Of course, asynchronous circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages. Furthermore, research in asynchronous circuits continues to generate interesting insights, some of which can improve synchronous circuits too.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.22. These forms are called *finite state machines* (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two general classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the outputs depend only on the current state of the machine. In *Mealy machines*, the outputs depend on both the current state and the current inputs. Finite state machines provide a systematic way to design synchronous sequential circuits given a functional specification. This method will be explained in the remainder of this section, starting with an example.

Moore and Mealy machines are named after their promoters, researchers who developed *automata theory*, the mathematical underpinnings of state machines, at Bell Labs.

Edward F. Moore (1925–2003), not to be confused with Intel founder Gordon Moore, published his seminal article, *Gedanken-experiments on Sequential Machines* in 1956. He subsequently became a professor of mathematics and computer science at the University of Wisconsin.

George H. Mealy published *A Method of Synthesizing Sequential Circuits* in 1955. He subsequently wrote the first Bell Labs operating system for the IBM 704 computer. He later joined Harvard University.

3.4.1 FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic Ave. They are busy reading about FSMs in their favorite

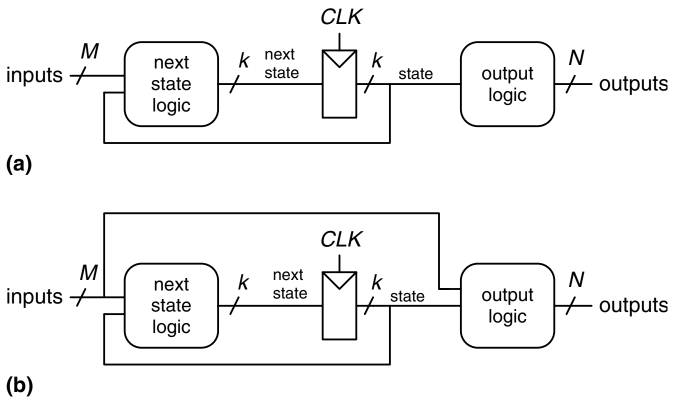


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

textbook and aren't looking where they are going. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. They are tossing the ball back and forth and aren't looking where they are going either. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install a traffic light before there are fatalities.

Ben decides to solve the problem with an FSM. He installs two traffic sensors, T_A and T_B, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs two traffic lights, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.23. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides a reset button so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.24 shows a black box view of the state machine.

Ben's next step is to sketch the *state transition diagram*, shown in Figure 3.25, to indicate all the possible states of the system and the transitions between these states. When the system is reset, the lights are green on Academic Ave. and red on Bravado Blvd. Every 5 seconds, the controller examines the traffic pattern and decides what to do next. As long

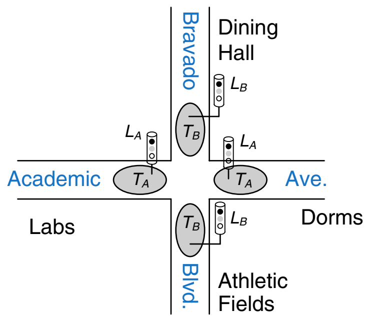


Figure 3.23 Campus map

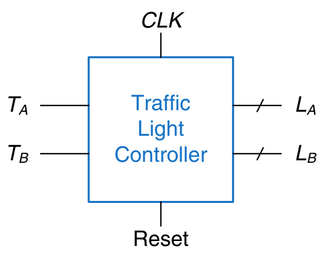


Figure 3.24 Black box view of finite state machine

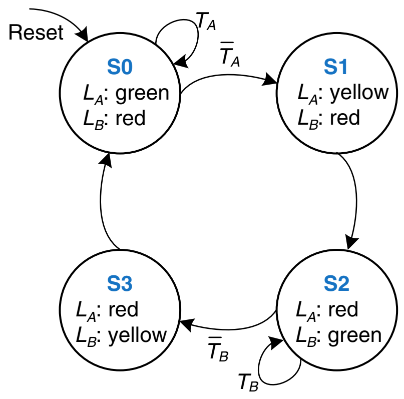


Figure 3.25 **State transition diagram**

as traffic is present on Academic Ave., the lights do not change. When there is no longer traffic on Academic Ave., the light on Academic Ave. becomes yellow for 5 seconds before it turns red and Bravado Blvd.'s light turns green. Similarly, the Bravado Blvd. light remains green as long as traffic is present on the boulevard, then turns yellow and eventually red.

