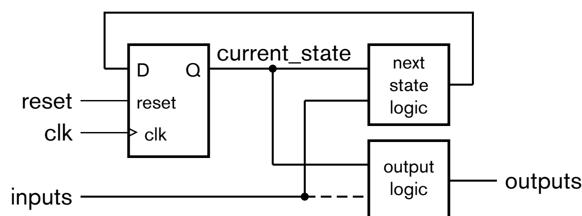

FIGURE 4.31 Circuit structure for a finite-state machine.

In general terms, a finite-state machine is defined by a set of *inputs*, a set of *outputs*, a set of *states*, a *transition function* that governs transitions between states, and an *output function*. The states are just abstract values that mark steps in a sequence of operations. The machine is called "finite-state" because the set of states is finite in size. The finite-state machine has a *current state* in a given clock cycle. The transition function determines the *next state* for the next clock cycle based on the current state and, possibly, the values of inputs in the given clock cycle. The output function determines the values of the outputs in a given clock cycle based on the current state and, possibly, the values of inputs in the given clock cycle.

Figure 4.31 shows a schematic representation of a finite-state machine. The register stores the current state in binary coded form. One of the states in the state set is designated the *initial state*. When the system is reset, the register is reset to the binary code for the initial state; thus, the finite-state machine assumes the initial state as its current state. During each clock cycle, the value of the next state is computed by the next state logic, which is a combinational circuit that implements the transition function. Also, the outputs are driven with the value computed by the output logic, which is a combinational circuit that implements the output function. The outputs are the control signals that govern operation of a datapath. On the rising clock edge marking the beginning of the next clock cycle, the current state is updated with the computed next-state value. The next state may be the same as the previous state, or it may be a different state.

Finite-state machines are often divided into two classes. In a *Mealy* finite-state machine, the output function depends on both the current state and the values of the inputs. In such a machine, the connection drawn with a dashed line in Figure 4.31 is present. If the input values change during a clock cycle, the output values may change as a consequence. In a *Moore* finite-state machine, on the other hand, the output function depends only on the current state, and not on the input values. The dashed connection in Figure 4.31 is absent in a Moore machine. If the input values change during a clock cycle, the outputs remain unchanged.

In theory, for any Mealy machine, there is an equivalent Moore machine, and *vice versa*. However, in practice, one or the other kind of machine will be most appropriate. A Mealy machine may be able to implement a given control sequence with fewer states, but it may be harder to meet timing constraints, due to delays in arrival of inputs used to compute the next state. As we present examples of finite-state machines, we will identify whether they are Mealy or Moore machines.

In many finite-state machines, there is an idle state that indicates that the system is waiting to start a sequence of operations. When an input indicates that the sequence should start, the finite-state machine follows a sequence of states on successive clock cycles, with the output values controlling the operations in a datapath. Eventually, when the sequence of operations is complete, the finite-state machine returns to the idle state.

EXAMPLE 4.16 Design a finite-state machine to implement the control sequence for the complex multiplier described in Example 4.15. The control sequence is initiated by input_rdy being 1 during the clock cycle in which new data arrives at the datapath inputs.

SOLUTION Our finite-state machine needs five states, one for each of the steps of the control sequence. Let's call them step1 through step5. We also need to deal with the case of waiting for input data to arrive. We could consider a separate idle state for that case. When, in the idle state, input_rdy is 1, we would then transition to state1 to start the multiplication; otherwise, we would stay in the idle state. The problem with this is that it wastes a clock cycle, since we would not perform the first multiplication until after the cycle in which data arrived.

The alternative is to use step1 as the idle state. If it turns out that new data has not arrived in a given clock cycle while in this state, we simply repeat step1 as the next state. On the other hand, if new data has arrived, indicated by input_rdy being 1 in the clock cycle, the real parts are multiplied during that clock cycle and can be stored on the next clock edge. We would then transition to step2, and on subsequent clock cycles to step3, step4 and step5. At the end of the step5 clock cycle, the complete complex product is stored in the output registers of the datapath, so we can transition back to step1 in the next clock cycle.

In summary, our finite-state machine has the signal input_rdy as its single input, and the control signals listed in Example 4.15 as outputs. The state set is {step1, step2, step3, step4, step5}, with step1 being the initial state. The transition function is defined in Table 4.2. The output function is defined in Table 4.1. Since the output function depends only on the current state and not on the input value, this finite-state machine is a Moore machine.

---

current_ state	input_ rdy	next_ state
step1	0	step1
step1	1	step2
step2	–	step3
step3	–	step4
step4	–	step5
step5	–	step1

TABLE 4.2 The transition function for the complex multiplier finite-state machine.

