triggered again and sequence the machine to an incorrect next state. If the time between consecutive pulses is too short, then the machine will be triggered while in an unstable condition, resulting in unpredictable behavior.

The pulse width restrictions that are dominant in pulse-mode sequential machines can be eliminated by including *D* flip-flops in the feedback path from the *SR* latches to the δ next-state logic. Providing edge-triggered *D* flip-flops as a constituent part of the implementation negates the requirement of precisely controlled input pulse durations. This is by far the most reliable means of synthesizing pulse-mode machines. The *SR* latches — in conjunction with the *D* flip-flops — form a master-slave configuration.

Example 8.11 A Mealy pulse-mode sequential machine will be designed which has two inputs x_1 and x_2 and one output z_1. Output z_1 is asserted coincident with every second x_2 pulse, if and only if the pair of x_2 pulses is immediately preceded by an x_1 pulse.

A combination of dataflow modeling — using the continuous assignment statement — and structural modeling will be used in the implementation. The storage elements will consist of *SR* latches and *D* flip-flops in a master-slave configuration. The design will be implemented with NOR logic for the *SR* latches and the logic primitives. A representative timing diagram is shown in Figure 8.65 and the state diagram is shown in Figure 8.66.

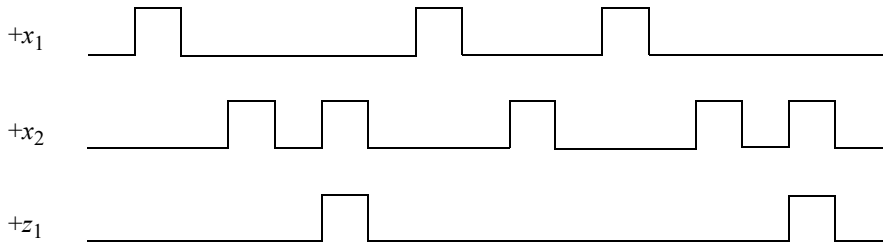


Figure 8.65 Representative timing diagram for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

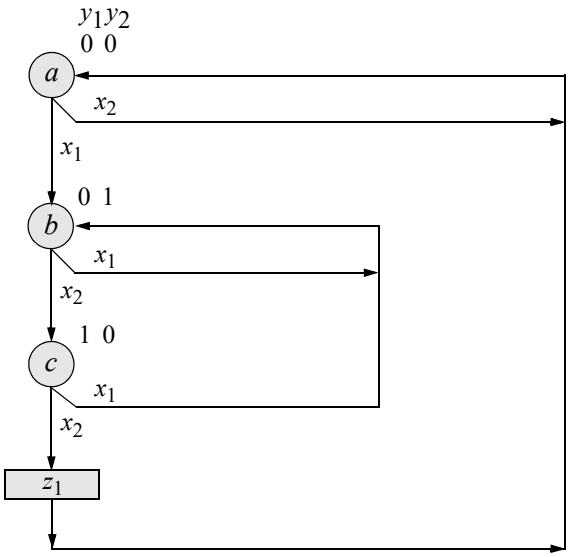


Figure 8.66 State diagram for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

The input maps and input equations are shown in Figure 8.67 and Equation 8.13, respectively. The output map and output equation are shown in Figure 8.68 and Equation 8.14, respectively. The logic diagram is shown in Figure 8.69, which instantiates positive-edge-triggered *D* flip-flops that were previously designed using behavioral modeling.

The mixed-design module — using dataflow and structural modeling is shown in Figure 8.70. The test bench is shown in Figure 8.71, which sequences the machine through all the state transitions shown in the state diagram. The outputs and waveforms are shown in Figure 8.72 and Figure 8.73, respectively.

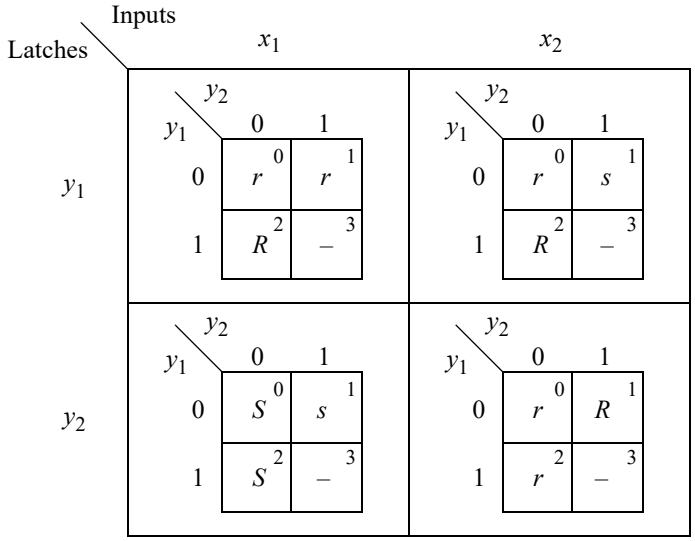


Figure 8.67 Input maps for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

$$
\begin{aligned}
S\mathrm{L}y_1 &= y_2 x_2 \\
R\mathrm{L}y_1 &= x_1 + y_1 x_2 \\
\\
S\mathrm{L}y_2 &= x_1 \\
R\mathrm{L}y_2 &= x_2
\end{aligned}
\tag{8.13}
$$

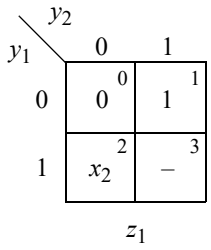


Figure 8.68 Output map for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

$$z_1 = y_1 x_2 \tag{8.14}$$

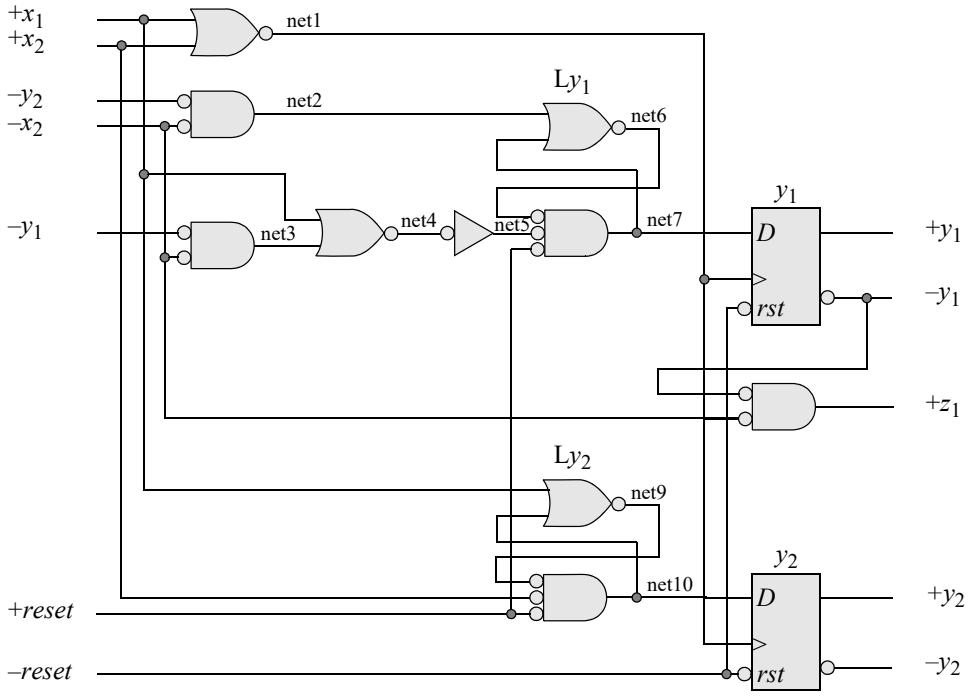


Figure 8.69 Logic diagram for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

```
//dataflow/structural mealy pulse-mode
//asynchronous sequential machine
module pm_asm7 (rst_n, rst, x1, x2, y1, y2, z1);

input rst_n, rst, x1, x2;
output y1, y2, z1;

//define internal nets
wire  net1, net2, net3, net4, net5,
      net6, net7, net9, net10;

//design the D flip-flop clock
assign net1 = ~(x1 | x2);

//design for latch Ly1
assign   net2 = ~(~y2 | ~x2),
         net3 = ~(~y1 | ~x2),
         net4 = ~(x1 | net3),
         net5 = ~net4,
         net6 = ~(net2 | net7),
         net7 = ~(net6 | net5 | rst);

//instantiate the D flip-flop for y1
d_ff_bh inst1 (
   .rst_n(rst_n),
   .clk(net1),
   .d(net7),
   .q(y1)
   );

//design for latch Ly2
assign   net9 = ~(x1 | net10),
         net10 = ~(net9 | x2 | rst);

//instantiate the D flip-flop for y2
d_ff_bh inst2 (
   .rst_n(rst_n),
   .clk(net1),
   .d(net10),
   .q(y2)
   );

//design the logic for output z1
assign   z1 = ~(~y1 | ~x2);

endmodule
```