In a state transition diagram, circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur. The arc labeled Reset pointing from outer space into state S0 indicates that the system should enter that state upon reset, regardless of what previous state it was in. If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. For example, when in state S0, the system will remain in that state if T_A is TRUE and move to S1 if T_A is FALSE. If a state has a single arc leaving it, that transition always occurs regardless of the inputs. For example, when in state S1, the system will always move to S2. The value that the outputs have while in a particular state are indicated in the state. For example, while in state S2, L_A is red and L_B is green.

Ben rewrites the state transition diagram as a *state transition table* (Table 3.1), which indicates, for each state and input, what the next state, S', should be. Note that the table uses don't care symbols (X) whenever the next state does not depend on a particular input. Also note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

The state transition diagram is abstract in that it uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}. To build a real circuit, the states and outputs must be assigned *binary encodings*. Ben chooses the simple encodings given in Tables 3.2 and 3.3. Each state and each output is encoded with two bits: $S_{1:0}$, $L_{A1:0}$, and $L_{B1:0}$.

Table 3.1 State transition table

Current State S	Inputs T_A	T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Ben updates the state transition table to use these binary encodings, as shown in Table 3.4. The revised state transition table is a truth table specifying the next state logic. It defines next state, S', as a function of the current state, S, and the inputs. The revised output table is a truth table specifying the output logic. It defines the outputs, L_A and L_B, as functions of the current state, S.

From this table, it is straightforward to read off the Boolean equations for the next state in sum-of-products form.

$$\begin{aligned} S_1' &= \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B \\ S_0' &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.1}$$

The equations can be simplified using Karnaugh maps, but often doing it by inspection is easier. For example, the T_B and $\overline{T}_B$ terms in the S'_1 equation are clearly redundant. Thus S'_1 reduces to an XOR operation. Equation 3.2 gives the *next state equations*.

Table 3.4 State transition table with binary encodings

Current State S_1	S_0	Inputs T_A	T_B	Next State S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 **Output table**

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned} S_1' &= S_1 \oplus S_0 \\ S_0' &= \overline{S}_1\overline{S}_0\overline{T}_A + S_1\overline{S}_0\overline{T}_B \end{aligned} \tag{3.2}$$

Similarly, Ben writes an *output table* (Table 3.5) indicating, for each state, what the output should be in that state. Again, it is straightforward to read off and simplify the Boolean equations for the outputs. For example, observe that L_{A1} is TRUE only on the rows where S_1 is TRUE.

$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S}_1 S_0 \\ L_{B1} &= \overline{S}_1 \\ L_{B0} &= S_1 S_0 \end{aligned} \tag{3.3}$$

Finally, Ben sketches his Moore FSM in the form of Figure 3.22(a). First, he draws the 2-bit state register, as shown in Figure 3.26(a). On each clock edge, the state register copies the next state, $S'_{1:0}$, to become the state, $S_{1:0}$. The state register receives a synchronous or asynchronous reset to initialize the FSM at startup. Then, he draws the next state logic, based on Equation 3.2, which computes the next state, based on the current state and inputs, as shown in Figure 3.26(b). Finally, he draws the output logic, based on Equation 3.3, which computes the outputs based on the current state, as shown in Figure 3.26(c).

Figure 3.27 shows a timing diagram illustrating the traffic light controller going through a sequence of states. The diagram shows *CLK*, Reset, the inputs T_A and T_B, next state S', state S, and outputs L_A and L_B. Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change. Dashed lines indicate the rising edge of *CLK* when the state changes.

The clock has a 5-second period, so the traffic lights change at most once every 5 seconds. When the finite state machine is first turned on, its state is unknown, as indicated by the question marks. Therefore, the system should be reset to put it into a known state. In this timing diagram, This schematic uses some AND gates with bubbles on the inputs. They might be constructed with AND gates and input inverters, with NOR gates and inverters for the non-bubbled inputs, or with some other combination of gates. The best choice depends on the particular implementation technology.

