7 ALGORITHMIC STATE MACHINES

OVERVIEW

Now that we know how flip-flops work, we turn to the major object of digital design—the state machine. The state machine is found everywhere, because all controller implementations are state machines and every nontrivial application requires a controller. Our goal is to learn the modern algorithmic state machine (ASM) method after a brief look at the traditional state diagram method.

We start by discussing the style of the ASM method, because it implements top-down design, then discuss the temptations offered by the risky bottom-up design style. Next we present the basic concepts of state machines: state variable, state, state assignment, branch, unconditional output, conditional output, asynchronous machine, and synchronous machine. We discuss traditional state machine diagrams, notation, and methods for analysis and synthesis. We cover the Moore state machine, which has only unconditional outputs, as well as the Mealy state machine, which has only conditional outputs. The generality is the combined Mealy and Moore circuit, which allows for an easy transition to modern ASM notation and methods.

Next we present the vocabulary and notation of algorithmic state machines. Using this notation, we assemble ASM charts. Given such a chart and timing waveforms for input variables, we determine the sequence(s) of states and the corresponding outputs.

Given an algorithm, we learn how to synthesize a circuit implementing that algorithm. A direct synthesis method always yields a circuit that works. Another method analyzes a circuit to produce the ASM chart representing the circuit. This method is also direct. These methods allow us to analyze and design state machines of any complexity. Specific worked examples provide experience with the methods.

The chapter closes with discussions of control loops in ASM charts and the very important linked ASM charts.

INTRODUCTION

What we are about to discuss are *synchronous state machines,* machines that change state only when a clock edge occurs. Machines without clocks are known as *asynchronous state machines;* they change state when an input changes state. The RS latch is an asynchronous state machine, and so are D and JK flip-flops, in the sense that they are designed by asynchronous methods. (In practice, however, the flip-flops are used and thought of as synchronous devices.) Synthesis and analysis methods for asynchronous machines are to-

tally different from methods for synchronous machines. Chapter 11 is devoted to asynchronous state machines.

The algorithmic state machine (ASM) method uses straightforward procedures to create reliable sequential digital designs known as *sequential state machines*. By the ASM method, we can develop designs with style and elegance. However, neither the ASM method nor any other method eliminates the need for creativity. Creativity is required to construct the basic algorithm that solves the current problem. At the same time, the ASM method is an excellent assistant during the creative phase. The issues of creativity and the conversion of words into ASM charts are addressed in this chapter's examples and in later chapters.

The ASM chart represents the controller.

Good design style is intrinsic to the ASM method. One important attribute of the method is *top-down design*. Top-down design starts with a careful study of the problem. Details are ignored at this stage. The relevant questions include the following, amongst others.

1. Is the problem statement clear?
2. Is the problem statement definitive?
3. Can the problem statement be simplified by restatement?
4. Is the problem part of a larger problem?

The data path processes the data.

The ASM state machine controls the data path.

Another attribute of the ASM method is explicit separation of the controller from the data path controlled by the controller. The *controller* is the state machine—the master. The *data path*—the slave—is the hardware that processes information. The controller manipulates the data path so it executes the required operations on data at the appropriate time. The ASM method emphasizes the creation of a clear and detailed definition of the control algorithm prior to implementing a detailed hardware design.

Good design style does not exclude bottom-up considerations. A typical example of a *bottom-up* process is a selection of "the solution," i.e., some integrated circuit, from an IC data book. The bottom-up process is continued by grafting other parts onto "the solution." In this way we can find an (apparent) solution. The usual justifications are the chip(s) saved, the reputedly short design time, and the presumed higher performance. These are fallacious, however. The contribution of chips to the cost of most products is less than 25%. And the cost of field repairs can exceed the cost of a poorly designed product over the product's lifetime. Finally, a properly executed top-down design yields comparable, if not increased, performance. There is no valid justification for the short-term point of view bottom-up design represents.

There are a great many difficulties with bottom-up designs, including the following.

1. They require a great deal of artistic skill unavailable to most bottom-up designers. (Invariably the result is a high-risk mix of asynchronous and synchronous methods.)
2. No systematic theoretical method is used that in effect certifies the design as reliable. (Explain to management how you are going to resolve the mysterious problems that invariably arise during production.)
3. There is an incomplete understanding of the circuit behavior that forecloses clarity in documentation, such as test, repair, and instruction manuals. ("Just how does this work so I can fix it?")

Nevertheless, an intimate knowledge of hardware influences any design decisions. The challenge is to use this knowledge in a way that avoids wasting time on unproductive choices. Discipline keeps bottom-up design in perspective.

The Controller and the Data Path An important attribute of the ASM method is *explicit* separation of the data path from the controller that controls the data path. Controller outputs manipulate the data path so it can perform the required data operations at the appropriate times.

The top-down design process begins with the selection of the data processing algorithm and its translation into a data path circuit. The algorithm might be a known procedure or one of our own creation. The selected algorithm is translated into a hardware configuration that processes the data as desired. One example is an algorithm multiplying two 12-bit binary numbers using a shift-and-add process, which is translated into a data path consisting of an adder, a counter, and shifters. Another example is a memory system whose data path consists of address decoders and an array of memory chips, a case in which data processing is limited to reading data from and writing data to storage.

Once the data path is defined, we list the inputs required to configure it and the inputs required for data. For example, configuration inputs select one of several sources to load into a selected register. Data path configuration inputs become controller outputs. Furthermore, we list the data path outputs required by the task as well as those the controller needs as inputs for confirming execution or ascertaining status of data path operations.

The top-down process ends with the design and implementation of the controller. Controller inputs represent external events or data path outputs reporting data processing results. Controller outputs configure the data path so desired actions take place. If this digital system is a subsystem of another system, there may be additional controller outputs reporting to the higher-level system.

Knowing the inputs, the outputs, and the algorithm, the controller is designed and implemented. Note that the controller cannot be designed until the data path is known.

7.1 Fundamentals

We start by relating everyday activities to the language of state machines. Suppose you are at work reading a technical report and the boss calls and asks you to come into her office. You stop reading and do so. In other words, the boss's input forces you to branch from your present activity (reading the report) to the next activity (going to the boss's office). Until the input was asserted, you were in your present activity. New inputs transition you into the next activity. *Present state* is your activity before inputs change. *Next state* is your activity after you have reacted to changes in inputs. Your behavior is asynchronous. Moreover, you are storing the present state in a part of your mind, which is a *state register* that remembers.

A clock synchronizes activities. Suppose there is a clock in the background that emits a timing mark periodically (e.g., a high-pitched chirp emitted once a minute). Furthermore, suppose you agreed not to change activity until inputs change *and* the next timing mark occurs. Now your behavior has become synchronous. In a sequential digital design, a clock emits pulses periodically. Each clock pulse orders a move to the next state, which can be the same as the present state or a different state. You were dwelling for many clock periods in the read-the-report present state until the telephone rang *and* a timing mark occurred.

Many tasks require events to take place sequentialy over time. When the present event ends, the next event starts immediately thereafter. Or the present event cannot end until some input variable changes state. In the meantime, the task dwells in the present state. And there may be more than one next event possible. Or the present event is repeated n times. And so forth.

Let us state all this more precisely.

State variables represent the state of a machine.

State variable: a stored quantity capable of assuming the value 0 or 1.

Example: Flip-flop output q is a state variable.

State: The state of a machine encodes in a memory sufficient past history so that future behavior may be determined. Sufficient information is available to determine from the present state and the present inputs both the present outputs and the next state. The present state of the sequential digital circuit is represented by a set of state variables.

Example: A traffic light controller encodes as a state the fact that the east–west light is red and the north–south light is green.

Example: The state of a sequential digital circuit with three flip-flops is represented by the set of three flip-flop outputs q_j concatenated in some order such as $q_2q_1q_0$. (The three flip-flops constitute the state register—see *State assignment.*)

State assignment: The process of state assignment converts each state identifier to a pattern of state variables. The pattern is usually a minterm whose literals are the state variables. In effect the process of state assignment gives each state a number. The present-state number is stored in a group of flip-flops, called the *state register,* which is the physical representation of the state machine's present state. The state assignments can be arbitrary; however, there are preferred choices. Any set of numbers assigned to states affects in an unpredictable manner the complexity of the hardware computing the next-state number and the hardware implementing the output equations. When the number of states is small, the hardware consequences of choices may be investigated in a reasonable amount of time. For larger state machines, computer-aided design (CAD) tools are recommended. There is a large body of knowledge on this complex subject, which is outside the scope of this text. Here we will use a simple binary representation for the state identifier number.

Branch: Associated with each state are none, one, two, or more input variables. The next state is selected by the present (associated) state and the present value of these input variables in some combination. This next-state decision is referred to as a *branch.* The next state is determined without conditions when there are no input variables associated with the present state. Input variables *not* associated with a present state have "don't care" status.

Outputs: States may specify unconditional or conditional outputs. Unconditional outputs assigned to a state are active when the system is in that state, without regard to the input conditions. Conditional outputs assigned to a state are active when the system is in that state *and* when input conditions are as specified for the output (see *Branch* above).

Each state can specify unconditional and conditional outputs. The same output can be unconditional in one state and conditional in another state.

An asynchronous machine moves to the next state when an input change occurs.

Asynchronous machine: The present state is the state of the q_j at the time before inputs change. The next state is the state of the q_j after the circuit has reacted to input changes. (Restrictions on input changes are discussed in Chapter 11.)

Synchronous machine: The present state is the state of the q_j at the time before a clock edge occurs when edge triggered flip-flops are used. The next state is the state of the q_j after the clock edge occurs and after the circuit has reacted to the clock edge. The present state and the input values immediately before the clock edge determine *which* state is next. The clock edge determines *when* the state machine goes to the next state.

Algorithmic state machine chart: This documents the controller's algorithm. The chart is a network of states with unconditional outputs, branches, and conditional outputs. When we do not have the algorithm clearly in mind, the process of building the chart helps us refine the algorithm.