Figure 8.70 Mixed-design module for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

```
//test bench for mealy pulse-mode asynchronous machine
module pm_asm7_tb;

reg rst_n, rst, x1, x2;
wire y1, y2, z1;

initial                     //display inputs and output
$monitor ("x1x2 = %b, state = %b, z1 = %b",
            {x1, x2}, {y1, y2}, z1);

initial                     //define input sequence
begin
   #0    rst_n = 1'b0;      //reset to state_a (00)
         rst = 1'b1;
         x1 = 1'b0;  x2 = 1'b0;
   #5    rst_n = 1'b1;
         rst = 1'b0;

   #10   x1 = 1'b0;  x2 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  //remain in state_a (00)

   #10   x1 = 1'b1;  x2 = 1'b0;
   #10   x1 = 1'b0;  x2 = 1'b0;  //go to state_b (01)

   #10   x1 = 1'b1;  x2 = 1'b0;
   #10   x1 = 1'b0;  x2 = 1'b0;  //remain in state_b (01)

   #10   x1 = 1'b0;  x2 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  //go to state_c (10)

   #10   x1 = 1'b0;  x2 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  //go to state_a (00); set z1

   #10   x1 = 1'b1;  x2 = 1'b0;
   #10   x1 = 1'b0;  x2 = 1'b0;  //go to state_b (01)

   #10   x1 = 1'b0;  x2 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  //go to state_c (10)

   #10   x1 = 1'b0;  x2 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  //go to state_a (00); set z1

   #10   x1 = 1'b0;  x2 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  //remain in state_a (00)
   #10   $stop;
end                  //continued on next page
```

Figure 8.71 Test bench for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

```
//instantiate the module into the test bench
pm_asm7 inst1 (
   .rst_n(rst_n),
   .rst(rst),
   .x1(x1),
   .x2(x2),
   .y1(y1),
   .y2(y2),
   .z1(z1)
   );
endmodule
```

Figure 8.71 (Continued)

```
x1x2 = 00, state = 00, z1 = 0
x1x2 = 01, state = 00, z1 = 0
x1x2 = 00, state = 00, z1 = 0
x1x2 = 10, state = 00, z1 = 0
x1x2 = 00, state = 01, z1 = 0
x1x2 = 10, state = 01, z1 = 0
x1x2 = 00, state = 01, z1 = 0
x1x2 = 01, state = 01, z1 = 0
x1x2 = 00, state = 10, z1 = 0
x1x2 = 01, state = 10, z1 = 1
```

```
x1x2 = 00, state = 00, z1 = 0
x1x2 = 10, state = 00, z1 = 0
x1x2 = 00, state = 01, z1 = 0
x1x2 = 01, state = 01, z1 = 0
x1x2 = 00, state = 10, z1 = 0
x1x2 = 01, state = 10, z1 = 1
x1x2 = 00, state = 00, z1 = 0
x1x2 = 01, state = 00, z1 = 0
x1x2 = 00, state = 00, z1 = 0
```

Figure 8.72 Outputs for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

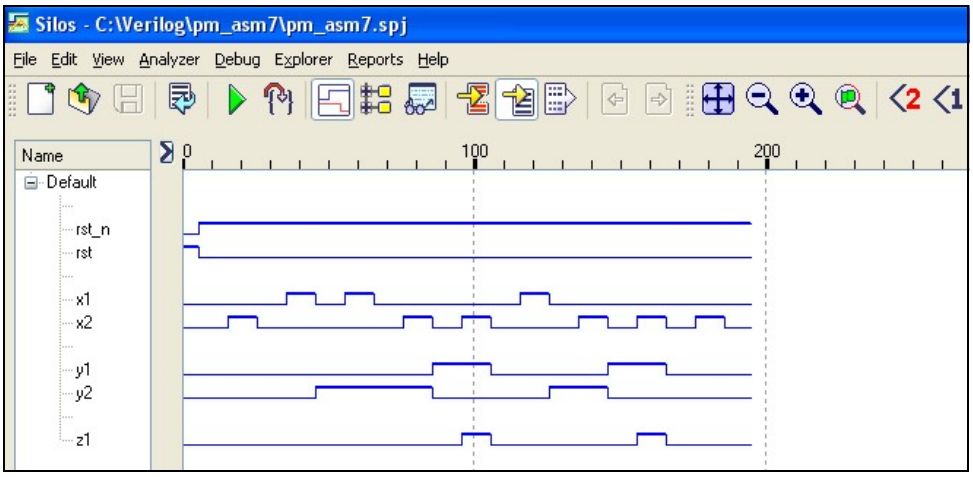


Figure 8.73 Waveforms for the Mealy pulse-mode asynchronous sequential machine of Example 8.11.

Example 8.12 A Moore pulse-mode asynchronous sequential machine will be designed which has three inputs x_1, x_2, and x_3 and one output z_1. Output z_1 will be asserted coincident with the assertion of the x_3 pulse if and only if the x_3 pulse was preceded by an x_1 pulse followed by an x_2 pulse. That is, the input vector must be $x_1x_2x_3 = 100, 000, 010, 000, 001$ to assert z_1. Output z_1 will be deasserted at the next active x_1 pulse or x_2 pulse. A representative timing diagram is shown in Figure 8.74.

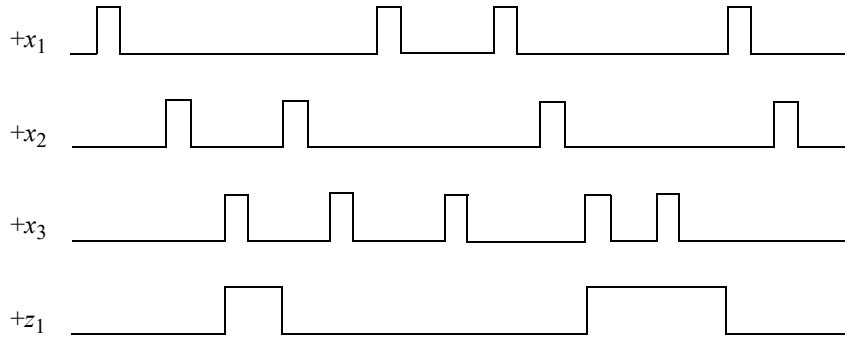


Figure 8.74 Timing diagram for the Moore pulse-mode asynchronous sequential machine of Example 8.12.

The machine will be implemented with NOR *SR* latches and positive-edge-triggered *D* flip-flops as the storage elements in a master-slave configuration. The design module will consist of dataflow modeling and structural modeling. The dataflow construct uses the continuous assignment statement **assign** for the combinational logic primitives, which consist of AND gates, OR gates, and NOR gates. The structural modeling method will instantiate positive-edge-triggered *D* flip-flops that were previously designed using behavioral modeling.

The state diagram is shown in Figure 8.75. The machine remains in state *a* until the first x_1 pulse occurs, which is the correct input to begin a sequence to assert output z_1. As mentioned previously, the state diagram does not contain any complemented variables because a pulse-mode machine with three inputs can have only

$$n + 1 = 3 + 1 = 4$$

combinations of the input variables, including $x_1x_2x_3 = 000$, instead of the 2^n combinations as in synchronous sequential machines.

The input maps are shown in Figure 8.76, where *S* indicates that the latch will be set; *s* indicates that the latch will remain set; *R* indicates that the latch will be reset; and *r* indicates that the latch will remain reset. The input equations are shown in Equation 8.13.

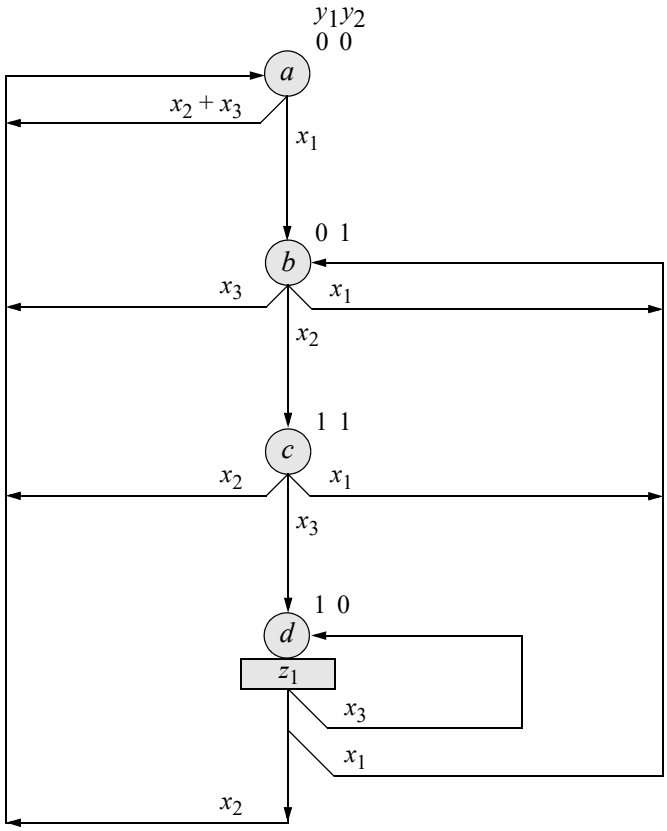


Figure 8.75 State diagram for the Moore pulse-mode machine of Example 8.12.