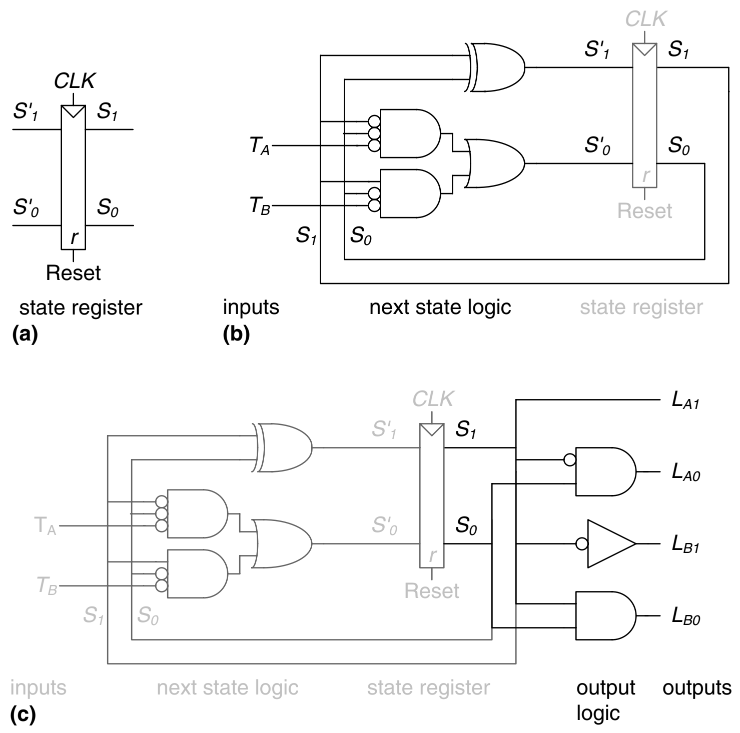


Figure 3.26 **State machine circuit for traffic light controller**

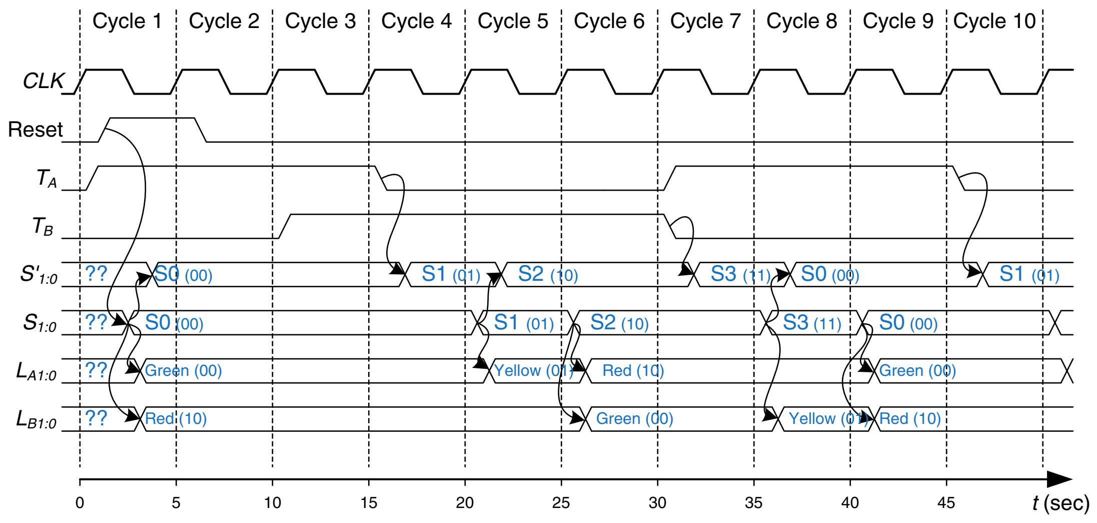


Figure 3.27 **Timing diagram for traffic light controller**

S immediately resets to S0, indicating that asynchronously resettable flip-flops are being used. In state S0, light L_A is green and light L_B is red.

In this example, traffic arrives immediately on Academic Ave. Therefore, the controller remains in state S0, keeping L_A green even though traffic arrives on Bravado Blvd. and starts waiting. After 15 seconds, the traffic on Academic Ave. has all passed through and T_A falls. At the following clock edge, the controller moves to state S1, turning L_A yellow. In another 5 seconds, the controller proceeds to state S2 in which L_A turns red and L_B turns green. The controller waits in state S2 until all the traffic on Bravado Blvd. has passed through. It then proceeds to state S3, turning L_B yellow. 5 seconds later, the controller enters state S0, turning L_B red and L_A green. The process repeats.

Despite Ben's best efforts, students don't pay attention to traffic lights and collisions continue to occur. The Dean of Students next asks him to design a catapult to throw engineering students directly from their dorm roofs through the open windows of the lab, bypassing the troublesome intersection all together. But that is the subject of another textbook.