A synchronous machine moves to the next state when an active clock edge occurs.

To summarize, then: A synchronous sequential logic machine includes a sequence-of-events controller that moves the machine from state to state when clock edges occur. Each clock edge asks the question "Which next state do we move to now?" The controller uses the present state and the present values of the input variables to decide which next state to move to, and clock edges decide when to move. Constructing an algorithm requires gathering information on the desired sequence of output events. This information is translated into sequences of states linked by paths between states. A path may or may not include branch decision points. If a path does *not* include branch decision points, the next clock edge orders the machine to follow the path from its present state to the unique next state. When a path includes a branch decision point(s), the selected path is the path

activated by the inputs. (These ideas are illustrated in Figure 7.19 on page 372). The set of states linked by paths defines the possible sequences of states that can be specified by inputs. When a logic machine is in a state, none, one, two, or more output events can be specified to occur. The outputs may be conditional or unconditional.

7.2 State Diagram Method

Prior to development of the algorithm state machine method, designers used the *state diagram method*. This was during the era of the Mealy and the Moore machine structures. In state diagram terms, an ASM machine with only unconditional outputs is a Moore machine, and an ASM machine with only conditional outputs is a Mealy machine. The ASM method recognizes that the truly general machine is a combination of both structures: a Mealy-Moore machine structure. The basic goals of any state machine synthesis method is determination of the present-state output function and the next-state function. We start with an analysis of a state machine using the state diagram method.

7.2.1 State Machine Diagram Notation

Circles represent states and arrows point to next states.

State The circle is state s_j's symbol (Figure 7.1a). The state identifier s_j is placed inside the circle.

Unconditional outputs One or more unconditional outputs active in state s_j are listed beneath the state identifier inside the circle.

Arrow If n input variables are associated with a state, then there are 2^n next states. This means there are 2^n arrows connecting the present-state circle to the next-state circles. When $n = 2$, then four arrows connect the present state to four possible next states. The associated function of input variables is written above each arrow (Figure 7.1b).

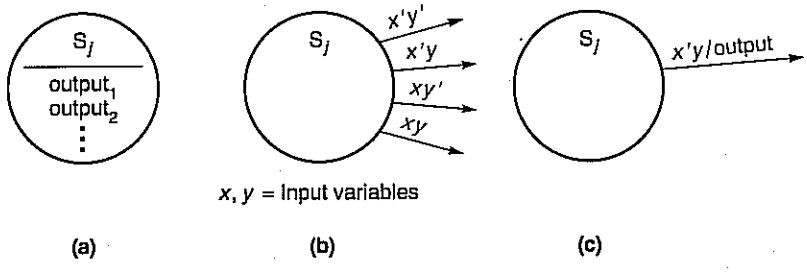


FIGURE 7.1
State diagram notation

Conditional outputs One or more conditional outputs active in state s_j are listed as expressions above the arrow (Figure 7.1c).

7.2.2 State Machine Analysis

Analysis of state machine circuits allows us to introduce a number of state machine design tools. The analysis method goes as follows.

STEP 1 Derive the flip-flop next-state equations from the circuit.

STEP 2 Derive the output equations from the circuit.

STEP 3 Make a state transition table with columns for the present state (PS), each input, the next state (NS), and each output.

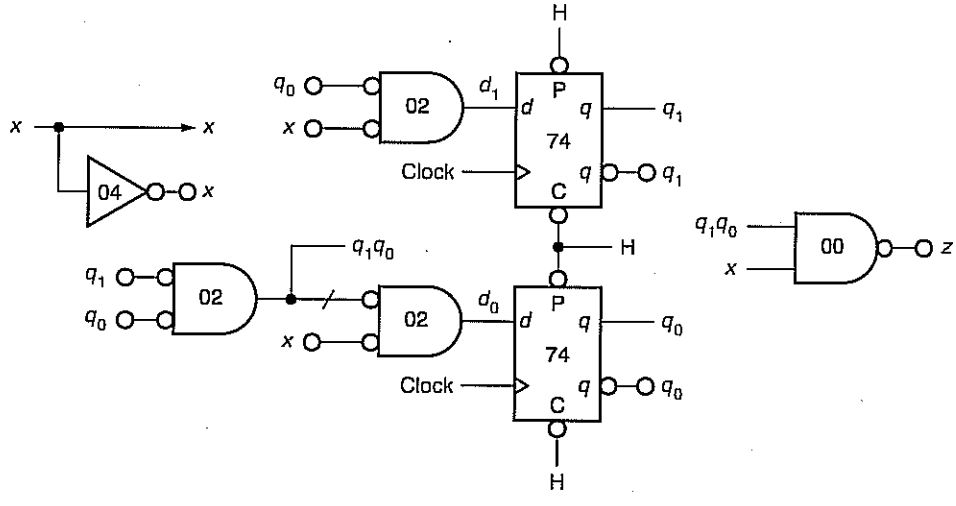
STEP 4 Fill the input side of the state transition table. List all combinations of values for the present-state and machine inputs.

STEP 5 Fill the output side of the state transition table. Evaluate the next-state equations and the output equations for each combination of values in the input side.

STEP 6 Use the information in the state table to make the state diagram.

Specific state machine analysis The goal is to determine the state diagram that represents the state machine in Figure 7.2. The next-state equations q_1^+, q_0^+ and the output z are readily derived from the circuit.

FIGURE 7.2
111 sequence detector



Use flip-flop defining equations.

$$q_1^+ = d_1 = xq_0$$
$$q_0^+ = d_0 = x(q_1q_0)'$$
$$z = xq_1q_0$$

Two flip-flop outputs q_1, q_0 represent two bits, which have four present states that are entered into the PS column of the state transition table in Figure 7.3a. The next-state equations show that input variable x is active in every state. This means there are two exit paths from each state: one when $x = 0$, the other when $x = 1$. Each exit path is represented by a row in the truth table. So at every state we enter 0 and 1 in the input column. (When the number of states is s, the number of inputs is i, and every input is active in every state, then the number of truth tables rows is 2^{s+i}.)

The NS and output columns are filled by evaluating the equations for each present state with x equal to 1 or 0. Finally, we replace the binary state numbers with state identifiers (s_0, s_1, s_2, s_3) and rewrite the table as shown in Figure 7.3b. This table is referred to as a *state table*. The columns of the state table in Figure 7.3b provide us with the information we need to draw the state diagram in Figure 7.4.

In a transition table, states are represented by binary state numbers assigned to the states.

In a state table, states are represented by state identifiers.

FIGURE 7.3
Tables for 111 sequence detector

(a) Transition Table

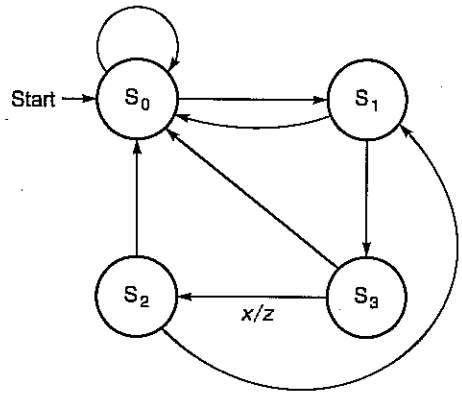
PS[1]	PI[2]	NS[3]	PO[4]
$q_1\,q_0$	x	$q_1^+q_0^+$	z
0 0	0	0 0	0
0 0	1	0 1	0
0 1	0	0 0	0
0 1	1	1 1	0
1 0	0	0 0	0
1 0	1	0 1	0
1 1	0	0 0	0
1 1	1	1 0	1

(b) State Table

PS	Inputs	NS	Outputs
s_j	x	s_j	z
s_0	x'	s_0	0
s_0	x	s_1	0
s_1	x'	s_0	0
s_1	x	s_3	0
s_2	x'	s_0	0
s_2	x	s_1	0
s_3	x'	s_0	0
s_3	x	s_2	1

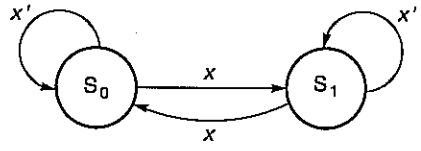
[1] Present state.
[2] Present inputs.
[3] Next state.
[4] Present outputs.

FIGURE 7.4
State diagram for 111 sequence detector—Mealy type



EXERCISE 7.1 The next-state equation for a circuit is $q_0^+ = q_0$ xor x. Derive the state diagram. *Hint:* derive the state transition table.

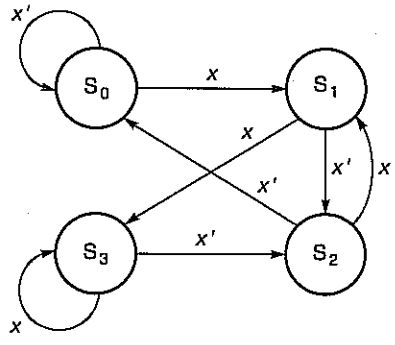
Answer:



■

EXERCISE 7.2 The next-state equations for a circuit are $q_1^+ = q_0$, $q_0^+ = x$. Derive the state diagram. *Hint:* derive the state transition table.

Answer:



■

7.2.3 State Machine Synthesis

State machine synthesis begins with the development of a state diagram. This difficult task has been replaced by the ASM chart method

making the transition to the ASM synthesis method we will take up next.

Given a state diagram, the synthesis process starts by taking information off the state diagram to construct the state transition table. This step includes selection of the type of flip-flop to be used in the state register. Next, the flip-flop input equations and the output equations are derived from the table. Finally, the state machine and output circuits are synthesized from the equations. We use the state machine in Figure 7.5 to explain the process.