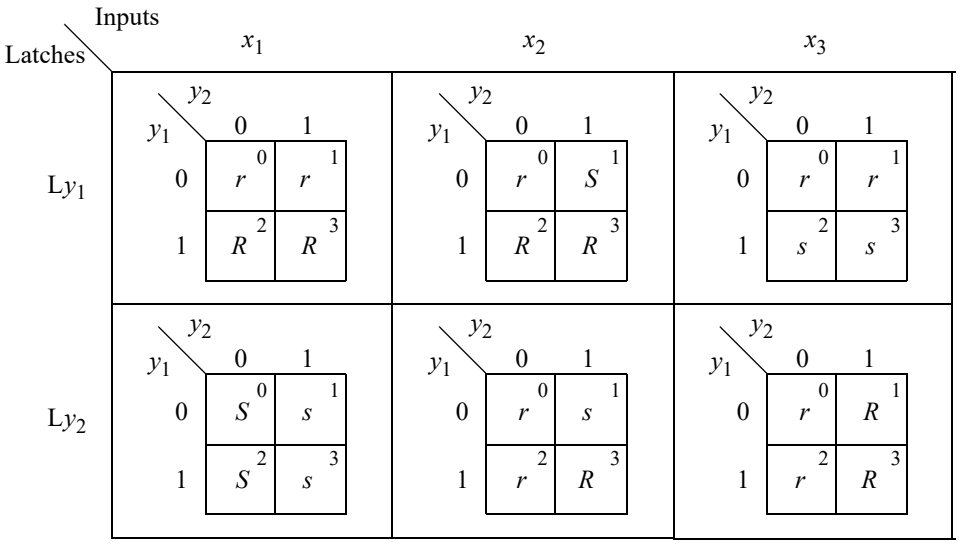


Figure 8.76 Input maps for the Moore pulse-mode machine of Example 8.12.

$$SLy_1 = y_1' y_2 x_2 \qquad RLy_1 = x_1 + y_1 x_2$$
$$SLy_2 = x_1 \qquad RLy_2 = y_1 x_2 + x_3 \tag{8.15}$$

The output map and equation are shown in Figure 8.77 and Equation 8.16, respectively. The logic diagram is shown in Figure 8.78.

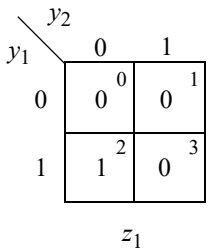


Figure 8.77 Output map for the Moore pulse-mode machine of Example 8.12.

$$z_1 = y_1 y_2' \tag{8.16}$$

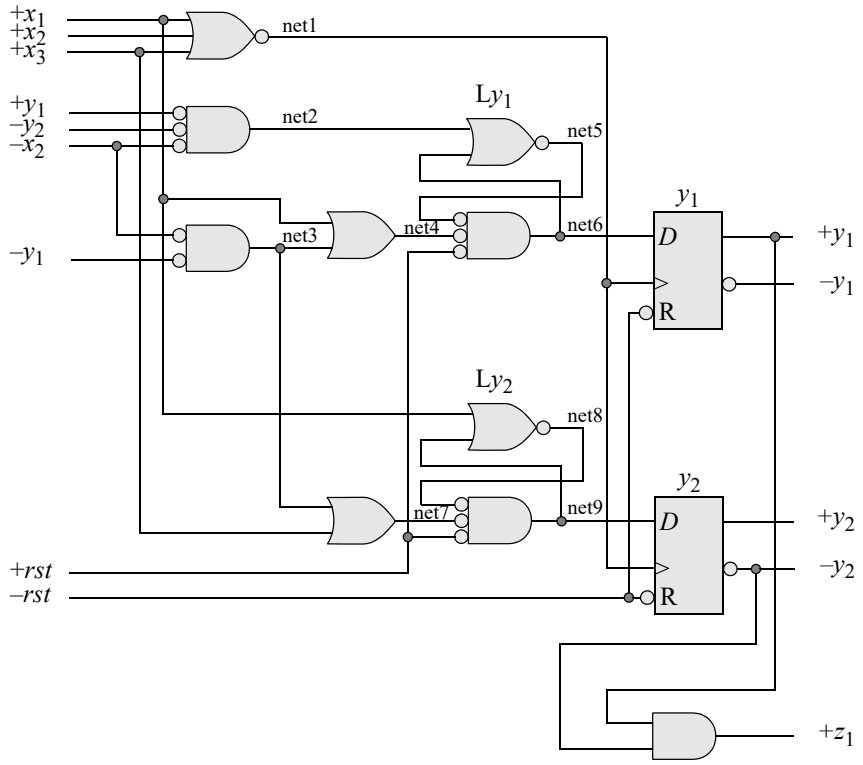


Figure 8.78 Logic diagram for the Moore pulse-mode machine of Example 8.12.

The mixed-design module is shown in Figure 8.79, where the continuous assignment statement is used to design the primitive logic gates by assigning the logic function to the net names. The test bench is shown in Figure 8.80, which sequences the machine through the different paths in the state diagram. The outputs and waveforms are shown in Figure 8.81 and Figure 8.82, respectively.

```
//dataflow/structural moore pulse-mode
//asynchronous sequential machine
module pm_asm11 (rst_n, rst, x1, x2, x3, y, z1);

input rst_n, rst, x1, x2, x3;
output [1:2] y;
output z1;

//define internal nets
wire  net1, net2, net3, net4, net5, net6,
      net7, net8, net9;

//design for the D flip-flops clock
assign net1 = ~(x1 | x2 | x3);

//design for latch Ly1
assign   net2 = ~(y[1] | ~y[2] | ~x2),
         net3 = ~(~y[1] | ~x2),
         net4 = (net3 | x1),
         net5 = ~(net2 | net6),
         net6 = ~(net5 | net4 | rst);

//instantiate the D flip-flop for y[1]
d_ff_bh inst1 (
   .rst_n(rst_n),
   .clk(net1),
   .d(net6),
   .q(y[1])
   );

//design for latch for Ly2
assign   net7 = (net3 | x3),
         net8 = ~(x1 | net9),
         net9 = ~(net8 | net7 | rst);

//continued on next page
```

Figure 8.79 Mixed-design module for the Moore pulse-mode machine of Example 8.12.

```
//instantiate the D flip-flop for y[2]
d_ff_bh inst2 (
   .rst_n(rst_n),
   .clk(net1),
   .d(net9),
   .q(y[2])
   );

//design the logic for output z1
assign   z1 = y[1] & ~y[2];

endmodule
```

Figure 8.79 (Continued)

```
//test bench for moore pulse-mode
//asynchronous sequential machine
module pm_asm11_tb;

reg rst_n, rst, x1, x2, x3;
wire [1:2] y;
wire z1;

//display inputs and output
initial
$monitor ("x1x2x3 = %b, state = %b, z1 = %b",
            {x1, x2, x3}, y, z1);

//define input sequence
initial
begin
   #0    rst_n = 1'b0;     //reset to state_a (00)
         rst = 1'b1;
         x1 = 1'b0;
         x2 = 1'b0;
         x3 = 1'b0;

   #5    rst_n = 1'b1;
         rst = 1'b0;

   #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
   #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_a

//continued on next page
```

Figure 8.80 Test bench for the Moore pulse-mode machine of Example 8.12.

```
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b1;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_a

    #10   x1 = 1'b1;  x2 = 1'b0;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_b

    #10   x1 = 1'b1;  x2 = 1'b0;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_b

    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b1;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_a

    #10   x1 = 1'b1;  x2 = 1'b0;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_b

    #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_c

    #10   x1 = 1'b1;  x2 = 1'b0;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_b

    #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_c

    #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_a

    #10   x1 = 1'b1;  x2 = 1'b0;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_b

    #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_c

    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b1;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_d); z1

    #10   x1 = 1'b1;  x2 = 1'b0;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_b

    #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_c

    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b1;
    #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_d; z1

//continued on next page
```