3.4.2 State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. Unfortunately, there is no simple way to find the best encoding except to try all possibilities, which is infeasible when the number of states is large. However, it is often possible to choose a good encoding by inspection, so that related states or outputs share bits. Computer-aided design (CAD) tools are also good at searching the set of possible encodings and selecting a reasonable one.

One important decision in state encoding is the choice between binary encoding and one-hot encoding. With *binary encoding*, as was used in the traffic light controller example, each state is represented as a binary number. Because K binary numbers can be represented by $\log_2 K$ bits, a system with K states only needs $\log_2 K$ bits of state.

In *one-hot encoding*, a separate bit of state is used for each state. It is called one-hot because only one bit is "hot" or TRUE at any time. For example, a one-hot encoded FSM with three states would have state encodings of 001, 010, and 100. Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required. The best encoding choice depends on the specific FSM.

Example 3.6 FSM STATE ENCODING

A *divide-by-N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and one-hot state encodings.

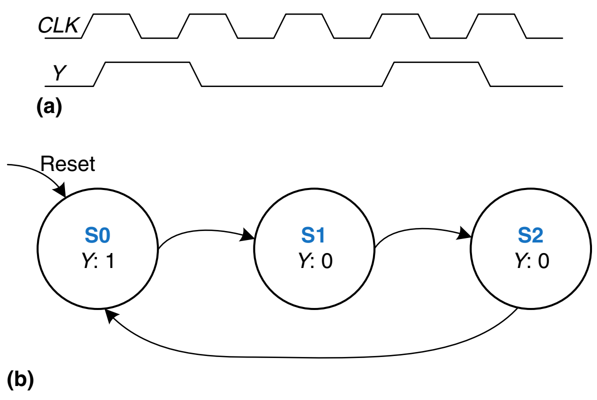


Figure 3.28 **Divide-by-3 counter (a) waveform and (b) state transition diagram**

Table 3.6 **Divide-by-3 counter state transition table**

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 **Divide-by-3 counter output table**

Current State	Output
S0	1
S1	0
S2	0

Solution: Tables 3.6 and 3.7 show the abstract state transition and output tables before encoding.

Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next-state and output equations are:

$$\begin{aligned} S'_1 &= \overline{S}_1 S_0 \\ S'_0 &= \overline{S}_1 \overline{S}_0 \end{aligned} \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next-state and output equations are as follows:

$$\begin{aligned} S'_2 &= S_1 \\ S'_1 &= S_0 \\ S'_0 &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

Figure 3.29 shows schematics for each of these designs. Note that the hardware for the binary encoded design could be optimized to share the same gate for Y and S'_0. Also observe that the one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset. The best implementation choice depends on the relative cost of gates and flip-flops, but the one-hot design is usually preferable for this specific example.

A related encoding is the *one-cold* encoding, in which K states are represented with K bits, exactly one of which is FALSE.

Table 3.8 Binary and one-hot encodings for divide-by-3 counter

State	Binary Encoding S_2	S_1	S_0	One-Hot Encoding S_1	S_0
S0	0	0	1	0	1
S1	0	1	0	1	0
S2	1	0	0	0	0

Table 3.9 State transition table with binary encoding

Current State S_1	S_0	Next State S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Table 3.10 State transition table with one-hot encoding

Current State S_2	S_1	S_0	Next State S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

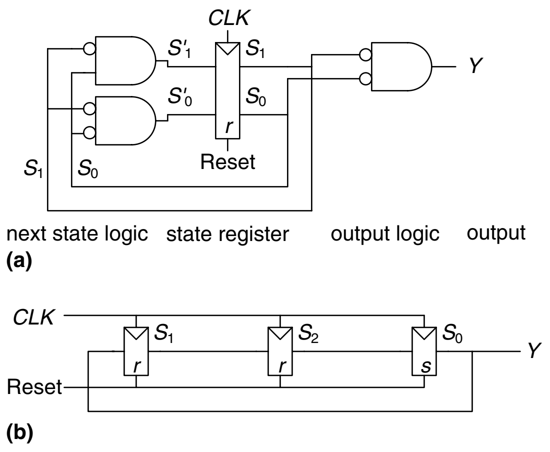
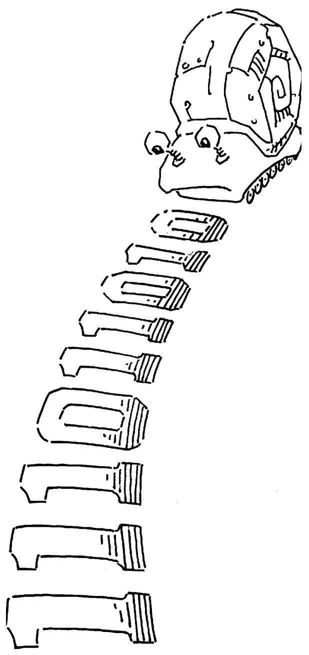


Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

3.4.3 Moore and Mealy Machines

An easy way to remember the difference between the two types of finite state machines is that a Moore machine typically has *more* states than a Mealy machine for a given problem.

So far, we have shown examples of Moore machines, in which the output depends only on the state of the system. Hence, in state transition diagrams for Moore machines, the outputs are labeled in the circles. Recall that Mealy machines are much like Moore machines, but the outputs can depend on inputs as well as the current state. Hence, in state transition diagrams for Mealy machines, the outputs are labeled on the arcs instead of in the circles. The block of combinational logic that computes the outputs uses the current state and inputs, as was shown in Figure 3.22(b).



Example 3.7 MOORE VERSUS MEALY MACHINES

Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last four bits that it has crawled over are, from left to right, 1101. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as your snail crawls along the sequence 111011010.

Solution: The Moore machine requires five states, as shown in Figure 3.30(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S4 to S2 when the input is 1?

In comparison, the Mealy machine requires only four states, as shown in Figure 3.30(b). Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.

Tables 3.11 and 3.12 show the state transition and output tables for the Moore machine. The Moore machine requires at least three bits of state. Consider using a binary state encoding: S0 = 000, S1 = 001, S2 = 010, S3 = 011, and S4 = 100. Tables 3.13 and 3.14 rewrite the state transition and output tables with these encodings (These four tables follow on page 128).

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that states 101, 110, and 111 do not exist. Thus, the corresponding next state and output for the nonexistent states are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$\begin{aligned} S_2' &= S_1 S_0 A \\ S_1' &= \overline{S}_1 S_0 A + S_1 \overline{S}_0 + S_2 A \\ S_0' &= \overline{S}_2 \overline{S}_1 \overline{S}_0 A + S_1 \overline{S}_0 \overline{A} \end{aligned} \tag{3.8}$$

$$Y = S_2 \tag{3.9}$$

Table 3.15 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires at least two bits of state. Consider using a binary state encoding: S0 = 00, S1 = 01, S2 = 10, and S3 = 11. Table 3.16 rewrites the state transition and output table with these encodings.

From these tables, we find the next state and output equations by inspection.

$$\begin{aligned} S_1' &= S_1\overline{S}_0 + \overline{S}_1 S_0 A \\ S_0' &= \overline{S}_1\overline{S}_0 A + S_1\overline{S}_0\overline{A} + S_1 S_0 A \end{aligned} \tag{3.10}$$

$$Y = S_1 S_0 A \tag{3.11}$$

The Moore and Mealy machine schematics are shown in Figure 3.31(a) and 3.31(b), respectively.

The timing diagrams for the Moore and Mealy machines are shown in Figure 3.32 (see page 131). The two machines follow a different sequence of states. Moreover, the Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output. When choosing your FSM design style, consider when you want your outputs to respond.