A state machine with eight states requires three state variables. Three variables x, y, and z generate eight (2^3) minterms, and each minterm is true for one combination of the constants 0 and 1 (which are not numbers). For example, the minterm m_3 ($= x'yz$) is true for the combination of constants 011. If we interpret 011 as a *number,* we can say a state machine is in state s_3, or state three, when the minterm $x'yz$ is True.

The state diagram representing the state machine has eight nodes (the circles in Figure 7.5), and arrows connect the nodes to each other. That this diagram represents a synchronous state machine is not shown on the diagram. This is an example of what we call *hidden logic*. At each clock edge the machine will move to the next state, as dictated by the arrows. State s_0 differs from the other states because the next state depends on the input variable p. Input variables such as p establish conditions for state transitions. An arrow without an input associated with it, such as the connection from state s_2 to s_3, has no conditions placed on it. The name END beneath state s_7 in the state-7 node is an unconditional output. When in state s_7, output END is True without conditions. We say END is unconditionally True when in state s_7. Let us synthesize a circuit implementing this state diagram.

Next state conditions are entered above the arrows.

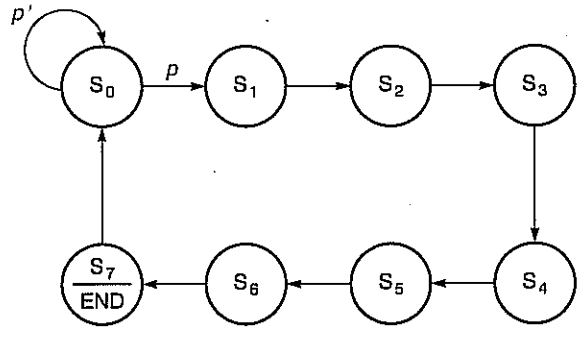


FIGURE 7.5
State diagram—Moore structure

Three-state variables $q_2q_1q_0$ are implemented with three flip-flops. The three flip-flops constitute the state register. Each flip-flop q output is a state variable. Concatenation of the q_j is the state register state: e.g., $s_5 = q_2q_1'q_0$. The state identifier is assigned a binary number equal to its subscript. These convenient assignments may not be the best set of assignments.

STEP 1 List the present states (PS) in a column, as shown in Figure 7.6. In effect this is a list of the states found in the state diagram (Figure 7.5).

STEP 2 Make one column per input variable (Figure 7.6). There is one row in the table for each combination of variables associated with each state. In this case, input variable p is associated only with state 000. Therefore state 000 has two rows corresponding to the p' and p combinations of one variable. We mark the column with 0 or 1 in the rows. Observe that p is a "don't care" input variable for all of the other states.

FIGURE 7.6
State transition table with state identifiers

Step: [1]		[2]	[3]		[4]	[5]
PS		PI	NS		Flip-Flop Inputs	PO
s_j	$q_2q_1q_0$	p	s_j^+	$q_2^+q_1^+q_0^+$	$t_2t_1t_0$	END
s_0	000	0	s_0	000	000	0
s_0	000	1	s_1	001	001	0
s_1	001	–	s_2	010	011	0
s_2	010	–	s_3	011	001	0
s_3	011	–	s_4	100	111	0
s_4	100	–	s_5	101	001	0
s_5	101	–	s_6	110	011	0
s_6	110	–	s_7	111	001	0
s_7	111	–	s_0	000	111	1

[1] Present state.
[2] Present inputs.
[3] Next state.
[4] Present outputs.

STEP 3 List the next states (NS) in a column, as shown in Figure 7.6. The state diagram arrows' destinations with or without conditions yield the NS list.

STEP 4 Make one column for each flip-flop input. Select a flip-flop type (select T for this example). (*Reminder:* if JKs are used, each JK flip-flop requires two columns.) We calculate from the flip-flop defining equation the input values setting up the next state.

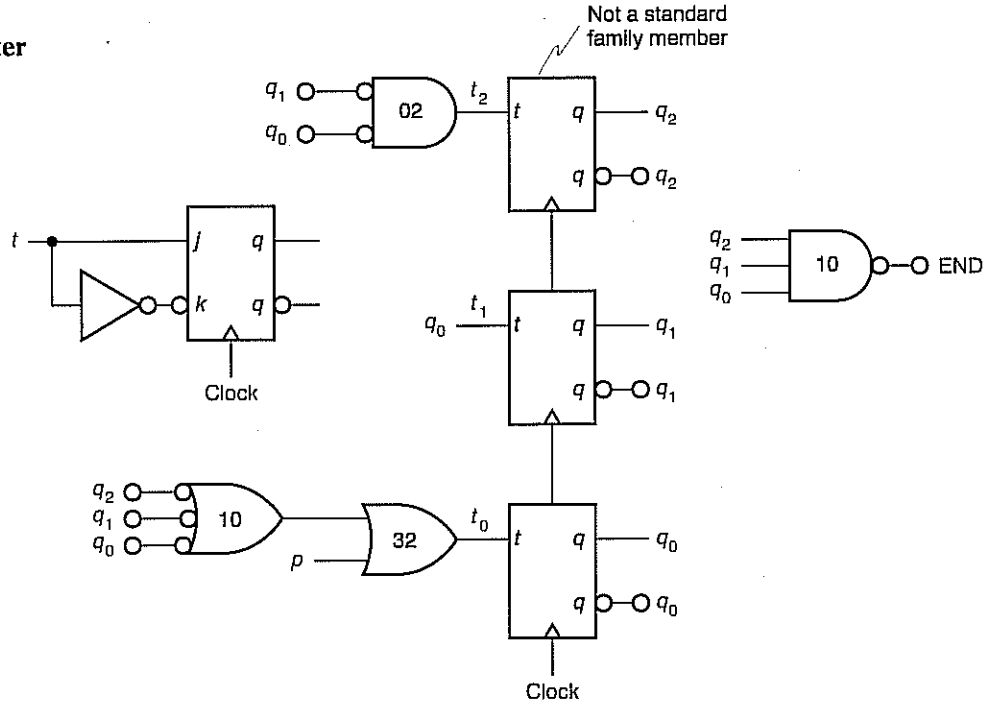
$$q^+ = tq' + t'q$$

Briefly stated, $t = 1$ toggles the flip-flop, and $t = 0$ holds the present value. Therefore if NS bit q_j^+ does not equal PS bit q_j, we let $t = 1$; and if NS bit q_j^+ equals PS bit q_j, we let $t = 0$.

STEP 5 Make one column per output variable. Mark the END column with 1 on the PS lines the output(s) is true.

STEP 6 Derive the t flip-flop input equations for $t_2 t_1 t_0$ from the state transition table. We use the present-state and associated inputs, but not the next state.

FIGURE 7.7
Three-bit counter



In the t_0 column there is only one 0, so we find the equation for t_0'.

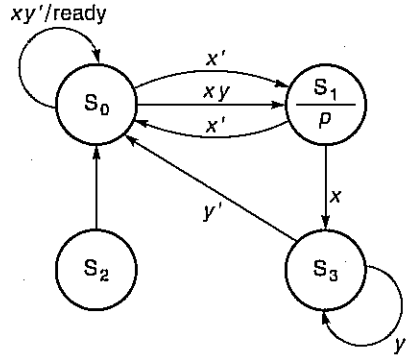
$$t_0' = m_0 p'$$

$$t_0 = m_0' + p = (q_2' q_1' q_0')' + p = q_2 + q_1 + q_0 + p$$

By inspection, $t_1 = q_0$, $t_2 = q_1 q_0$, and END $= q_2 q_1 q_0$.

STEP 7 Synthesize the state machine circuit from the flip-flop input equations. The circuit implementing the state diagram of Figure 7.5 is shown in Figure 7.7. A notation for parallel connection of the clock to flip-flop clock inputs is also illustrated in Figure 7.7. The clock wire is routed under each flip-flop.

EXAMPLE 7.1 Converting State Diagram to State Table



Four states require two state variables q_1, q_0. We enter four state identifiers in the PS column.

State s_0 has variables x and y associated with it. We enter four combinations of xy values in the x and y input columns.

State s_1 has variable x associated with it. We enter two combinations of x values in the x column and two "don't care" dashes in the y column.

State s_2 has no variables associated with it. We enter dashes in the x and y columns.

State s_3 has variable y associated with it. We enter two combinations of y values in the y column and two "don't care" dashes in the x column.

We enter the next-state numbers in the NS column. The arrows point to the next states.

The conditional ready output is asserted when the machine is in s_0 and the minterm xy' is asserted. The unconditional p output is asserted when the machine is in state s_1.

PS	PI		NS	PO	
s_j	x	y	s_j	ready	p
s_0	0	0	s_1	0	0
	0	1	s_1	0	0
	1	0	s_0	1	0
	1	1	s_1	0	0
s_1	0	–	s_0	0	1
	1	–	s_3	0	1
s_2	–	–	s_0	0	0
s_3	–	0	s_0	0	0
	–	1	s_3	0	0

EXAMPLE 7.2 Converting State Table to State Diagram

PS	PI		NS	PO	
s_j	x	y	s_j	c	u
s_0	0	–	s_2	1	0
	1	–	s_1	0	0
s_1	–	–	s_0	0	0
s_2	–	0	s_2	0	1
	–	1	s_0	0	1
s_3	–	–	s_0	0	0

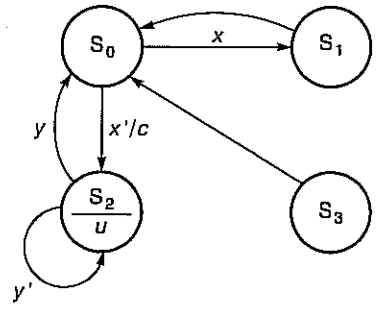
Four states require four state symbols. We draw four circles to start the diagram.

State s_0 has only variable x associated with it because y is a "don't care." We draw two arrows from s_0 to the next states s_2 and s_1. Output c is conditional, so we mark the s_0-to-s_2 arrow with x'/c and the s_0-to-s_1 arrow with x.