Figure 8.80 (Continued)

```
   #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_d; z1

   #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b1;
   #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_d; z1

   #10   x1 = 1'b0;  x2 = 1'b1;  x3 = 1'b0;
   #10   x1 = 1'b0;  x2 = 1'b0;  x3 = 1'b0; //to state_a
   #20   $stop;
end

//instantiate the module into the test bench
pm_asm11 inst1 (
   .rst_n(rst_n),
   .rst(rst),
   .x1(x1),
   .x2(x2),
   .x3(x3),
   .y(y),
   .z1(z1)
   );
endmodule
```

Figure 8.80 (Continued)

```
x1x2x3=000, state=00, z1=0
x1x2x3=010, state=00, z1=0
x1x2x3=000, state=00, z1=0
x1x2x3=001, state=00, z1=0
x1x2x3=000, state=00, z1=0
x1x2x3=100, state=00, z1=0
x1x2x3=000, state=01, z1=0
x1x2x3=100, state=01, z1=0
x1x2x3=000, state=01, z1=0
x1x2x3=001, state=01, z1=0
x1x2x3=000, state=00, z1=0
x1x2x3=100, state=00, z1=0
x1x2x3=000, state=01, z1=0
x1x2x3=010, state=01, z1=0
x1x2x3=000, state=11, z1=0
x1x2x3=100, state=11, z1=0
x1x2x3=000, state=01, z1=0
x1x2x3=010, state=01, z1=0
x1x2x3=000, state=11, z1=0
x1x2x3=010, state=11, z1=0
```

```
x1x2x3=000, state=00, z1=0
x1x2x3=100, state=00, z1=0
x1x2x3=000, state=01, z1=0
x1x2x3=010, state=01, z1=0
x1x2x3=000, state=11, z1=0
x1x2x3=001, state=11, z1=0
x1x2x3=000, state=10, z1=1
x1x2x3=100, state=10, z1=1
x1x2x3=000, state=01, z1 0
x1x2x3=010, state=01, z1=0
x1x2x3=000, state=11, z1=0
x1x2x3=001, state=11, z1=0
x1x2x3=000, state=10, z1=1
x1x2x3=001, state=10, z1=1
x1x2x3=000, state=10, z1=1
x1x2x3=001, state=10, z1=1
x1x2x3=000, state=10, z1=1
x1x2x3=010, state=10, z1=1
x1x2x3=000, state=00, z1=0
```

Figure 8.81 Outputs for the Moore pulse-mode machine of Example 8.12.

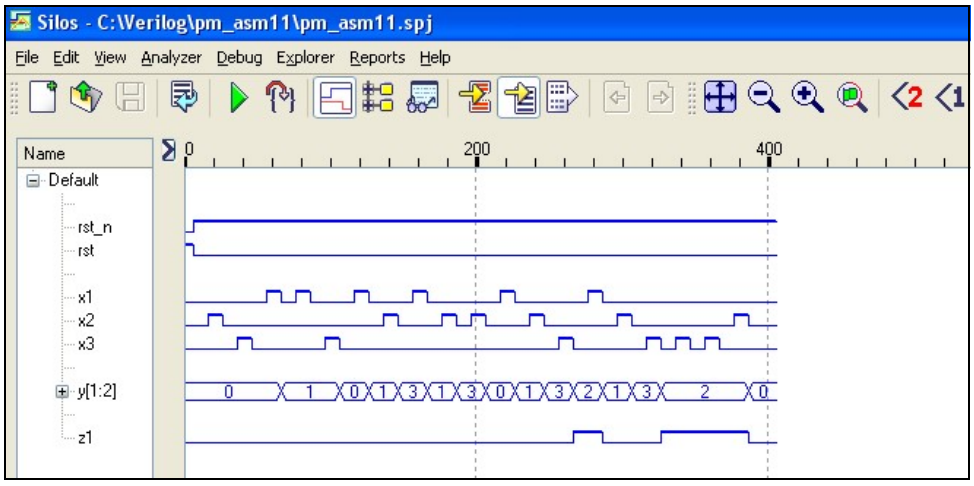


Figure 8.82 Waveforms for the Moore pulse-mode machine of Example 8.12.

Example 8.13 This example repeats Example 8.12, but uses only structural modeling and instantiates previously designed dataflow modules for the logic primitives and latches plus a behavioral module *D* flip-flop for the positive-edge-triggered *D* flip-flops.

The timing diagram, state diagram, input maps and equations, and the output map and equation are the same as those shown in Example 8.12 and will not be replicated. The logic diagram is reproduced in Figure 8.83 and shows the instantiation names and the net names that are used in the structural design module of Figure 8.84.

The test bench is identical except for the module name and the instantiated name of the design module. The outputs are also identical; therefore, the test bench and outputs will not be shown. The waveforms are the same as those of Example 8.12 and are shown in Figure 8.85 for comparison.

This section has presented an important class of asynchronous sequential machines in which pulses trigger state changes. The *SR* latch and *D* flip-flop in a master-slave relationship is by far the most reliable method to implement pulse-mode asynchronous sequential machines.

Other methods may also be used such as *SR* latches as storage elements without *D* flip-flops. As stated previously, the critical factor in the synthesis of pulse-mode sequential machines is controlling the pulse duration of the input signals. If the pulse duration is less than the minimal width to trigger a latch or greater than the maximal width that ensures only one state change, then the machine will not function reliably in a deterministic manner.

Another method to design pulse-mode machines is to use level-sensitive *T* flip-flops as storage elements that are designed with *SR* latches. A *T* flip-flop has one input *T* and two complementary outputs. If the flip-flop is reset, then an active pulse on the *T* input will toggle the flip-flop to the set state; if the flip-flop is set, then a pulse on the *T* input will toggle the flip-flop to the reset state.

Another flip-flop that is occasionally used in pulse-mode machines is the *SR*-*T* flip-flop. The *SR*-*T* flip-flop possesses the combined operational characteristics of both the *SR* latch and the *T* flip-flop. The *SR*-*T* flip-flop contains three inputs, set (*S*), reset (*R*), and toggle (*T*). If the flip-flop is reset, then a pulse on either the *S* input, the *T* input, or both will set the flip-flop. A pulse on the *R* input will have no effect. If the flip-flop is set, then a pulse on either the *R* input, the *T* input, or both will reset the flip-flop. A pulse on the *S* input will cause no change.

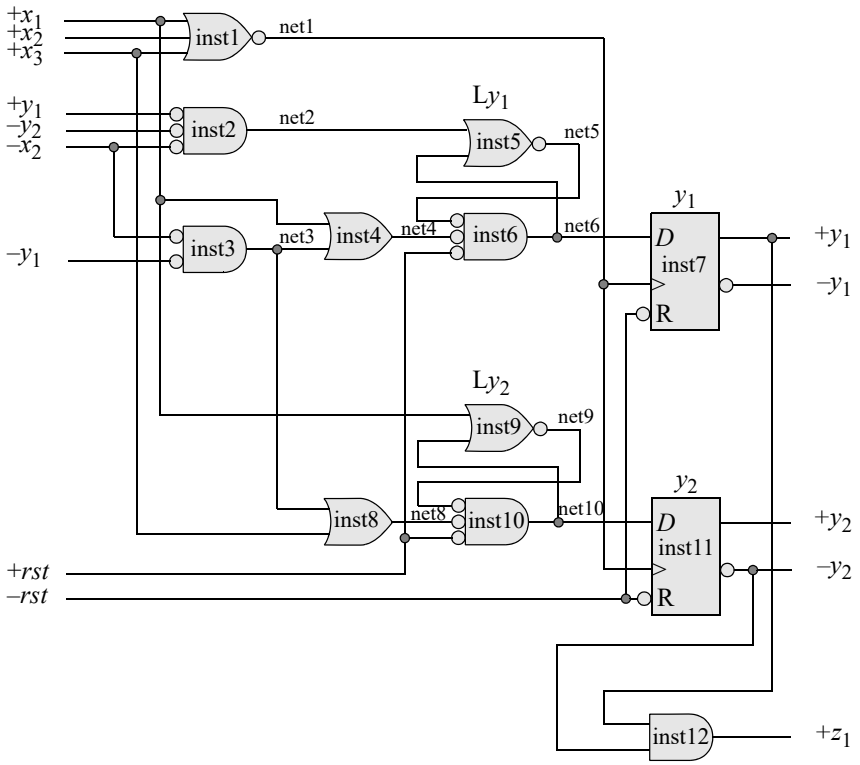


Figure 8.83 Logic diagram for the Moore pulse-mode machine of Example 8.13.

```
//structural moore pulse-mode
//asynchronous sequential machine
module pm_asm9a (rst_n, rst, x1, x2, x3, y, z1);

input rst_n, rst, x1, x2, x3;
output [1:2] y;
output z1;
//continued on next page
```