An important issue to consider when designing a finite-state machine is how to encode the state values. We glossed over that in Example 4.16 by treating the states as abstract values. As we discussed in Chapter 2, if we have N states, we need at least $\lceil \log_2 N \rceil$ bits in our code. However, we may choose to have more if that simplifies circuitry that uses encoded states. In particular, while a longer than minimal code length requires more flip-flops in the state register and more wires for the state signals, it may make the next-state and output logic circuits simpler and smaller. In general choosing an optimal state encoding is a complex mathematical problem. However, synthesis CAD tools incorporate methods for choosing a state encoding, so we may be able to let a tool make the choice for us. One aspect of state encoding is the choice of a code word to represent the initial state. In many cases, a good choice is a code word with all 0 bits, since that allows us to use a simple register with reset for the state register. If some other code word is chosen for the initial state, that code word must be loaded into the register on system reset.

Modeling Finite-State Machines in Verilog

Since a finite-state machine is composed of a register, next-state logic and output logic, a straightforward way to model a finite-state machine is to use the Verilog features that we already know for modeling registers and combinational logic. The only aspect we have not addressed is how to represent the state set, particularly when we want to take an abstract view and leave state encoding to the synthesis tool. In Verilog, we can use *parameter definitions* to specify a set of symbolic names associated with the binary code words for the states. For example, we can define parameters for the states in Example 4.16 as follows:

```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
```

This defines five parameters, named `step1` through `step5`, corresponding to the binary code words 000 through 100, respectively. In the rest of the state machine model, we just use the symbolic names, not the code word values. A synthesis tool may be able to recode the state parameters, that is, to choose an alternate encoding for the state set, to optimize the generated hardware for the state machine.

We can declare a variable to represent the current state of a state machine as follows:

```
reg [2:0] current_state;
```

This specifies that current_state is a vector that can take on parameter values representing states. So, for example, we could make the following assignment in a procedural block:

```
current_state <= step4;
```

to assign the value step4 to the variable.

EXAMPLE 4.17 Develop a Verilog model of the finite-state machine in Example 4.16.

SOLUTION We will augment the architecture declaration of Example 4.14 with the Verilog representation of the control section. The additional declarations of parameters for the set of states and variables for the current and next state are

```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
reg [2:0] current_state, next_state ;
```

The additional statements added to the module are

```
  always @(posedge clk or posedge reset)  // State register
    if (reset) current_state <= step1;
    else       current_state <= next_state;

   always @*  // Next-state logic
    case (current_state)
      step1: if (!input_rdy) next_state = step1;
             else            next_state = step2;
      step2:                 next_state = step3;
      step3:                 next_state = step4;
      step4:                 next_state = step5;
      step5:                 next_state = step1;
    endcase

   always @* begin  // Output_logic
    a_sel = 1'b0; b_sel = 1'b0;  pp1_ce = 1'b0; pp2_ce = 1'b0;
    sub = 1'b0;   p_r_ce = 1'b0; p_i_ce = 1'b0;
    case (current_state)
      step1: begin
               pp1_ce = 1'b1;
             end
```

(continued)

```
      step2: begin
               a_sel = 1'b1; b_sel = 1'b1; pp2_ce = 1'b1;
             end
      step3: begin
               b_sel = 1'b1; pp1_ce = 1'b1;
               sub = 1'b1;   p_r_ce = 1'b1;
             end
      step4: begin
               a_sel = 1'b1; pp2_ce = 1'b1;
             end
      step5: begin
               p_i_ce = 1'b1;
             end
    endcase
  end
```

The first always block models the state storage for the finite-state machine. It is based on the template for a register with asynchronous reset. When the reset input is active, the block resets the current state to the initial state, step1. Otherwise, on a rising clock edge, the block updates the current state with the computed next state.

The next state is computed by the second always block, which models the transition function of Table 4.2. The statement inside the block is a *case statement*. It uses the value of the current_state variable to choose among alternatives for updating next_state. The alternative for step1 uses a nested if statement to determine whether to proceed to step2 or stay in step1, depending on the value of input_rdy. All other alternatives simply advance the state unconditionally.

The output values are computed by the third always block, which models the output function of Table 4.1. This block also includes a case statement that chooses alternatives for assigning values to the outputs depending on the value of current_state. Rather than including an assignment for every output in each alternative of the case statement, we precede the case statement with a default assignment of 0 for each output, and only include overriding assignments of 1 in those alternatives where they are required. This style for modeling the output function usually makes the always block more succinct, and helps to avoid inadvertent introduction of latches due to omission of an output assignment in an alternative.