State s_1 has no variables associated with it. We draw one arrow from s_1 to the next state s_0.

State s_2 has only variable y associated with it because x is a "don't care." We draw two arrows from s_2 to the next states s_2 and s_0. We mark the s_2-to-s_2 arrow with y' and the s_2-to-s_0 arrow with y. Output u is unconditional, so enter u in the s_2 circle.

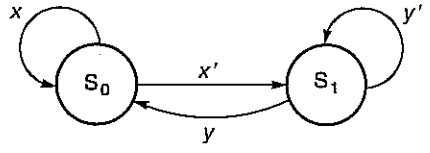
State s_3 has no variables associated with it. We draw one arrow from s_3 to the next state s_0.



EXERCISE 7.3 Find the next-state equation for the following state diagram. Use the state assignment:

(a) $s_0 = 0, s_1 = 1$ (b) $s_0 = 1, s_1 = 0$

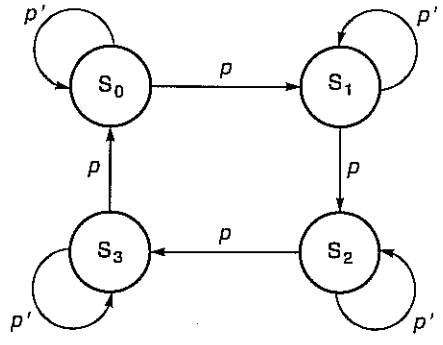
Answer: (a) $q_0^+ = q_0'x' + q_0y'$ (b) $q_0^+ = q_0'x + q_0y$



■

EXERCISE 7.4 Find the next-state equation for the following state diagram. Use the state assignments $s_0 = 00$, $s_1 = 01$, $s_2 = 10$, and $s_3 = 11$.

Answer: $q_1^+ = q_1 \text{ xor } pq_0$, $q_0^+ = q_0 \text{ xor } p$



■

7.3 ASM Chart Notation

ASM charts use three symbols: state box, branch diamond, and conditional output oval.

The algorithmic state machine method uses straightforward procedures to create reliable sequential state machines. Via the ASM method we can develop designs with style and elegance. However, the ASM method does not eliminate the need for creativity. Creativity is required to construct the basic algorithm that solves the current problem. At the same time, the ASM method is an excellent assistant during the creative phase. Furthermore, the ASM chart notation consists of only three symbols: state, branch, and conditional output.

State The rectangle is state s_j's symbol (Figure 7.8a). The symbol has only one entrance and one exit.

For a synchronous state machine each active clock transition causes a change of state from the present state to the next state. Given the present state, the next state must be determined without ambiguity for any values of state and input variables. Having arrived at the next state, this next state becomes the present state.

Whereas any number of paths may lead to a state rectangle's single entry point, only one path may lead away from the state rectangle's one and only exit point.

Branch The diamond is the branch symbol (Figure 7.8b). The extended diamond is a compact form representing a *tree* of diamonds (Figures 7.8b and 7.9b). *Decision box* is an alias for *branch diamond.*

The diamond symbol with True and False exit paths represents a decision. The condition placed in the box may be any Boolean function of input variables. The True (False) exit path is taken when the condition is True (False). The extended diamond has one exit path per minterm of the input variables involved in the decision. (These are *not* minterms of all the state machine input variables.) In Figure 7.8b, a decision box with two variables x, y has one exit for each minterm ($x'y'$, $x'y$, xy', and xy). Boolean variables such as x in a branch function (Figure 7.8d) usually represent inputs from the outside world. In addition, the inputs can originate from within the same logic machine.

Suppose y is a "don't care" when x is True. This means the two exits corresponding to minterms xy' and xy merge into one exit (Figure 7.8e). This is why the y diamond on the x = T side of the tree can be removed.

Each branch diamond has only one entry point. A branch exit path leads either to only one other branch diamond or to only one (next) state so that the next state is determined without ambiguity for any values of state and input variables. Only one of the parallel exit paths can be active if the next state is to be uniquely determined.

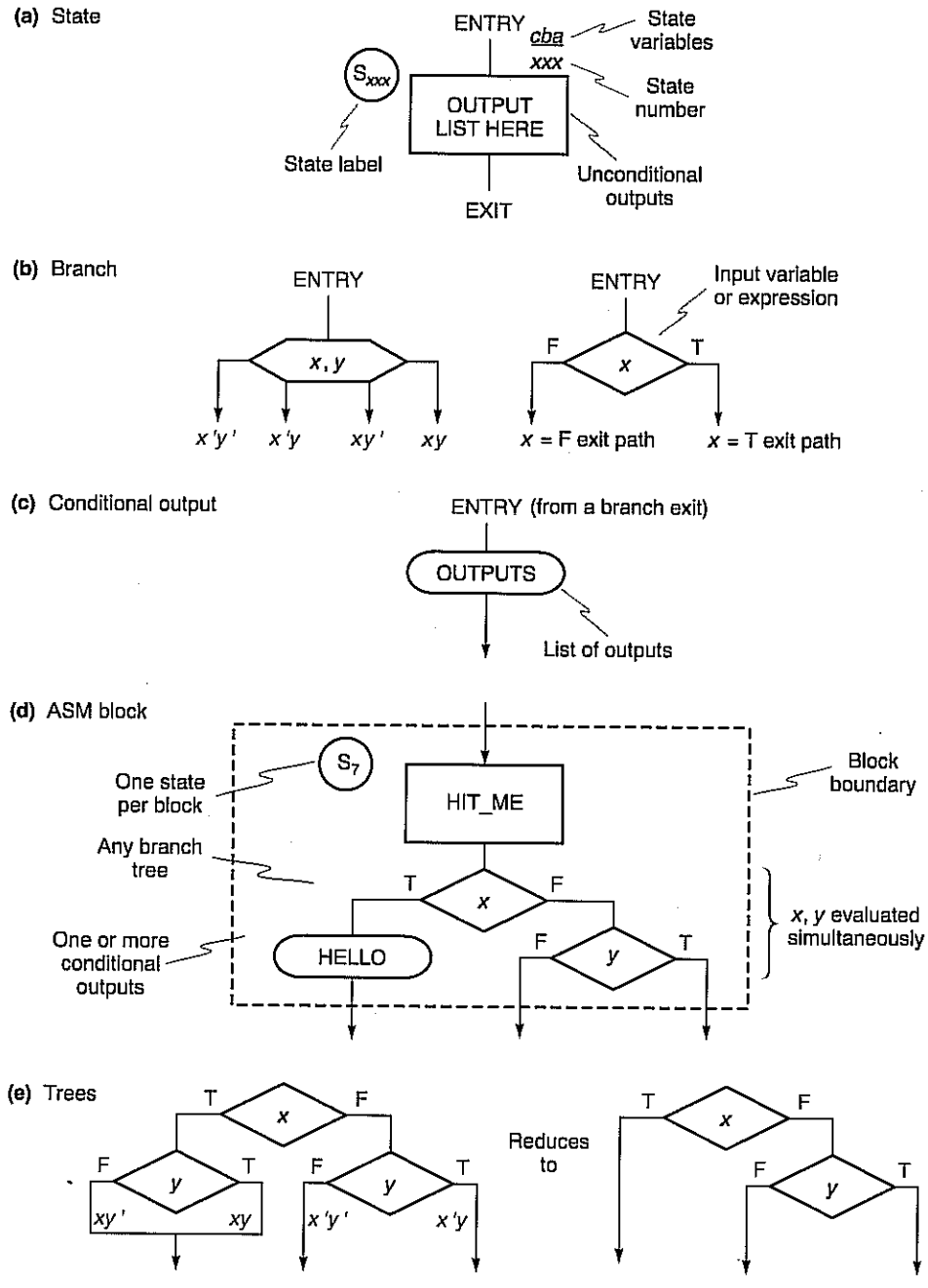
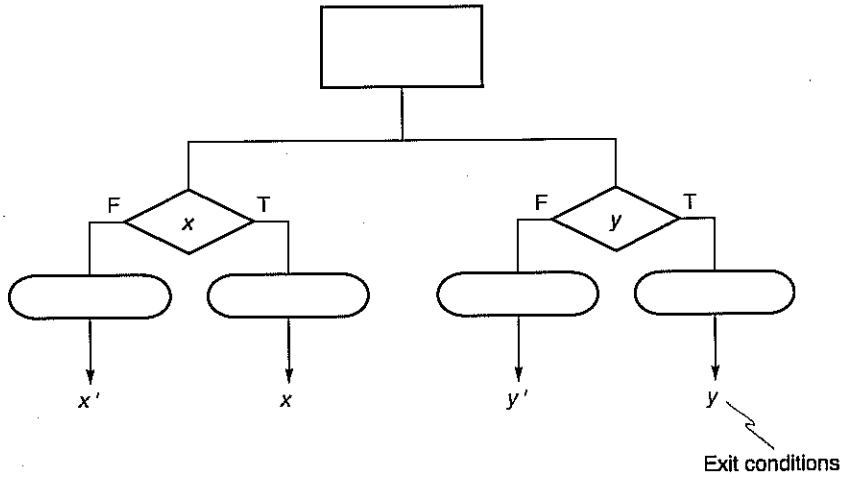
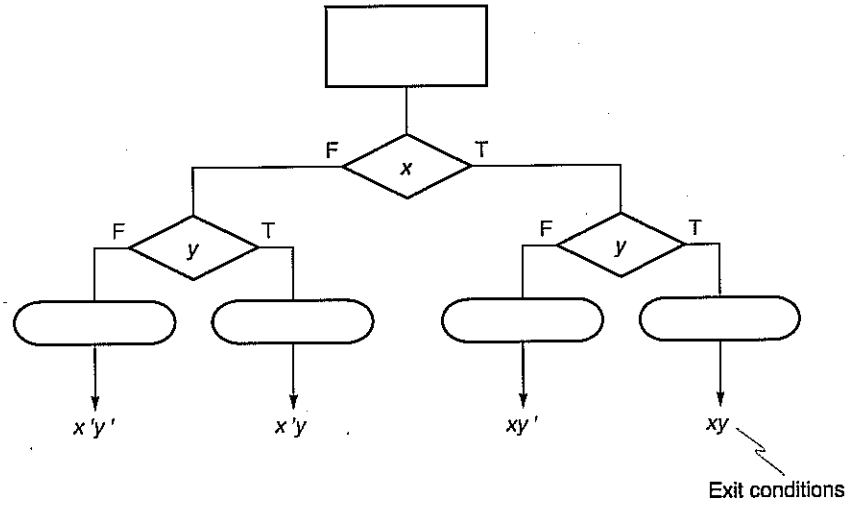


FIGURE 7.8
ASM chart notation

FIGURE 7.9
Incorrect and correct versions of an ASM binary tree



(a) Incorrect: always two simultaneous exit paths (*x* and *y* independent)



(b) Correct: always only one exit path at at time

An input variable does not have to be associated with every decision. Input variables not involved in a decision are treated as "don't cares" in that decision. For example, there may be five input variables, but only x, y are associated with some state.