Figure 8.84 Structural module for the Moore pulse-mode machine of Example 8.13.

```
//define internal nets
wire  net1, net2, net3, net4, net5, net6,
      net8, net9, net10;

//design for the D flip-flops clock
nor3_df inst1 (
   .x1(x1),
   .x2(x2),
   .x3(x3),
   .z1(net1)
   );

//instantiate the logic for latch Ly1
nor3_df inst2 (
   .x1(y[1]),
   .x2(~y[2]),
   .x3(~x2),
   .z1(net2)
   );

nor2_df inst3 (
   .x1(~y[1]),
   .x2(~x2),
   .z1(net3)
   );

or2_df inst4 (
   .x1(x1),
   .x2(net3),
   .z1(net4)
   );

nor2_df inst5 (
   .x1(net2),
   .x2(net6),
   .z1(net5)
   );

nor3_df inst6 (
   .x1(net5),
   .x2(net4),
   .x3(rst),
   .z1(net6)
   );

//continued on next page
```

Figure 8.84 (Continued)

```
//instantiate the D flip-flop for y[1]
d_ff_bh inst7 (
   .rst_n(rst_n),
   .clk(net1),
   .d(net6),
   .q(y[1])
   );

//instantiate the logic for latch Ly2
or2_df inst8 (
   .x1(x3),
   .x2(net3),
   .z1(net8)
   );

nor2_df inst9 (
   .x1(x1),
   .x2(net10),
   .z1(net9)
   );

nor3_df inst10 (
   .x1(net9),
   .x2(net8),
   .x3(rst),
   .z1(net10)
   );

//instantiate the D flip-flop for y[2]
d_ff_bh inst11 (
   .rst_n(rst_n),
   .clk(net1),
   .d(net10),
   .q(y[2])
   );

//instantiate the logic for output z1
and2_df inst12 (
   .x1(y[1]),
   .x2(~y[2]),
   .z1(z1)
   );

endmodule
```

Figure 8.84 (Continued)

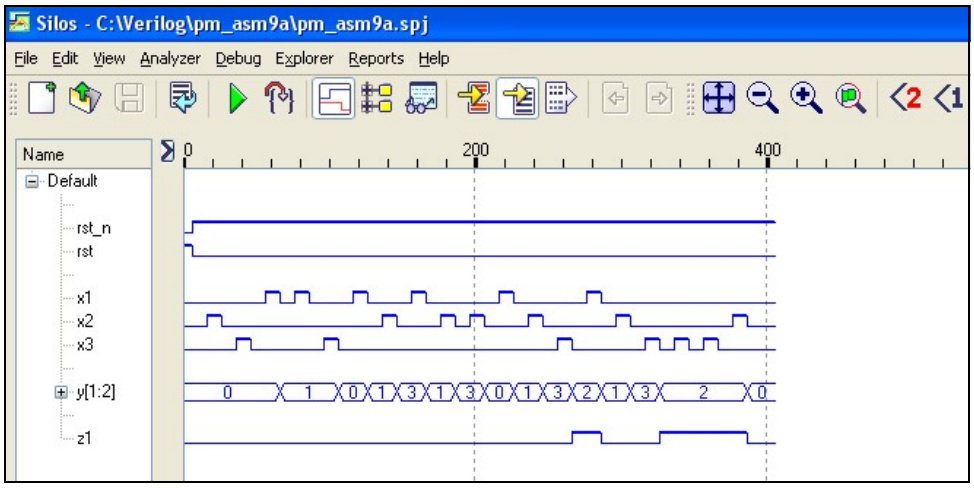
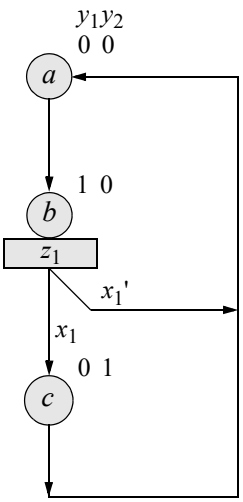


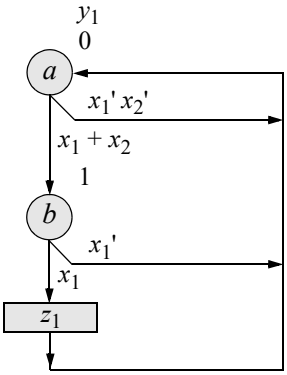
Figure 8.85 Waveforms for the Moore pulse-mode machine of Example 8.13.

8.4 Problems

8.1 Given the state diagram shown below for a Moore synchronous sequential machine, obtain the input maps for negative-edge-triggered *JK* flip-flops and the logic diagram. Then implement the machine using structural modeling. Obtain the test bench, the outputs, and the waveforms.

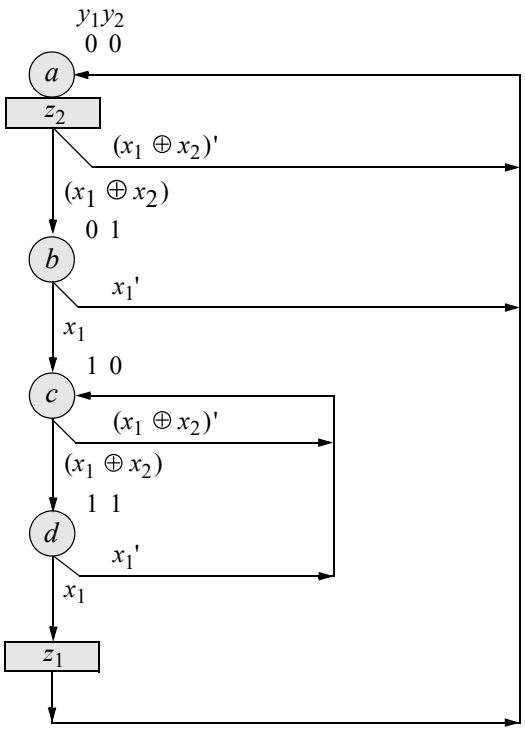


8.2 Use structural modeling to design the Mealy synchronous sequential machine that is represented by the state diagram shown below. Use a positive-edge-triggered *JK* flip-flop and additional AND gates and OR gates. Obtain the structural module, the test bench module, the outputs, and the waveforms.



8.3 A state diagram is shown below for a synchronous sequential machine with both Moore and Mealy outputs. Use behavioral modeling with the **parameter** keyword and the **case** statement to design the machine. The outputs are implemented with the continuous assignment construct.

Obtain the behavioral module using negative-edge-triggered flip-flops, the test bench module, the outputs, and the waveforms.

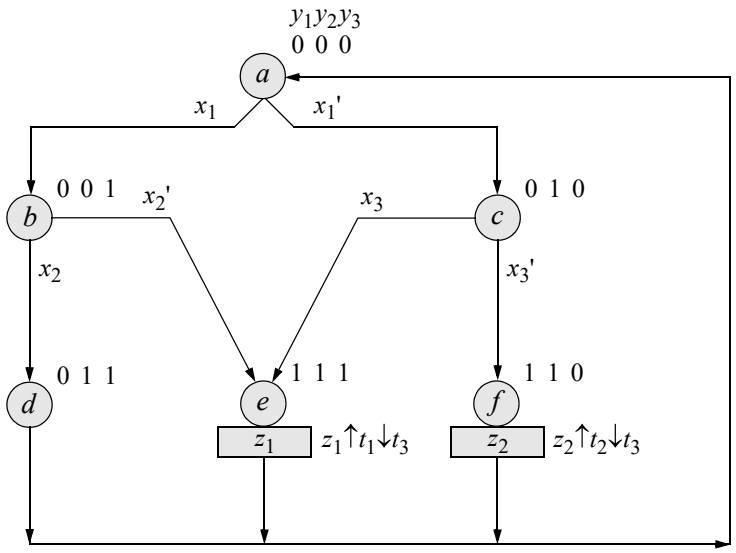


8.4 Design a modulo-10 counter with a self-starting state of $y_1y_2y_3y_4 = 0000$; that is, any count that is not part of the modulo-10 counting sequence has a next state of $y_1y_2y_3y_4 = 0000$. The counting sequence is as follows: $y_1y_2y_3y_4 = 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 0000$. Obtain the input maps, the input equations, and the logic diagram.

Obtain the structural module using dataflow logic primitives of AND gates and OR gates plus positive-edge-triggered *D* flip-flops as the storage elements that are defined using behavioral modeling. Obtain the test bench module, the outputs, and waveforms.