State Transition Diagrams

A *state transition diagram* is an abstract diagrammatic representation of a finite-state machine. It uses a circle, or "bubble," to represent each state. Directed arcs between state bubbles represent transitions from one state to another. An arc may be labeled with a combination of input values

that allow the transition to occur. To illustrate, Figure 4.32 shows a state transition diagram for a finite-state machine with states s1, s2 and s3. Each arc is labeled with the values of two inputs, a_1 and a_2, that are required for the transition. Thus, when the finite-state machine is in state s1 and the inputs are both 1, the state of the machine in the next clock cycles is s3. If the machine is in state s1 and both inputs are 0, the machine stays in state s1. From state s1, if the inputs are 0 and 1, or 1 and 0, the machine transitions to state s2. Note that we have omitted a label on the arc from s2 to s3. This is a common convention to indicate an unconditional transition; that is, when the machine is in state s2, the next state is s3 regardless of the input values. Another important point is that all possible combinations of input values are accounted for in each state, and that no combination is repeated on more than one arc from a given state.

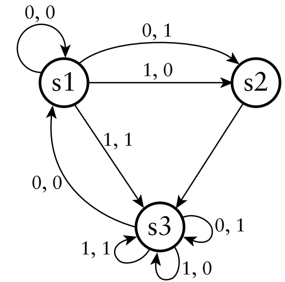

FIGURE 4.32 A state transition diagram.

A bubble diagram may also be labeled with the values of outputs. Since Moore-machine outputs depend only on the current state, we attach the labels for such outputs to the state bubbles. This is shown on the augmented bubble diagram in Figure 4.33. For each state, we list the values of two Moore-style outputs, x_1 and x_2, in that order.

Mealy-machine outputs, on the other hand, depend on both the current state and the current input values. Usually, the input conditions are the same as those that determine the next state, so we usually attach Mealy-output labels to the arcs. This does not imply that the outputs change at the time of the transition, only that the output values are driven when the current state is the source state of the arc and the input values are those of the arc label. If the inputs change while in the source state, the outputs change to those listed on some other arc labeled with

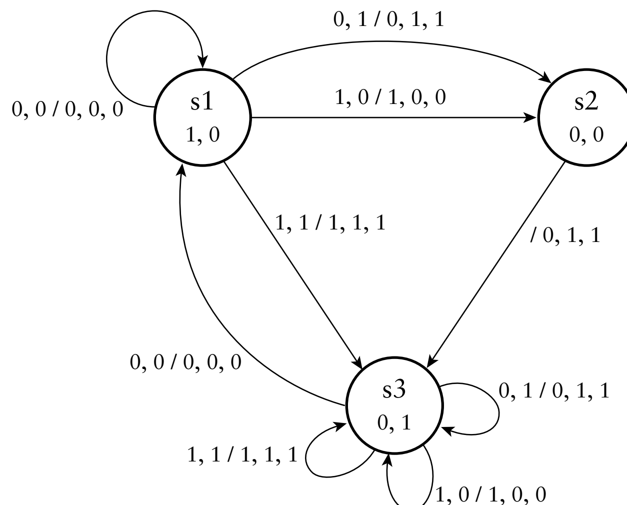

FIGURE 4.33 A state transition diagram augmented with Moore- and Mealy-style output values.

the new input values. Mealy-style outputs are also shown on the arcs in Figure 4.33. In each case, the output values are listed after the "/" in the order y_1, y_2 and y_3.

EXAMPLE 4.18 Draw a state transition diagram for the finite-state machine of Example 4.16. Include the output values in the order of their occurrence in Table 4.1.

SOLUTION The diagram is shown in Figure 4.34. There is a transition from `step1` to `step2` that occurs when `input_rdy` is 1, and a transition from `step1` back to itself when `input_rdy` is 0. All other transitions are unconditional. Since it is a Moore machine, the output values are all drawn in the state bubbles.

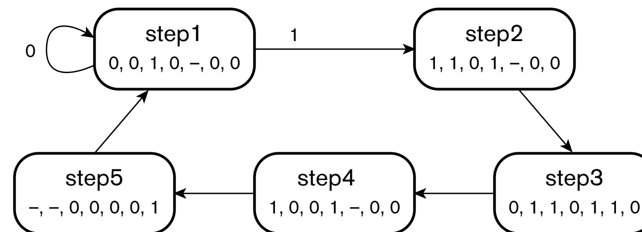

FIGURE 4.34 State transition diagram for the complex multiplier.

In many applications, a state transition diagram is a useful notation, since it graphically conveys the control organization of a sequential design. Many CAD tools provide graphical editors for entering state transition diagrams, and can automatically generate Verilog code for simulation and synthesis. The disadvantage of the notation is that the annotations of input conditions and output values can clutter the diagram, obscuring the control organization. Also, for large and complex state machines, the diagram can become unwieldy. In those cases, a Verilog model in textual form may be more intelligible. Ultimately, since state transition diagrams and Verilog models of state machines encapsulate the same information, it is a question of personal preference or project guidelines that determine the method to use.

KNOWLEDGE TEST QUIZ

1. What is the purpose of the datapath in a digital system?
2. What is the purpose of the control section in a digital system?
3. What are control signals and status signals?
4. What is the distinction between a Moore and a Mealy finite-state machine?