A branch must be associated with a state to have meaning. There must be only one possible next state for each set of input conditions.

Fundamental Rule Every path must lead to only one state. Therefore we activate only one path at a time.

When this is not the case, the logic machine cannot select a unique next state, which is a design error (Figure 7.9a).

The branch is activated during the entire state time. *State time* is the clock period prior to the next clock edge that executes the decision. The exit path selected by the input values immediately before the clock edge occurs determines the next state.

Unconditional output One or more unconditional outputs are specified by entering the outputs' names inside the desired state rectangle (Figure 7.8a). There is no special symbol for an unconditional output; the only symbol is the name.

Unconditional outputs depend solely on the state. They do not depend on inputs in any way. Unconditional outputs are active during the associated state time. (In state diagram terms, an ASM machine with only unconditional outputs is a Moore machine.)

Note: When there are no unconditional outputs, a state is an "empty box." We do not delete the empty box from the ASM chart simply because it is empty. There may be other reasons why the state exists.

Conditional output The oval is the symbol of conditional output (Figure 7.8c). The oval is always associated with a branch diamond. Conditional output names are entered in the oval. A conditional output is active during a state time if the associated branch condition is True during that state time. The output is conditioned by the state and branch that feeds it. (In state diagram terms, an ASM machine with only conditional outputs is a Mealy machine.) In Figure 7.8d output HELLO is active when the system is in state s_7 and when x is True.

Note: When there are no conditional outputs, the oval is "empty." The empty oval is deleted because there are no other possible reasons for its presence.

ASM block An ASM block consists of one entry line, one state rectangle, and any number of associated branch diamonds, conditional output ovals, and mutually exclusive exit lines (Figure 7.8d). Any ASM chart is an assembly of ASM blocks connected by paths (see Example 7.3).

All events in an ASM block execute simultaneously.

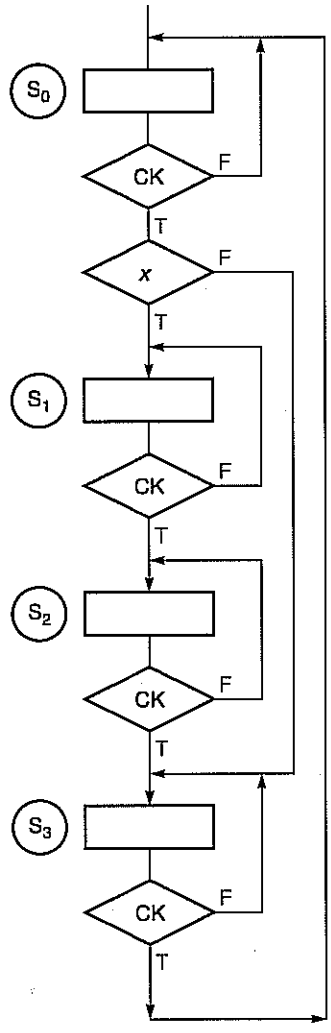
An ASM block representing state$_j$ describes the state machine's operation while the state machine is in state$_j$. The time spent in state$_j$ may be an indefinite number of clock periods when a branch diamond is associated with the state. The time spent in state$_j$ ends when the input variables close the path that returns to the present state$_j$ and open a path to a next state$_k$.

In an ASM chart the operations described in an ASM block are executed simultaneously, i.e., in parallel. In the ASM block of Figure 7.8d the following events occur during time spent in state s_7: Unconditional output HIT_ME is activated, and, in parallel, input variables x and y are evaluated simultaneously, activating the path corresponding to the $f(x, y)$ minterm that is True. When either minterm m_2 or m_3 is True, x is True. This activates conditional output HELLO. When m_0 or m_1 is true, HELLO is not activated and the $x'y'$ or $x'y$ exit is taken. Until this ASM block is embedded in an ASM chart, and the characteristics of input x are known, we do not know how many clock periods are spent in state s_7. This differs from a flow chart, where the events occur sequentially and event duration is not specified.

EXAMPLE 7.3 From Statement to ASM Chart

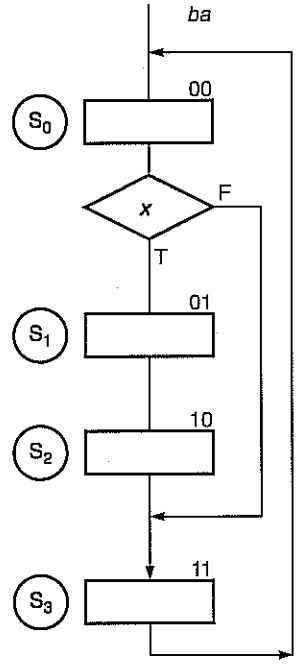
A state machine is needed to cycle through states s_0, s_1, s_2, and s_3 when input variable x is asserted, and to cycle through states s_0 and s_3 when input variable x is not asserted. The essential deduction is that s_1 and s_2 are skipped when x is not asserted. That is, from present state s_0 the next state is s_1 when x = T, and when x = F the next state is s_3. Because x is evaluated in state s_0, input variable x is active in, associated with, s_0. Furthermore, the machine steps unconditionally from s_1 to s_2, s_2 to s_3 and s_3 to s_0. The ASM chart follows.

Two ASM charts are shown [with and without clock (ck) as an input]. When clock is omitted as an input associated with every state, the implicit assumption is that the state machine is a synchronous machine. The ASM chart with clock omitted as an input to every state is not only simplified significantly, it is clear.



Note: CK = clock edge occurred

(a)



(b)

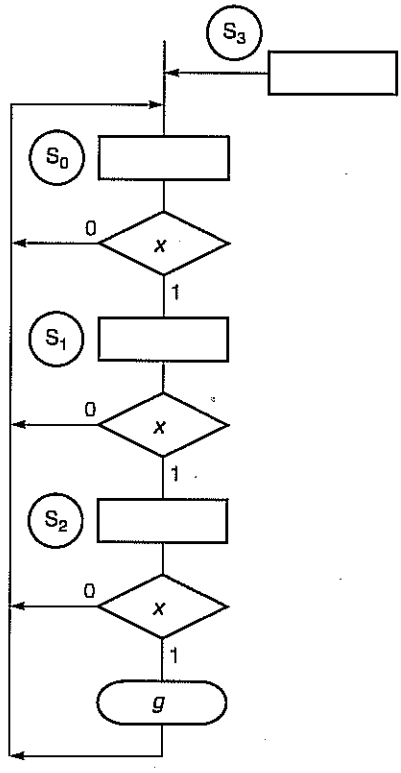
EXAMPLE 7.4 From Sequencer Statement to ASM Chart

Suppose a sequential circuit has one input x and one output g. Output g is asserted whenever the most recent inputs are 111, where the most

recent input is the last digit in the string. Overlapping of 111 sequences is not allowed, but "adjacent" 111 sequences can occur (e.g., . . . 01111110 . . . produces . . . 00010010 . . .).

The number of states required is not known at the outset. Let state s_0 be the rest state. The circuit remains in the s_0 state when zeros are received ($x = 0$). Let the machine advance to state s_1 when a one is received ($x = 1$). Therefore s_1 represents one 1 received. We draw s_0 and s_1 state boxes and a branch diamond with input x at the s_0 exit. We complete the path from s_0 to s_0 by drawing an arrow from the diamond x-equals-0 output to the s_0 state box entrance. We draw an arrow from the diamond x-equals-1 output to the s_1 state box entrance.

If a zero is received while in s_1, a return to s_0 restarts the sequence detection process. At the s_1 state box exit we draw a branch diamond with input x. We complete the path from s_1 to s_0 by drawing an arrow from the diamond x-equals-0 output to the s_0 state box entrance. If a one is received, we advance to s_2 so that s_2 represents the fact that a sequence of two 1s has been received. We draw an arrow from the s_1 diamond x-equals-1 output to the s_2 state box entrance.