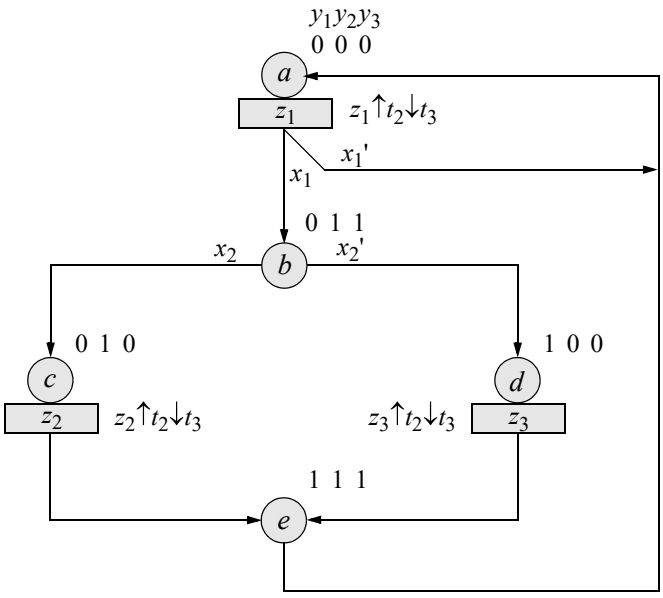
8.5 Design a Moore synchronous sequential machine that generates the following sequence: $y_1y_2y_3y_4 = 1000, 0100, 0010, 0001, 1000, \ldots$. The unused states can be treated as "don't care" states. Obtain the input maps, the input equations, and the logic diagram. Obtain the structural module using positive-edge-triggered *D* flip-flops with active-low set inputs and active-low reset inputs. Obtain the test bench module, the outputs, and the waveforms.

8.6 Given the state diagram shown below for a synchronous sequential machine, obtain the next-state table, the input maps for *JK* flip-flops, the input equations, and the logic diagram using AND gates and OR gates and positive-edge-triggered *JK* flip-flops.



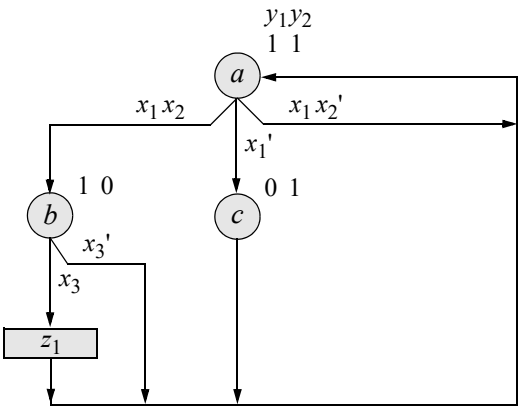
Since there may be a glitch on output z_2 for a transition from state *c* ($y_1y_2y_3 = 010$) to state *e* ($y_1y_2y_3 = 111$), the assertion/deassertion is $z_2\uparrow t_2\downarrow t_3$. Obtain the structural design module by instantiating dataflow modules for the logic primitives and a behavioral module for the positive-edge-triggered *JK* flip-flops. Obtain the test bench module, the outputs, and the waveforms.

8.7 Use behavioral modeling to design the Moore synchronous sequential machine shown in the state diagram below. All outputs are asserted at time t_2 and deasserted at time t_3. Obtain the test bench, the outputs, and the waveforms.

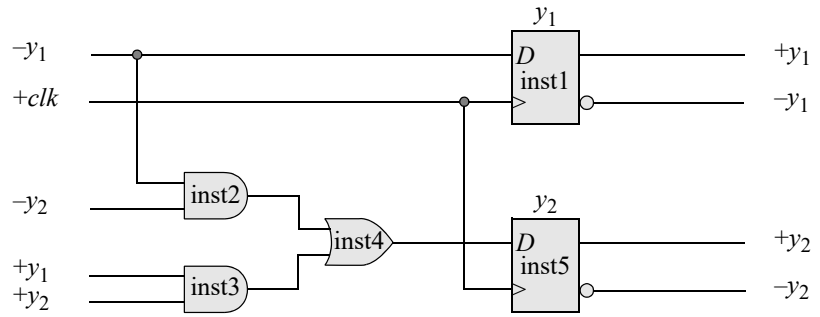


8.8 Given the state diagram shown below for a Mealy synchronous sequential machine, obtain next-state table, the input maps for *JK* flip-flops, the input equations, the output map, the output equation, and the logic diagram using NOR logic for the logic gates and positive-edge-triggered *JK* flip-flops for the storage elements.

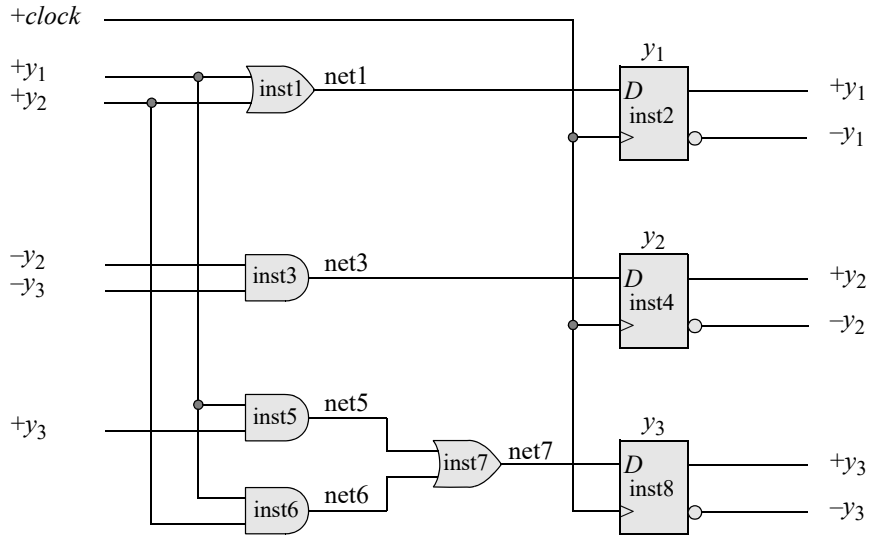
Then design the machine using structural modeling and instantiate logic primitives that were designed using dataflow modeling. The *JK* flip-flops are designed using behavioral modeling. Obtain the test bench, the outputs, and the waveforms.



8.9 Determine the counting sequence for the counter shown below by implementing the machine in a structural module. The counter is reset initially to $y_1y_2 = 00$, where y_2 is the low-order flip-flop. Obtain the test bench, outputs, and waveforms.



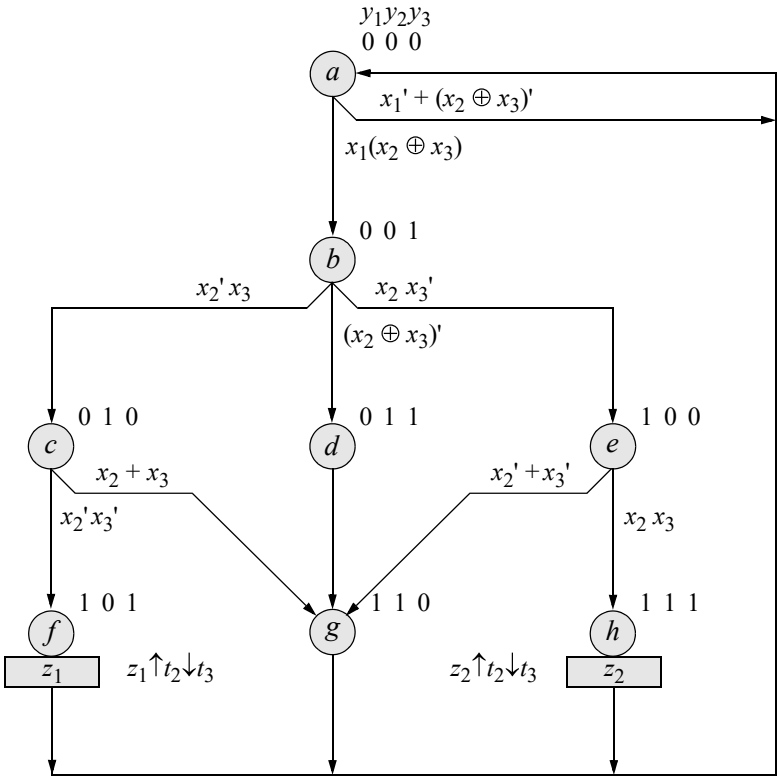
8.10 Determine the counting sequence for the counter shown below. The counter is reset initially to $y_1y_2y_3 = 000$, where y_3 is the low-order flip-flop. Use structural modeling in the implementation. Obtain the test bench, the outputs, and the waveforms.