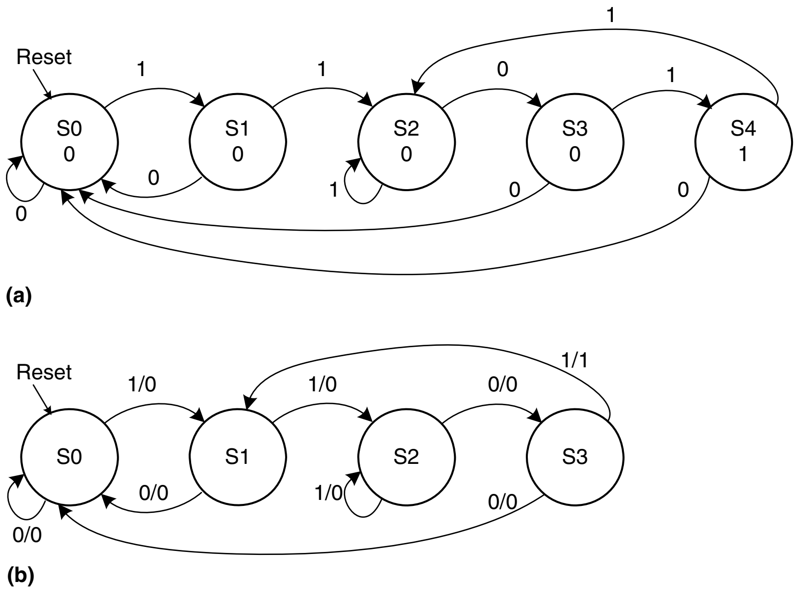


Figure 3.30 **FSM state transition diagrams: (a) Moore machine, (b) Mealy machine**

Table 3.11 Moore state transition table

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S1
S1	0	S0
S1	1	S2
S2	0	S3
S2	1	S2
S3	0	S0
S3	1	S4
S4	0	S0
S4	1	S2

Table 3.12 Moore output table

Current State S	Output Y
S0	0
S1	0
S2	0
S3	0
S4	1

Table 3.13 Moore state transition table with state encodings

Current State			Input	Next State		
S_2	S_1	S_0	A	S'_2	S'_1	S'_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	1	0	1	0

Table 3.14 Moore output table with state encodings

Current State			Output
S_2	S_1	S_0	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

Table 3.15 **Mealy state transition and output table**

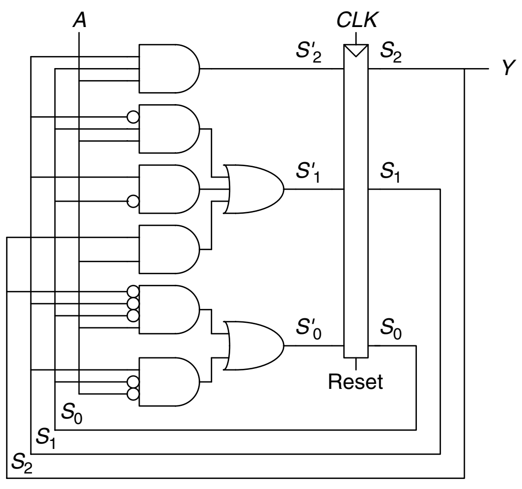
Current State S	Input A	Next State S'	Output Y
S0	0	S0	0
S0	1	S1	0
S1	0	S0	0
S1	1	S2	0
S2	0	S3	0
S2	1	S2	0
S3	0	S0	0
S3	1	S1	1

Table 3.16 **Mealy state transition and output table with state encodings**

Current State S_1	Current State S_0	Input A	Next State S'_1	Next State S'_0	Output Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

3.4.4 Factoring State Machines

Designing complex FSMs is often easier if they can be broken down into multiple interacting simpler state machines such that the output of some machines is the input of others. This application of hierarchy and modularity is called *factoring* of state machines.



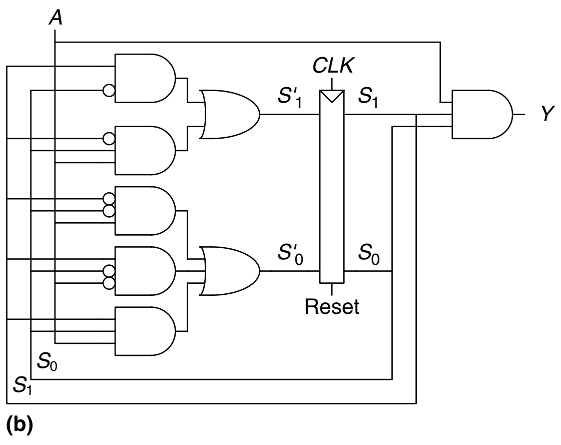


Figure 3.31 FSM schematics for (a) Moore and (b) Mealy machines

Example 3.8 UNFACTORED AND FACTORED STATE MACHINES

Modify the traffic light controller from Section 3.4.1 to have a parade mode, which keeps the Bravado Boulevard light green while spectators and the band march to football games in scattered groups. The controller receives two more inputs: P and R. Asserting P for at least one cycle enters parade mode. Asserting R for at least one cycle leaves parade mode. When in parade mode, the controller proceeds through its usual sequence until L_B turns green, then remains in that state with L_B green until parade mode ends.