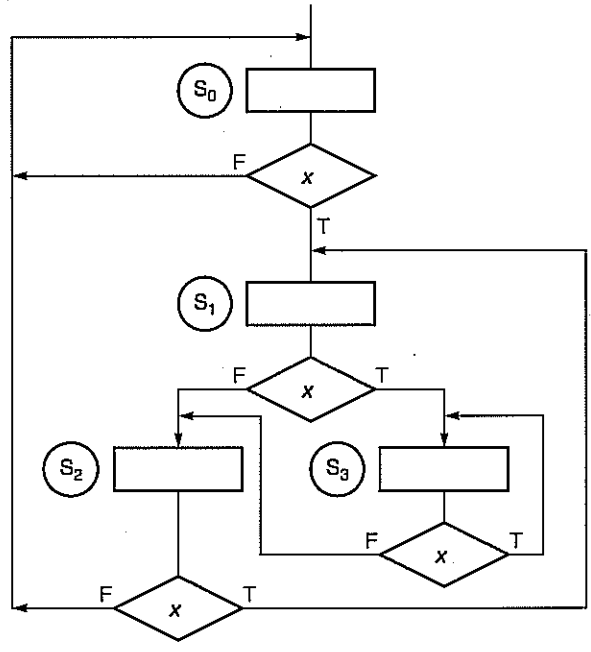


If a zero is received while in s_2, a return to s_0 restarts the sequence detection process. At the s_2 state box exit, we draw a branch diamond with input x. We complete the path from s_2 to s_0 by drawing an arrow from the diamond x-equals-0 output to the s_0 state box entrance. If a third one is received, we output a one ($g = 1$) on return to s_0, which then represents that a sequence of three 1s has been received. We draw an arrow from the s_2 diamond x-equals-1 output to an oval's entrance. We enter g in the oval. This will output the one report. We draw an arrow from the oval's exit to the s_0 state box entrance.

When three state numbers are encoded with two binary digits, a fourth state (s_3) is possible. Good practice* dictates that all unused states have a next state by design. The "rest" state s_0 is a practical next state for unused states. So we add a state box for unused state s_3. We draw an arrow from the s_3 box exit to the s_0 state box entrance.

EXERCISE 7.5 Derive the ASM chart for Figure P6.2 on page 326.

Answer:



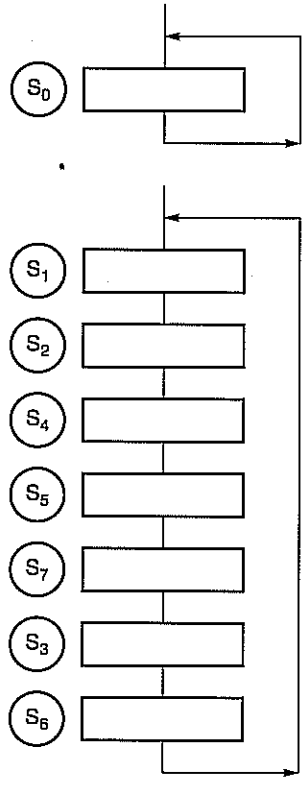
■

---

* s_3 can be entered when the power is turned on, in which case the next clock edge moves the machine to s_0 so that it is ready to execute the algorithm correctly.

EXERCISE 7.6 Derive the ASM chart for Figure P6.6 on page 327.

Answer:



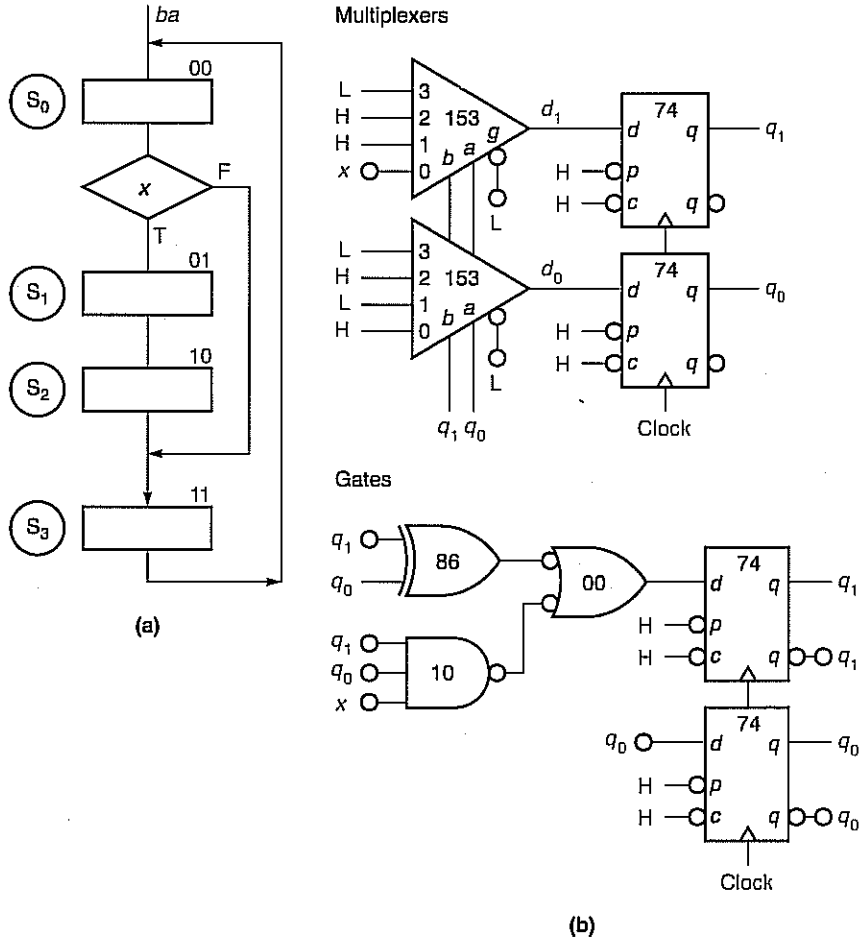
■

EXAMPLE 7.5 From ASM Chart to State Machine

A state machine is needed to cycle through states s_0, s_1, s_2, and s_3 when input variable x is asserted, and to cycle through states s_0 and s_3 when input variable x is not asserted. The ASM chart for this application was developed in Example 7.3. The PS-Input-NS truth table is derived from the ASM chart via a process similar to the state machine synthesis process described in Section 7.2.3. The complete synthesis process, from ASM to circuit, is described in Section 7.5.

PS s_j	q_1q_0	PI x	NS s_j	$d_1d_0 = q_1^+q_0^+$
s_0	00	0	s_3	11
		1	s_1	01
s_1	01	–	s_2	10
s_2	10	–	s_3	11
s_3	11	–	s_0	00

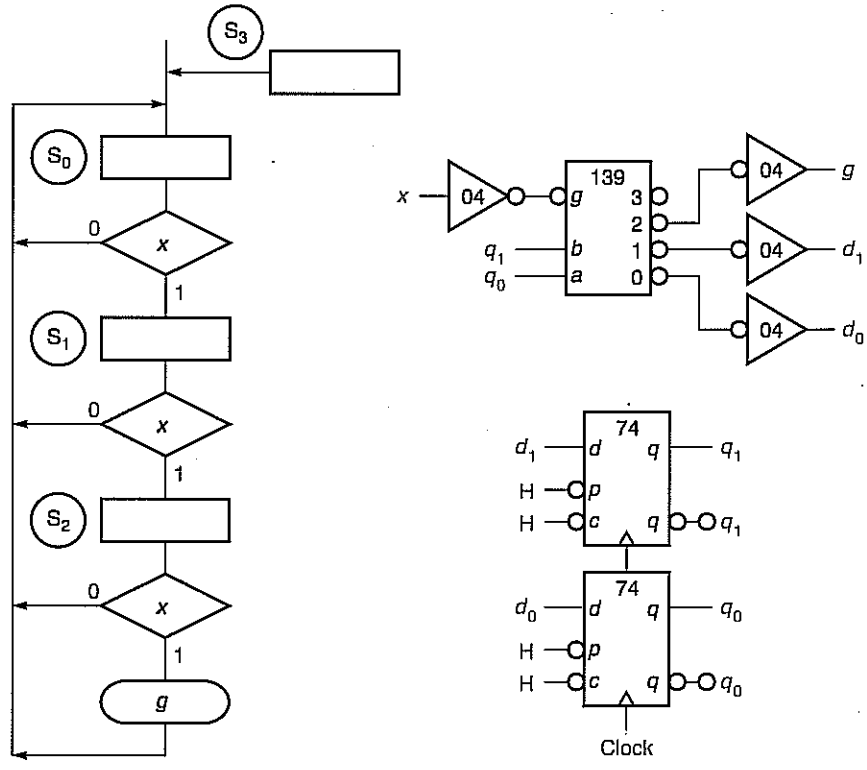
$$d_1 = x's_0 + s_1 + s_2 = x'q_1'q_0' + q_1 \text{ xor } q_0$$
$$d_0 = s_0 + s_2 = q_1'q_0' + q_1q_0' = q_0'$$



EXAMPLE 7.6 From ASM Chart to Sequencer State Machine

A sequential circuit has one input x and one output g. Output g is asserted whenever the most recent inputs are 111, where the most recent input is the last digit in the string. Overlapping of 111 sequences is not allowed. This sequencer ASM chart was developed in Example 7.4.

PS		PI	NS		PO
s_j	$q_1 q_0$	x	s_j	$d_1 d_0 = q_1^+ q_0^+$	g
s_0	00	0	s_0	00	0
		1	s_1	01	0

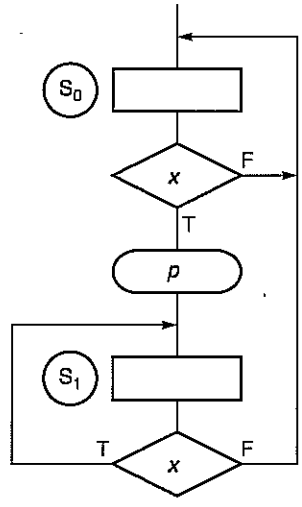


(a) (b)

PS		PI	NS		PO
s_j	$q_1 q_0$	x	s_j	$d_1 d_0 = q_1^+ q_0^+$	g
s_1	01	0	s_0	00	0
		1	s_2	10	0
s_2	10	0	s_0	00	0
		1	s_0	00	1
s_3	11	–	s_0	00	0

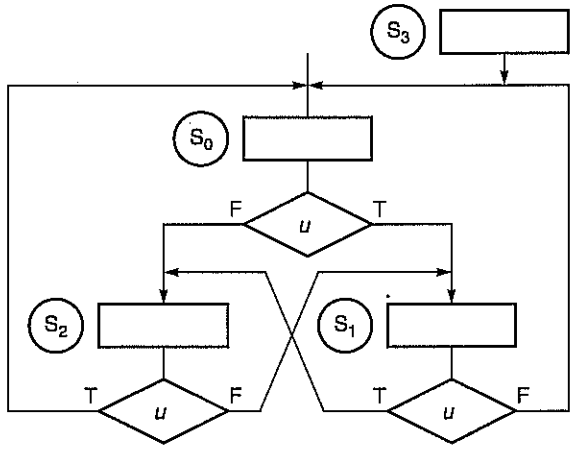
$$d_1 = xs_1 = xq_1'q_0$$
$$d_0 = xs_0 = xq_1'q_0'$$
$$g = xs_2 = xq_1q_0'$$

EXERCISE 7.7 Derive the next-state equation and the output equation for the following ASM chart (for state assignments $s_0 = 0$, $s_1 = 1$).