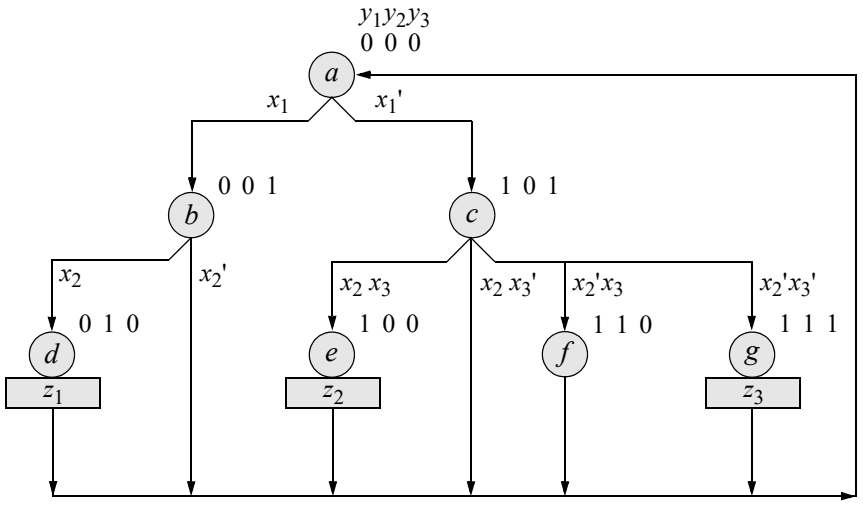
8.11 Given the state diagram shown below for a Moore synchronous sequential machine with three inputs x_1, x_2, and x_3, derive the input maps for flip-flops y_1, y_2, and y_3. Obtain the logic diagram using linear-select multiplexers,

logic primitives, and positive-edge-triggered *D* flip-flops as the storage elements.

The multiplexers are designed using behavioral modeling; the logic primitives are designed using dataflow modeling. The *D* flip-flops are designed using behavioral modeling. Outputs $z_1$and z_2 have the following assertion/deassertion times: $\uparrow t_2 \downarrow t_3$. Obtain the structural module, the test bench module, the outputs, and the waveforms.



8.12 Given the state diagram shown below for a Moore synchronous sequential machine, implement the design using nonlinear-select multiplexers for the δ next-state logic together with logic primitives. Let flip-flops $y_1y_3 = s_1s_0$. Use positive-edge-triggered *D* flip-flops for the storage elements and a decoder for the λ output logic. Since the state codes may cause glitches on the outputs for some state transition sequences, z_1, z_2, and z_3 should be asserted at time t_2 and deasserted at time t_3.Obtain the structural module, the test bench module, the outputs, and the waveforms. Design the 4:1 multiplexer using built-in primitives; design the *D* flip-flops using behavioral modeling; design the decoder using dataflow modeling.



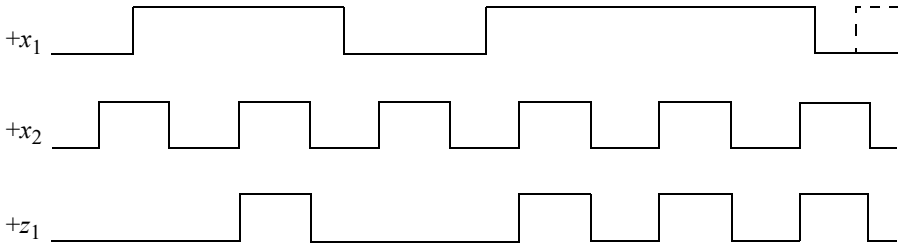
8.13 Design a Moore machine that has four parallel inputs x_1, x_2, x_3, and x_4 and two outputs z_1 and z_2. Output $z_1 \uparrow t_1 \downarrow t_3$; output $z_2 \uparrow t_2 \downarrow t_3$. The inputs constitute a 4-bit word *x[1:4]*. There is one bit space between words. The machine operates as follows:

(a) Output $z_1 = 1$ if the 4-bit word contains an odd number of 1s.
(b) Output $z_2 = 1$ if the 4-bit word contains an even number of 1s, including zero 1s.

Use behavioral modeling to implement the machine, then obtain the test bench, the outputs, and the waveforms.

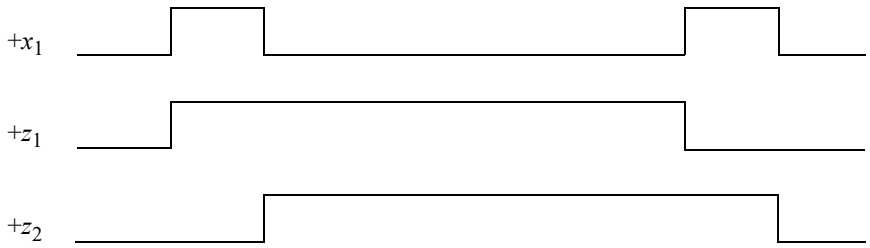
8.14 Design a Mealy asynchronous sequential machine using structural modeling that generates an output z_1 whenever a serial data line x_1 contains a sequence of three consecutive 1s. Overlapping sequences are allowed. Obtain the state diagram, the input maps, and the logic diagram using AND gates and OR gates for the logic primitives and positive-edge-triggered *D* flip-flops as the storage elements. Obtain the structural module, the test bench module showing overlapping sequences, the outputs, and the waveforms.

8.15 This design replicates the asynchronous sequential machine of Example 7.12 and is to be designed using dataflow modeling. The machine has two inputs x_1 and x_2 and one output z_1, as shown in the timing diagram below. Input x_1 acts as a gate for x_2; that is, x_2 will be gated to output z_1 only if x_1 precedes the assertion of x_2. If x_1 becomes deasserted while x_2 is asserted, then the full width of the x_2 pulse will appear on z_1 — the width of the x_2 pulse will not be decreased.



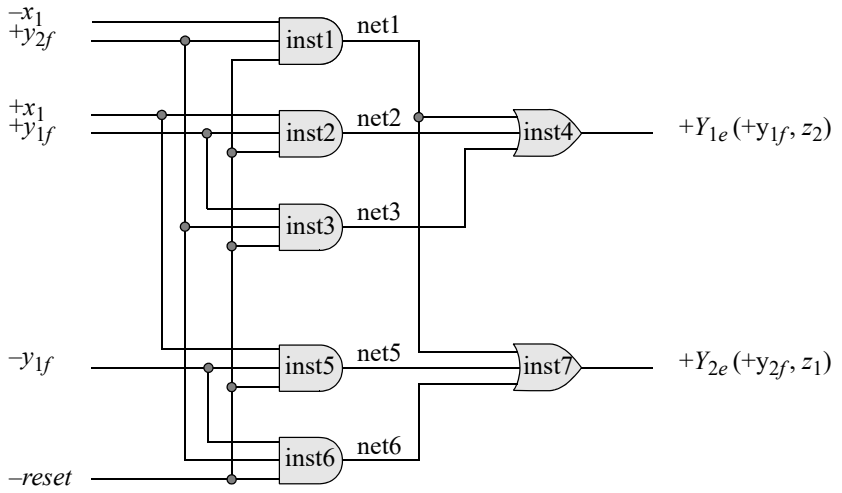
Refer to Example 7.12 for the design procedure for this problem. The resulting logic diagram is shown below with one excitation variable Y_{1e}. Obtain the design module using dataflow modeling, the test bench module, the outputs, and the waveforms.

8.16 Refer to Example 7.14 for the design procedure for this problem. The timing diagram for the asynchronous sequential machine is shown below with one input x_1 and two outputs z_1 and z_2. The outputs will be implemented with the least amount of logic which may not necessarily be the fastest response time.

