To see how this works, let's choose a data word, say, 0110, whose error correction code is 011. Here are the four 1-bit error possibilities for this data: 1110, 0010, 0100, and 0111. Now look at the data item with the same code (011), which is the entry with the value 0001. If the error correction decoder received one of the four possible data words with an error, it would have to choose between correcting to 0110 or 0001. While these four words with error have only one bit changed from the correct pattern of 0110, they each have two bits that are different from the alternate correction of 0001. Hence, the error correction mechanism can easily choose to correct to 0110, since a single error is a much higher probability. To see that two errors can be detected, simply notice that all the combinations with two bits changed have a different code. The one reuse of the same code is with three bits different, but if we correct a 2-bit error, we will correct to the wrong value, since the decoder will assume that only a single error has occurred. If we want to correct 1-bit errors and detect, but not erroneously correct, 2-bit errors, we need a distance-4 code.

Although we distinguished between the code and data in our explanation, in truth, an error correction code treats the combination of code and data as a single word in a larger code (7 bits in this example). Thus, it deals with errors in the code bits in the same fashion as errors in the data bits.

While the above example requires $n-1$ bits for n bits of data, the number of bits required grows slowly, so that for a distance-3 code, a 64-bit word needs 7 bits and a 128-bit word needs 8. This type of code is called a *Hamming code*, after R. Hamming, who described a method for creating such codes.

C.10 Finite-State Machines

As we saw earlier, digital logic systems can be classified as combinational or sequential. Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system. Thus, a sequential system cannot be described with a truth table. Instead, a sequential system is described as a **finite-state machine** (or often just *state machine*). A finite-state machine has a set of states and two functions, called the **next-state function** and the *output function*. The set of states corresponds to all the possible values of the internal storage. Thus, if there are n bits of storage, there are 2^n states. The next-state function is a combinational function that, given the inputs and the current state, determines the next state of the system. The output function produces a set of outputs from the current state and the inputs. Figure C.10.1 shows this diagrammatically.

finite-state machine
A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function
A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

The state machines we discuss here and in Chapter 4 are *synchronous*. This means that the state changes together with the clock cycle, and a new state is computed once every clock. Thus, the state elements are updated only on the clock edge. We use this methodology in this section and throughout Chapter 4, and we do not

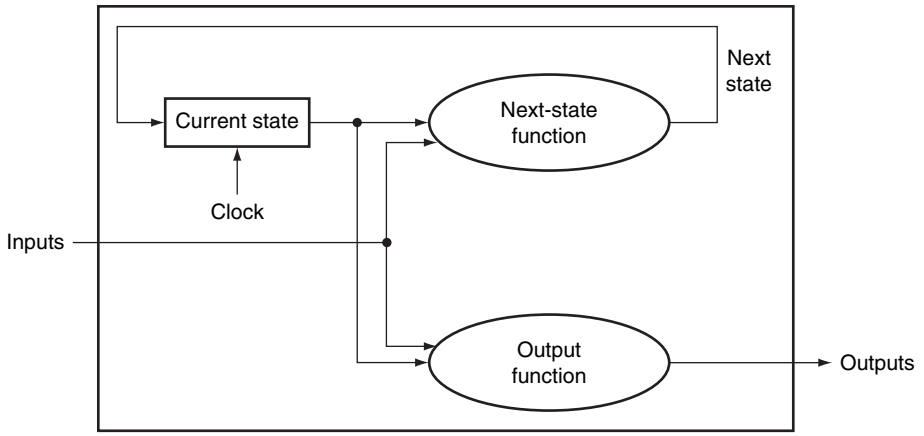

FIGURE C.10.1 A state machine consists of internal storage that contains the state and two combinational functions: the next-state function and the output function. Often, the output function is restricted to take only the current state as its input; this does not change the capability of a sequential machine, but does affect its internals.

usually show the clock explicitly. We use state machines throughout Chapter 4 to control the execution of the processor and the actions of the datapath.

To illustrate how a finite-state machine operates and is designed, let's look at a simple and classic example: controlling a traffic light. (Chapters 4 and 5 contain more detailed examples of using finite-state machines to control processor execution.) When a finite-state machine is used as a controller, the output function is often restricted to depend on just the current state. Such a finite-state machine is called a *Moore machine*. This is the type of finite-state machine we use throughout this book. If the output function can depend on both the current state and the current input, the machine is called a *Mealy machine*. These two machines are equivalent in their capabilities, and one can be turned into the other mechanically. The basic advantage of a Moore machine is that it can be faster, while a Mealy machine may be smaller, since it may need fewer states than a Moore machine. In Chapter 5, we discuss the differences in more detail and show a Verilog version of finite-state control using a Mealy machine.

Our example concerns the control of a traffic light at an intersection of a north-south route and an east-west route. For simplicity, we will consider only the green and red lights; adding the yellow light is left for an exercise. We want the lights to cycle no faster than 30 seconds in each direction, so we will use a 0.033 Hz clock so that the machine cycles between states at no faster than once every 30 seconds. There are two output signals:

- *NSlite:* When this signal is asserted, the light on the north-south road is green; when this signal is deasserted, the light on the north-south road is red.
- *EWlite:* When this signal is asserted, the light on the east-west road is green; when this signal is deasserted, the light on the east-west road is red.

In addition, there are two inputs:

- *NScar:* Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south).
- *EWcar:* Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west).

The traffic light should change from one direction to the other only if a car is waiting to go in the other direction; otherwise, the light should continue to show green in the same direction as the last car that crossed the intersection.

To implement this simple traffic light we need two states:

- *NSgreen:* The traffic light is green in the north-south direction.
- *EWgreen:* The traffic light is green in the east-west direction.