Answer: $q_0^+ = x, \quad p = xq_0'$ ■

EXERCISE 7.8 Derive the next-state equations and the output equation for the following ASM chart ($s_0 = 00$, $s_1 = 01$, $s_2 = 10$, $s_3 = 11$).



Answer: $q_1^+ = q_1'(q_0 \text{ xor } u')$, $q_0^+ = q_0'(q_1 \text{ xor } u)$ ■

7.4 ASM Sequences of States

Sequences of states are determined by input minterms.

The purpose of any state machine is execution of one or more sequences of states in response to various combinations of input variable values. Each combination of values sets up paths in the chart; paths determine a sequence of states.[1] (The minterms of a function of the input variables represent combinations of values.) In principle there is one sequence of states for each constant combination of values (minterm) that executes in, say, n clock periods[2] ($n\tau$). For a fixed set of input values there corresponds a sequence of states determined by the paths set up by those input values.

When the input minterm is held constant, the state machine executes the corresponding sequence of states either once or with period $n\tau$, depending on the ASM algorithm. The input minterm is held constant when there are modes of operation. For example, one minterm sets up the machine to generate a periodic clock waveform; another minterm sets up the machine to generate a single pulse whenever the single-pulse input is activated. Then there is the infinite variety of sequences of states generated by an infinite number of sequences of input minterms. For example, the sequence of minterms changes the

[1] In Example 7.3 on page 352 the sequence of states is $s_0s_1s_2s_3$ when input x is True, and s_0s_3 when x is False.

[2] In Example 7.3, $n = 4$ when input x is True, and $n = 2$ when x is False.

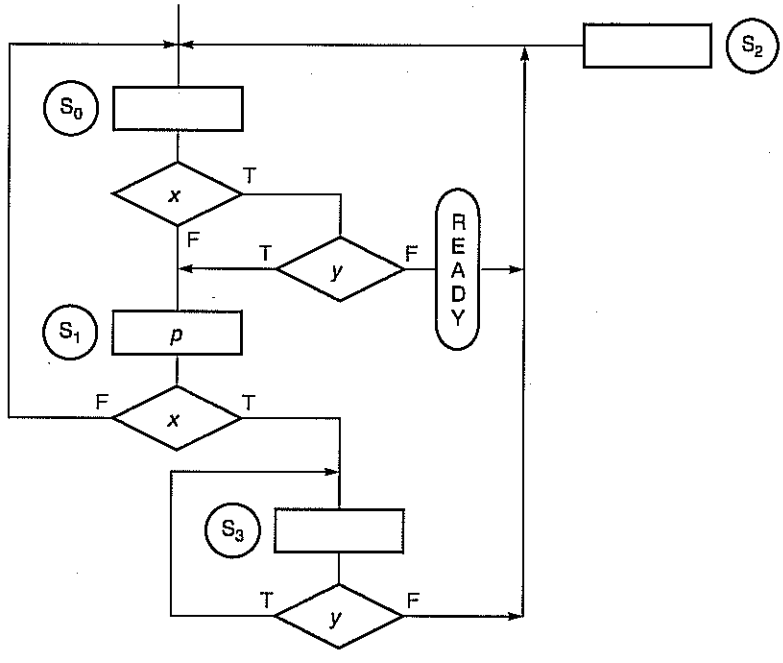
operating mode from single-pulse to periodic clock generator, and vice versa.

In many cases a state machine is designed to perform some set of tasks, where each task is selected by an input minterm. Then various sequences of input minterms are used to create various combinations of tasks in time sequence. The clock generator is one example: switch to periodic clock, switch to single pulse for testing, switch back to periodic clock, and so forth. This switching back and forth can be done manually. On the other hand, a memory module controller is one example where tasks are selected by electronic means in rapid-fire order: read, read, write, nop ("no operation"), read, write, write, and so forth.

Input minterms held constant Two variables x and y are inputs to the ASM chart in Figure 7.10. A function of two variables has four minterms. This implies that the chart generates four sequences of states when input minterms are held constant. In this case two of the sequences are identical for the paths set up by the minterms $x'y$ and $x'y'$. Starting from s_0 we trace the paths when x = F to verify this statement; notice how the branch diamonds with input y are bypassed. Tracing a path raises the *when* and *which* questions regarding state-to-state moves.

Input minterms select which state is the next state.

FIGURE 7.10
ASM chart



Active clock edges determine when a move to the next state occurs.

A synchronous state machine moves from any present state to the next state when a clock edge occurs. Clock edges determine *when* moves occur. This is why sequences of states are intimately related to timing diagrams. Input variables determine *which* state is the next state to move to from the present state. A path from state s_j to s_k may have no branch diamonds in it, in which case the next-state decision is independent of the input variables. (A path from state s_j to s_k with no branch diamonds in it may be thought of as a path with a full binary tree of x and y diamonds, as in Figure 7.9b, with all exits leading to the same state.)

Producing timing diagrams from ASM charts with input minterms held constant Here is a straightforward process leading to the sequences of states for input terms $x'(= x'y' + x'y)$, xy', and xy.

$x = F\ (x' = T)$ Let us assume the machine is resting in state zero (s_0) of the ASM chart in Figure 7.10. Observe that the input variable x is associated with s_0. With x = F, the path from s_0 to s_1 is active; therefore the clock edge ending cycle 1 (Figure 7.11) moves the machine from s_0 to s_1. We note that x is also associated with s_1. Input x is still False; thus the clock edge ending cycle 2 moves the machine from s_1 to s_0. Clearly, input variable y has no role in this particular repetitive sequence consisting of s_0s_1 cycles. Furthermore, each time the ASM is in s_1, the output variable p is True (Figure 7.11). This is the periodic mode.

$x = T, y = T\ (xy = T)$ Again, let us assume the machine is resting in s_0 of the ASM chart in Figure 7.10. With x = T and y = T, another path from s_0 to s_1 is active, so that the clock edge ending cycle 1 moves the machine from s_0 to s_1. The clock edge ending cycle 2 moves the machine from s_1 to s_3 because x = T. Input variable y = T activates the path from the s_3 exit to the s_3 entrance. This is how the state machine finds itself stuck in s_3. Output variable p is True for only one clock period in this sequence of states (Figure 7.12).

FIGURE 7.11
Timing diagram when $x = F(x' = T)$

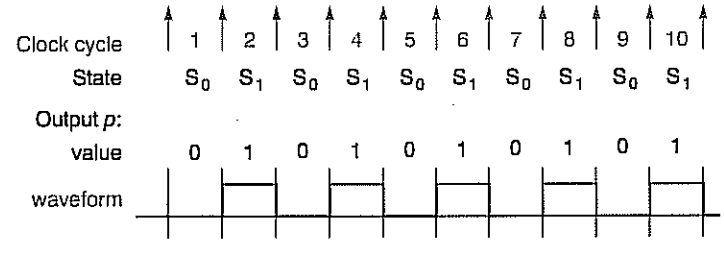
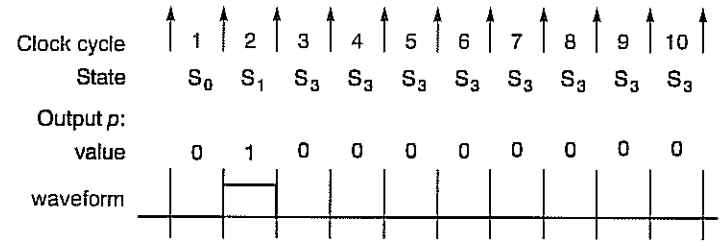


FIGURE 7.12
Timing diagram when x = T, y = T (xy = T)



x = T, y = F (xy' = T) Let's assume the machine is resting in s_3 of the ASM in Figure 7.10 with y = T. When y switches to F in cycle 2, the machine moves from s_3 to s_0 at the next clock edge. Now, with x = T and y = F, yet another path from s_0 is active; this time the path leads back to s_0, so that the clock edge ending cycle 3 moves the machine from s_0 to s_0. This is how the state machine finds itself stuck in s_0 with the ready output active. The machine does not pass through s_1, so output p is never True in this sequence of states (Figure 7.13).