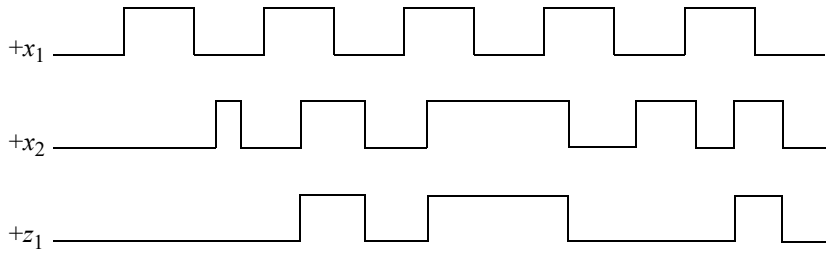


The primitive flow table and the logic diagram are shown below. The logic diagram is to be implemented using structural modeling. The logic primitives are to be designed using dataflow modeling. Obtain the structural design module, the test bench module, the outputs, and the waveforms.

x_1	0	1	z_1	z_2
	(a)	b	0	0
	c	(b)	1	0
	(c)	d	1	1
	a	(d)	0	1



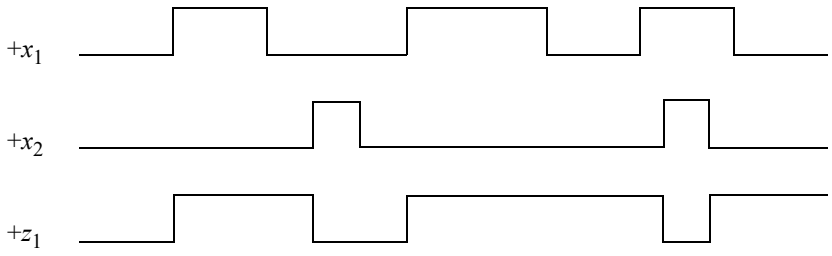
8.17 Design an asynchronous sequential machine which has two inputs x_1 and x_2 and one output z_1. Output z_1 is asserted for the duration of x_2 if and only if x_1 is already asserted. Assume that the initial state of the machine is $x_1x_2z_1$ = 000. A representative timing diagram is shown below. Design the machine using dataflow modeling, and obtain the test bench, the outputs, and the waveforms.



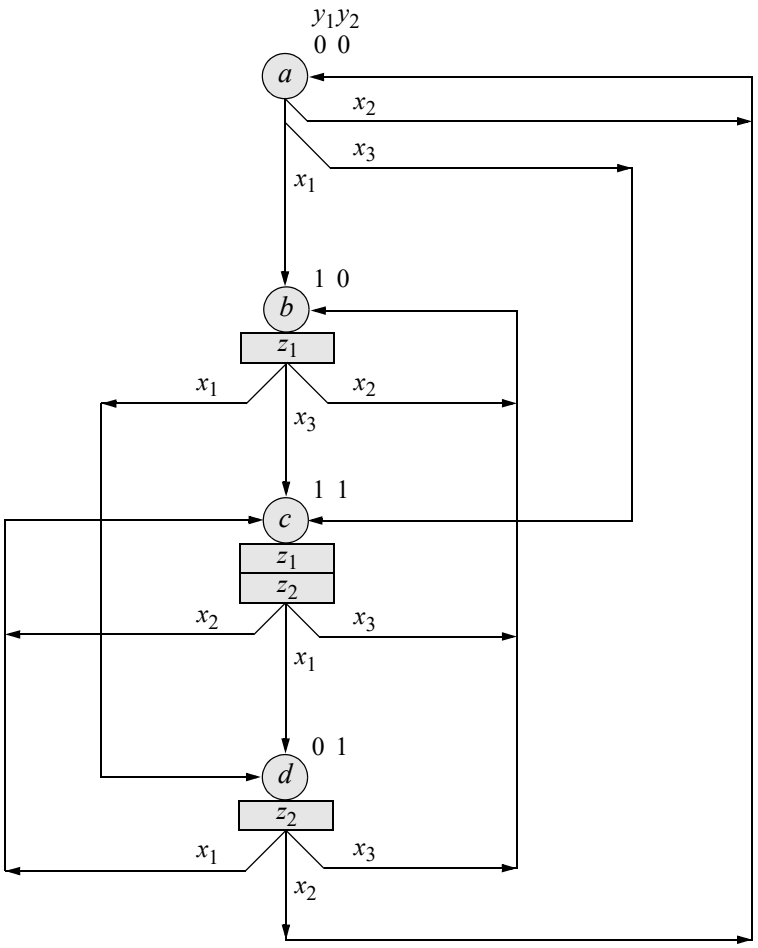
8.18 An asynchronous sequential machine has two inputs x_1 and x_2 and one output z_1. The machine operates according to the following specifications:

If x_1x_2 = 00, then the state of z_1 is unchanged.
If x_1x_2 = 01, then z_1 is deasserted.
If x_1x_2 = 10, then z_1 is asserted.
If x_1x_2 = 11, then z_1 changes state.

Design the machine using dataflow modeling. The inputs are available in both high and low assertion. Assume that the initial conditions are $x_1x_2z_1$ = 000. The output must change as fast as possible. There must be no output glitches. A representative timing diagram is shown below. Obtain the dataflow design module, the test bench module, the outputs, and the waveforms.

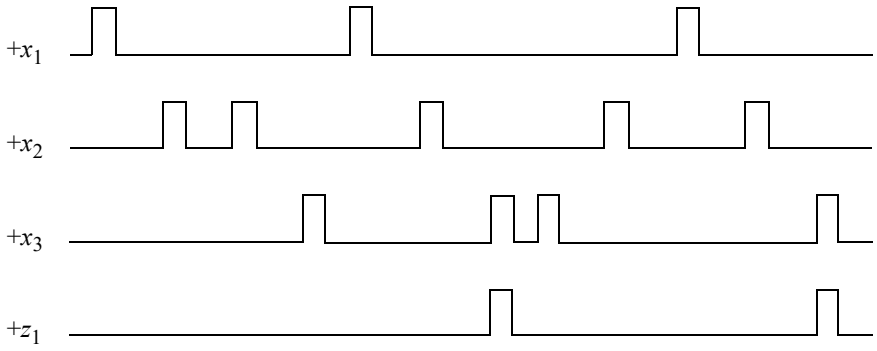


8.19 Given the state diagram shown below, design a Moore pulse-mode asynchronous sequential machine using *SR* latches and *D* flip-flops in a master-slave configuration. Obtain the input maps, the input equations, and the logic diagram. Use structural modeling for the design module by instantiating dataflow modules for the logic primitives and a behavioral module for the *D* flip-flops. Obtain the test bench, the outputs, and the waveforms.



8.20 Design a Mealy machine that has three pulse input variables x_1, x_2, and x_3 and one output z_1 that is asserted coincident with x_3 whenever the sequence $x_1x_2x_3$ = 100, 010, 001 occurs. The storage elements will consist of *SR* latches and positive-edge-triggered *D* flip-flops in a master-slave configuration.

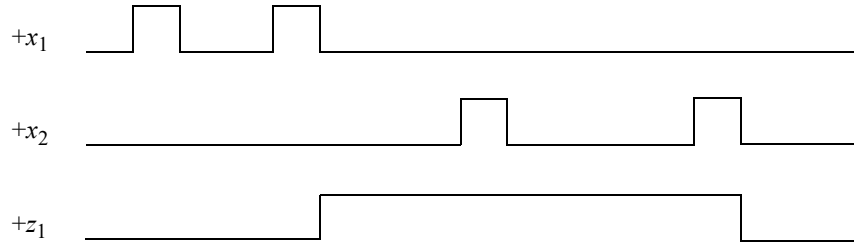
A representative timing diagram displaying valid input sequences and corresponding outputs is shown below. State code assignment is arbitrary for the state diagram, since input pulses trigger all state transitions and the machine does not begin to sequence to the next state until the input pulse, which initiated the transition, has been deasserted. Thus, output z_1 will not glitch. Obtain the input maps, the input equations, and the logic diagram. Then obtain the structural module using dataflow modeling for the logic primitives. Use NOR logic that is designed using dataflow modeling for the *SR* latches. Use behavioral modeling for the *D* flip-flop. Obtain the test bench, the outputs, and the waveforms.



8.21 Design a Moore pulse-mode asynchronous sequential machine that has two inputs x_1 and x_2 and one output z_1. The negative transition of every second consecutive x_1 pulse will assert output z_1 as a level. The output will remain set for all following contiguous x_1 pulses. The output will be deasserted at the negative transition of the second of two consecutive x_2 pulses. The machine will be implemented with NAND logic for the δ next-state logic. The storage elements will be NAND *SR* latches and positive-edge-triggered *D* flip-flops in a master-slave configuration.

A representative timing diagram is shown. Generate a state diagram that depicts all possible state transition sequences that conform to the functional specifications. Obtain the input maps, the input equations, and the logic diagram.

Then design the structural module using dataflow modeling for the logic primitives and behavioral modeling for the positive-edge-triggered *D* flip-flops. Use dataflow modeling for the implementation of output z_1. Obtain the test bench, the outputs, and the waveforms.



8.22 Given the state diagram shown below for a Moore pulse-mode asynchronous sequential machine, implement the machine using NAND gates for the *SR* latches and *D* flip-flops as the storage elements in a master-slave configuration. Use any type of gates for the logic primitives.

Derive the input maps and input equations, then obtain the logic diagram. Design the machine using dataflow and structural modeling. Use the continuous assignment construct for the logic primitives and latches; instantiate positive-edge-triggered *D* flip-flops that were designed using behavioral modeling.