First, sketch a state transition diagram for a single FSM, as shown in Figure 3.33(a). Then, sketch the state transition diagrams for two interacting FSMs, as

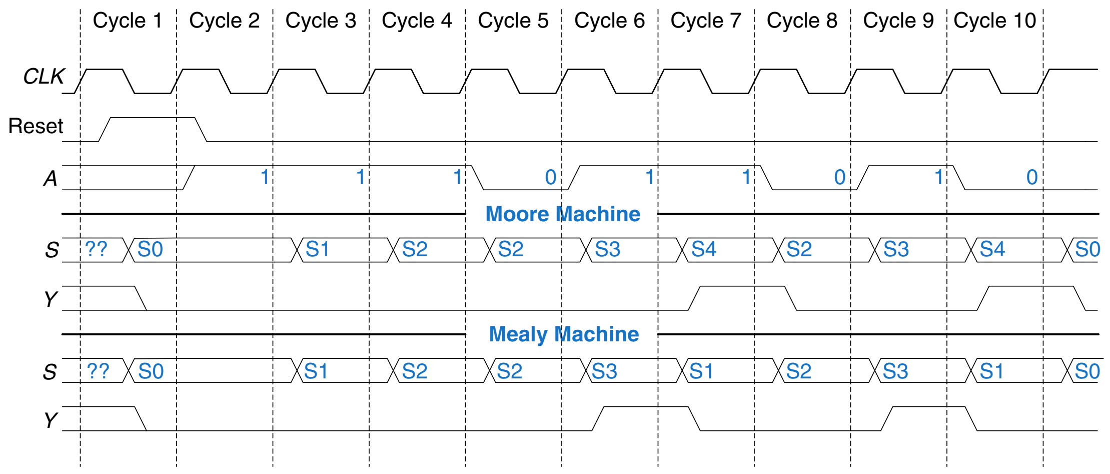


Figure 3.32 **Timing diagrams for Moore and Mealy machines**

shown in Figure 3.33(b). The Mode FSM asserts the output M when it is in parade mode. The Lights FSM controls the lights based on M and the traffic sensors, T_A and T_B.

Solution: Figure 3.34(a) shows the single FSM design. States S0 to S3 handle normal mode. States S4 to S7 handle parade mode. The two halves of the diagram are almost identical, but in parade mode, the FSM remains in S6 with a green light on Bravado Blvd. The P and R inputs control movement between these two halves. The FSM is messy and tedious to design. Figure 3.34(b) shows the factored FSM design. The mode FSM has two states to track whether the lights are in normal or parade mode. The Lights FSM is modified to remain in S2 while M is TRUE.

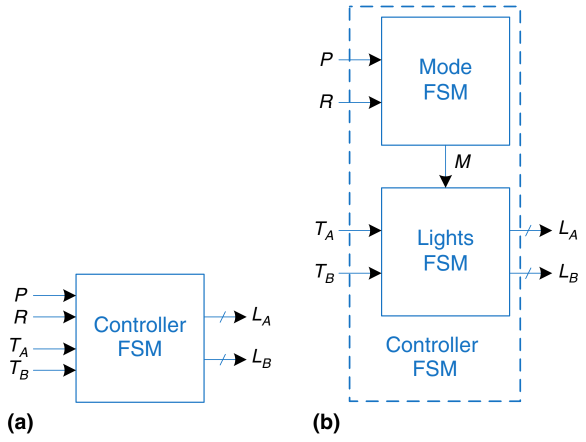


Figure 3.33 **(a) single and (b) factored designs for modified traffic light controller FSM**