The state machine is actually standing by reporting it is "ready" to perform when xy' = T. If y switches from F to T, output p is active for one clock period (Figure 7.12) and the machine dwells in s_3 until y switches back to F. A merger of the two sequences gives them purpose.

x = T, y = F, T, F Now let's assume the machine is resting in s_0 of the ASM in Figure 7.10 with x = T. With y = F the machine repeatedly cycles from s_0 to s_0 at each clock edge. This is how we found the machine apparently stuck in s_0. We say "apparently" because there is

FIGURE 7.13
Timing diagram when x = T, y = F (xy' = T)

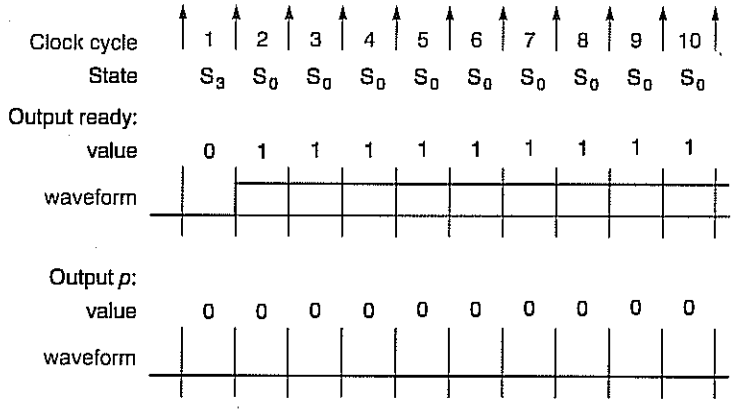
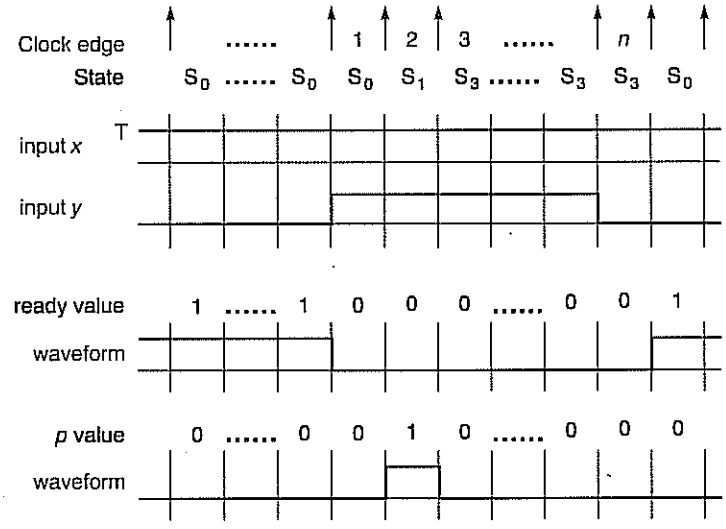


FIGURE 7.14
Timing diagram for single-pulse sequence of states



Indefinite time intervals marked as .....

another interpretation: the machine is in the ready-to-perform state when x = T and y = F. In Figure 7.12, when x = T and y = T we see that the output p is what is called a *single pulse*.

If we make y = T for some (indefinite) amount of time, then the F-T pair generates the $s_0s_1s_3$ sequence in cycles 1, 2, 3, followed by the dwell-in-s_3 sequence. When y switches back to the final F in cycle n, this allows the machine to return to s_0 to wait for the next y = T input. This is the *single-pulse mode*.

EXERCISE 7.9 What is the sequence of states in Figure P7.2 on page 400 when input minterm $x = 1$?

Answer: $s_0s_1s_1 \ldots$ ■

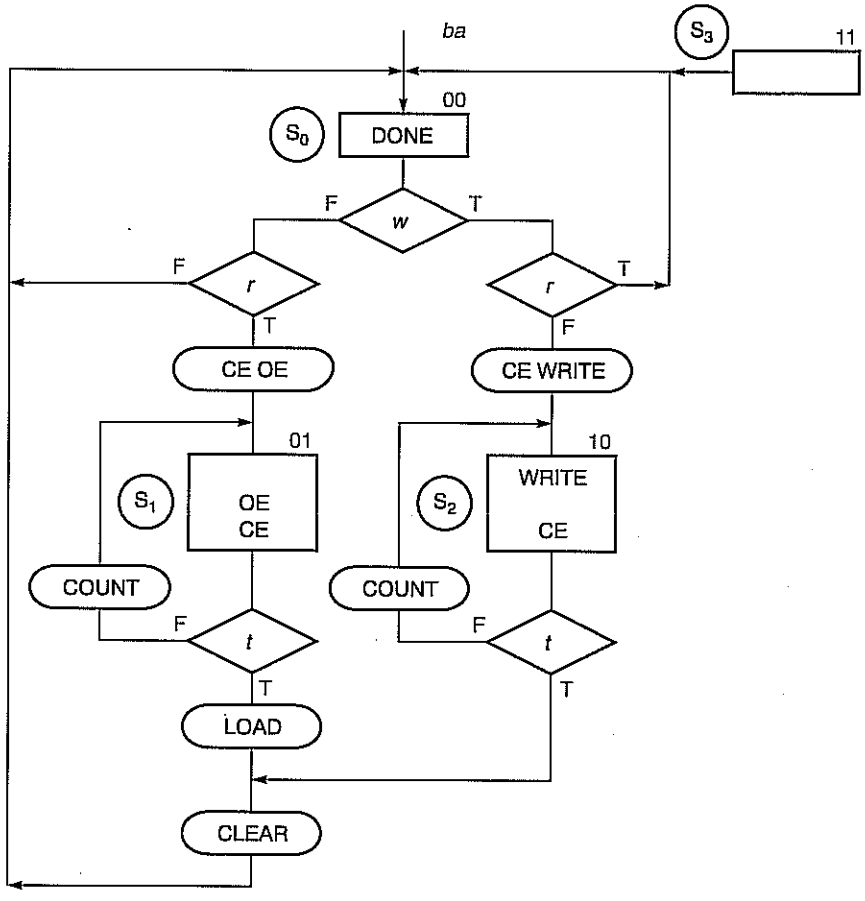
EXERCISE 7.10 What is the sequence of states in Figure P7.6 on page 406 when input minterm $x'y = 1$?

Answer: $s_0s_1s_2s_3s_0 \ldots$ ■

EXERCISE 7.11 What is the sequence of states in Figure 7.15 when input minterm $wr't = 1$?

Answer: $s_0s_2s_0s_2s_0 \ldots$ ■

FIGURE 7.15
Memory module ASM chart

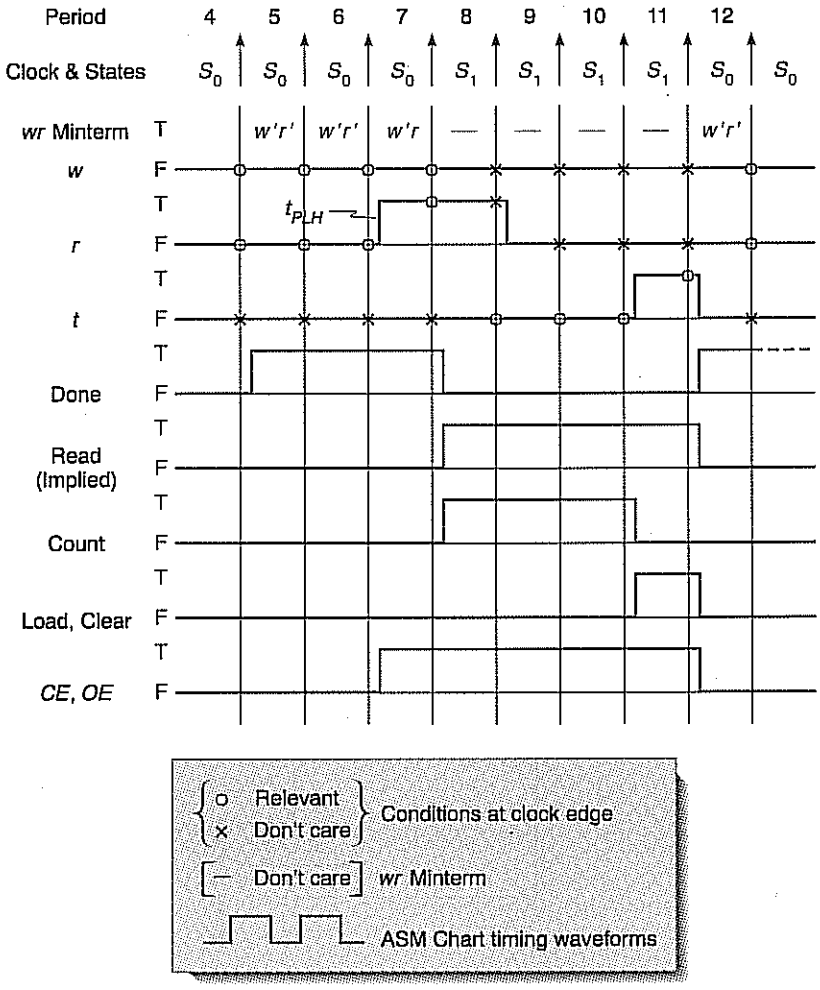


Sequence of input minterms To illustrate how we can determine the sequence of states corresponding to a sequence of input minterms, let us consider the ASM chart in Figure 7.15. The ASM chart represents a controller for a dynamic RAM system (discussed in detail in Chapter 9). For now we consider it a black box with three inputs (w, r, t) and a number of outputs (DONE, CE, OE, . . .). We assume the physical implementation is a synchronous sequential machine. Then each positive clock edge is a time when change to the next state takes place.

Input minterms implement branch decisions.

The timing diagram in Figure 7.16 shows inputs w, r, and t as functions of time. This is a time sequence of input minterms. With values for inputs r and w held constant, some path from the s_0 exit to the entrance of s_0 or s_1 or s_2 is active. On the ASM chart when min-

FIGURE 7.16
ASM chart sequence of states to read a memory



term $w'r'$ is True ($w = r =$ F), the active path is now from s_0 through the branch diamond tree to the s_0 entrance (Figure 7.15). Thus s_0 is the next state. Clock edge 6 ends period 6 as it sets the system to the next state s_0.

In some other part of the system we assume clock edge 6 also switches r to True after a small time delay t_{PLH} (see period 7). Now minterm $w'r$ is True ($w =$ F, $r =$ T).

At clock edge 7, the ASM is still in s_0, so that input t is a "don't care." With minterm $w'r =$ True, the ASM moves to state s_1 on clock edge 7 (Figure 7.16). In state s_1 input variable t is active and inputs w, r are "don't cares." At the end of periods 8, 9, and 10 the t input is