We also need to create the next-state function, which can be specified with a table:

	Inputs		
	NScar	EWcar	Next state
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Notice that we didn't specify in the algorithm what happens when a car approaches from both directions. In this case, the next-state function given above changes the state to ensure that a steady stream of cars from one direction cannot lock out a car in the other direction.

The finite-state machine is completed by specifying the output function.

Before we examine how to implement this finite-state machine, let's look at a graphical representation, which is often used for finite-state machines. In this representation, nodes are used to indicate states. Inside the node we place a list of the outputs that are active for that state. Directed arcs are used to show the next-state

	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

function, with labels on the arcs specifying the input condition as logic functions. Figure C.10.2 shows the graphical representation for this finite-state machine.

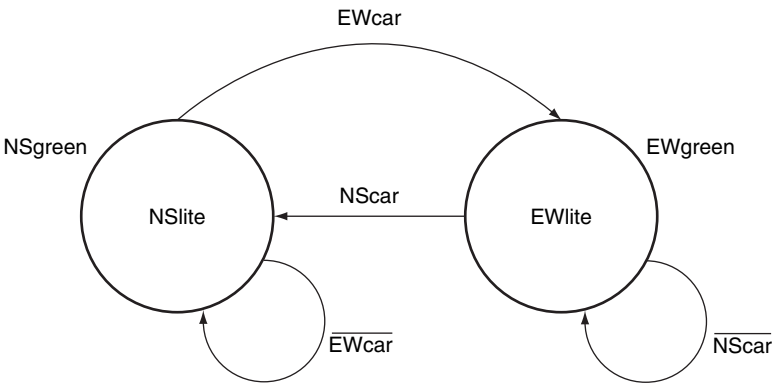

FIGURE C.10.2 The graphical representation of the two-state traffic light controller. We simplified the logic functions on the state transitions. For example, the transition from NSgreen to EWgreen in the next-state table is $(\overline{\text{NScar}} \cdot \text{EWcar}) + (\text{NScar} \cdot \text{EWcar})$, which is equivalent to EWcar.

A finite-state machine can be implemented with a register to hold the current state and a block of combinational logic that computes the next-state function and the output function. Figure C.10.3 shows how a finite-state machine with 4 bits of state, and thus up to 16 states, might look. To implement the finite-state machine in this way, we must first assign state numbers to the states. This process is called *state assignment*. For example, we could assign NSgreen to state 0 and EWgreen to state 1. The state register would contain a single bit. The next-state function would be given as

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

where CurrentState is the contents of the state register (0 or 1) and NextState is the output of the next-state function that will be written into the state register at the end of the clock cycle. The output function is also simple:

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

The combinational logic block is often implemented using structured logic, such as a PLA. A PLA can be constructed automatically from the next-state and output function tables. In fact, there are computer-aided design (CAD) programs

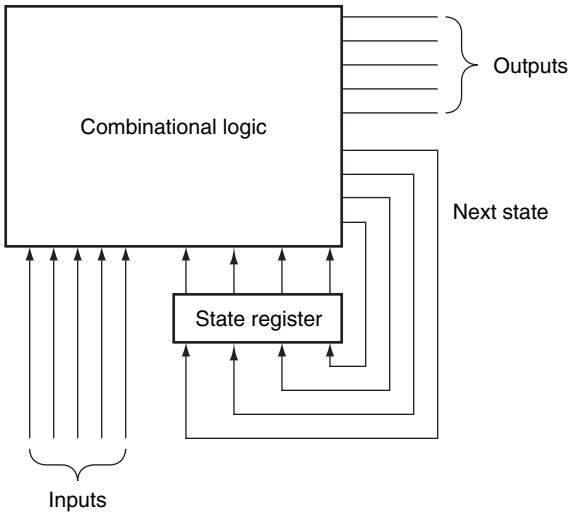

FIGURE C.10.3 A finite-state machine is implemented with a state register that holds the current state and a combinational logic block to compute the next state and output functions. The latter two functions are often split apart and implemented with two separate blocks of logic, which may require fewer gates.

that take either a graphical or textual representation of a finite-state machine and produce an optimized implementation automatically. In Chapters 4 and 5, finite-state machines were used to control processor execution. Appendix C discusses the detailed implementation of these controllers with both PLAs and ROMs.

To show how we might write the control in Verilog, Figure C.10.4 shows a Verilog version designed for synthesis. Note that for this simple control function, a Mealy machine is not useful, but this style of specification is used in Chapter 5 to implement a control function that is a Mealy machine and has fewer states than the Moore machine controller.

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);

   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //set initial state

//following two assignments set the output, which is based
only on the state variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state = 1

always @(posedge clock) // all state updates on a positive
clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar

      1: state = NSCar; //change state only if NSCar

   endcase
endmodule
```

FIGURE C.10.4 A Verilog version of the traffic light controller.

Check Yourself

What is the smallest number of states in a Moore machine for which a Mealy machine could have fewer states?

a. Two, since there could be a one-state Mealy machine that might do the same thing.

b. Three, since there could be a simple Moore machine that went to one of two different states and always returned to the original state after that. For such a simple machine, a two-state Mealy machine is possible.

c. You need at least four states to exploit the advantages of a Mealy machine over a Moore machine.

C.11 Timing Methodologies

Throughout this appendix and in the rest of the text, we use an edge-triggered timing methodology. This timing methodology has an advantage in that it is simpler to explain and understand than a level-triggered methodology. In this section, we explain this timing methodology in a little more detail and also introduce level-sensitive clocking. We conclude this section by briefly discussing the issue