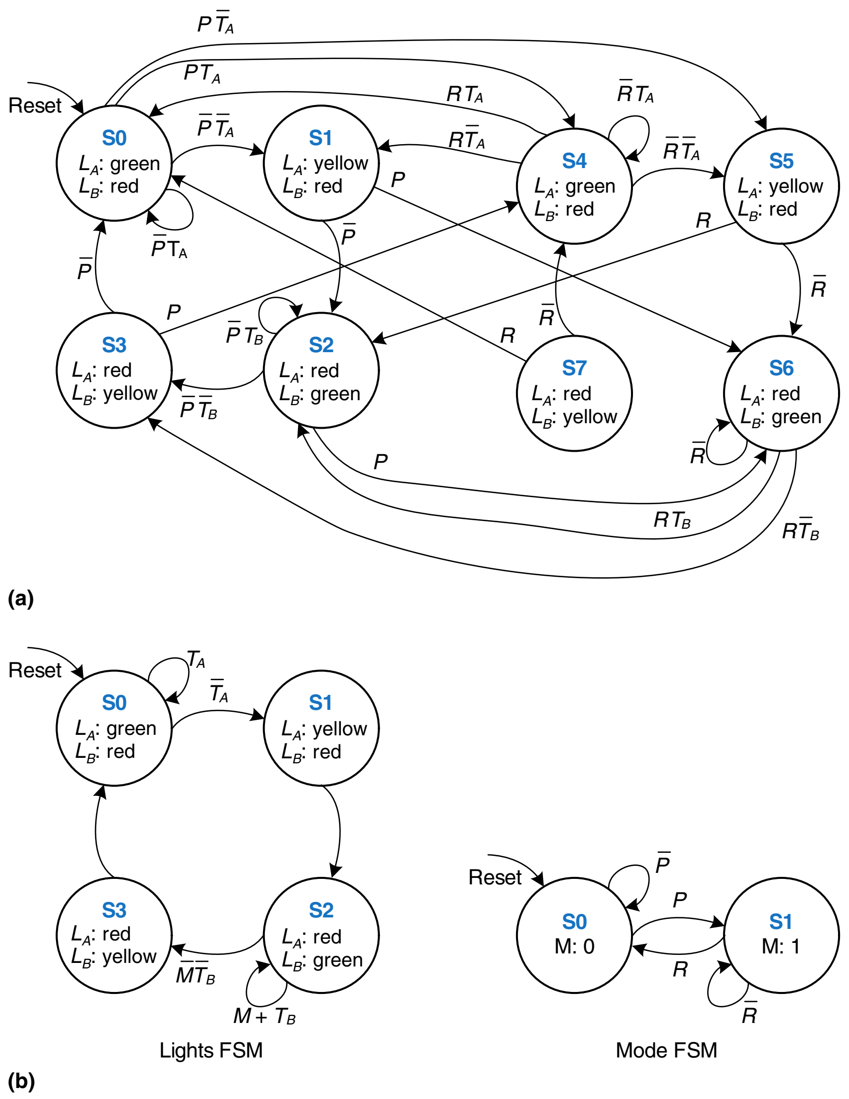


Figure 3.34 State transition diagrams: (a) unfactored, (b) factored

3.4.5 FSM Review

Finite state machines are a powerful way to systematically design sequential circuits from a written specification. Use the following procedure to design an FSM:

- Identify the inputs and outputs.
- Sketch a state transition diagram.

- For a Moore machine:
  - Write a state transition table.
  - Write an output table.
- For a Mealy machine:
  - Write a combined state transition and output table.
- Select state encodings—your selection affects the hardware design.
- Write Boolean equations for the next state and output logic.
- Sketch the circuit schematic.

We will repeatedly use FSMs to design complex digital systems throughout this book.

3.5 TIMING OF SEQUENTIAL LOGIC

Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called *sampling* D on the clock edge. If D is *stable* at either 0 or 1 when the clock rises, this behavior is clearly defined. But what happens if D is changing at the same time the clock rises?

This problem is similar to that faced by a camera when snapping a picture. Imagine photographing a frog jumping from a lily pad into the lake. If you take the picture before the jump, you will see a frog on a lily pad. If you take the picture after the jump, you will see ripples in the water. But if you take it just as the frog jumps, you may see a blurred image of the frog stretching from the lily pad into the water. A camera is characterized by its *aperture time,* during which the object must remain still for a sharp image to be captured. Similarly, a sequential element has an aperture time around the clock edge, during which the input must be stable for the flip-flop to produce a well-defined output.

The aperture of a sequential element is defined by a *setup* time and a *hold* time, before and after the clock edge, respectively. Just as the static discipline limited us to using logic levels outside the forbidden zone, the *dynamic discipline* limits us to using signals that change outside the aperture time. By taking advantage of the dynamic discipline, we can think of time in discrete units called clock cycles, just as we think of signal levels as discrete 1's and 0's. A signal may glitch and oscillate wildly for some bounded amount of time. Under the dynamic discipline, we are concerned only about its final value at the end of the clock cycle, after it has settled to a stable value. Hence, we can simply write $A[n]$, the value of signal A at the end of the n^{th} clock cycle, where n is an integer, rather than $A(t)$, the value of A at some instant t, where t is any real number.

The clock period has to be long enough for all signals to settle. This sets a limit on the speed of the system. In real systems, the clock does not