Chapter 8: Modeling Finite State Machines

In this chapter, we will look at modeling finite state machines (FSMs). An FSM is one of the most powerful circuits in a digital system because it can make decisions about the next output based on both the current and past inputs. Finite state machines are modeled using the constructs already covered in this book. In this chapter, we will look at the widely accepted three-process model for designing a FSM.

Learning Outcomes—After completing this chapter, you will be able to:

8.1 Describe the three-process modeling approach for FSM design.
8.2 Design a Verilog model for a FSM from a state diagram.

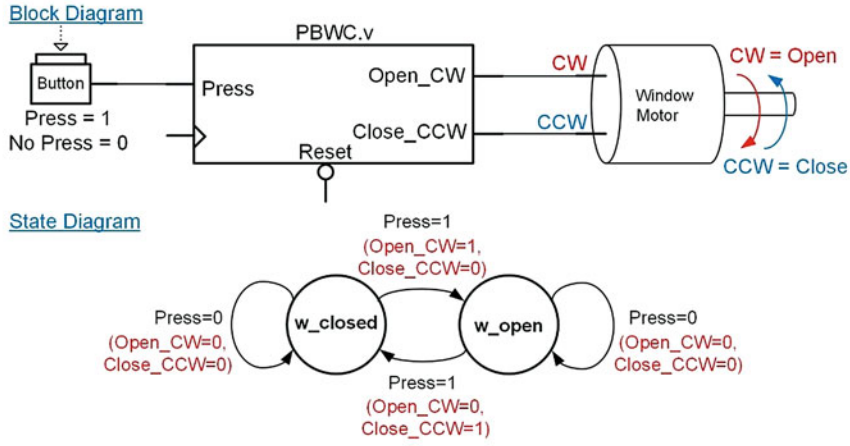
8.1 The FSM Design Process and a Push-Button Window Controller Example

The most common modeling practice for FSMs is to declare two signals of type reg that are called *current_state* and *next_state*. Then a parameter is declared for each descriptive state name in the state diagram. A parameter also requires a value, so the state encoding can be accomplished during the parameter declaration. Once the signals and parameters are created, all of the procedural assignments in the state machine model can use the descriptive state names in their signal assignments. Within the Verilog state machine model, three separate procedural blocks are used to describe each of the functional blocks, *state memory, next state logic,* and *output logic*. In order to examine how to model a finite state machine using this approach, let's use the push-button window controller example from Chap. 7. Example 8.1 gives the overview of the design objectives for this example and the state diagram describing the behavior to be modeled in Verilog.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-030-10552-5_8

Example: Push-Button Window Controller in Verilog – Design Description

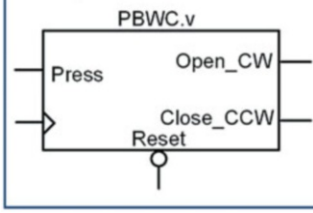
The window controller will send the appropriate control signals to a motor to open or close it whenever a button is pressed. The system must keep track whether the window is open or closed in order to send the correct signal, thus a state machine is needed. The block diagram and state diagram for this system is shown below.



Example 8.1
Push-button window controller in Verilog—design description

Let's begin by defining the ports of the module. The system has an input called *Press* and two outputs called *Open_CW* and *Close_CCW*. The system also has clock and reset inputs. We will design the system to update on the rising edge of the clock and have an asynchronous, active LOW, reset. Example 8.2 shows the port definitions for this example. Note that outputs are declared as type reg while inputs are declared as type wire.

Example: Push-Button Window Controller in Verilog – Port Definition




```
module PBWC (output reg  Open_CW, Close_CCW,
             input  wire Clock, Reset, Press);
                    :
                    :
```

Outputs are defined as type reg while inputs are defined of type wire.

Example 8.2
Push-button window controller in Verilog—port definition

8.1.1 Modeling the States

Now we begin designing the finite state machine in Verilog using behavioral modeling constructs. The first step is to create two signals that will be used for the state variables. In this text we will always name these signals *current_state* and *next_state*. The signal current_state will represent the outputs of the D-flip-flops forming the state memory and will hold the current state code. The signal next_state will represent the D inputs to the D-flip-flops forming the state memory and will receive the value from the next state logic circuitry. Since the FSM will be modeled using procedural assignment, both of these

signals will be declared of type reg. The width of the reg vector depends on the number of states in the machine and the encoding technique chosen. The next step is to declare parameters for each of the descriptive state names in the state diagram. The state encoding must be decided at this point. The following syntax shows how to declare the current_state and next_state signals and the parameters. Note that since this machine only has two states, the width of these signals is only 1-bit.

```
reg         current_state, next_state;
parameter   w_closed = 1'b0,
            w_open   = 1'b1;
```

8.1.2 The State Memory Block

Now that we have variables and parameters for the states of the FSM, we can create the model for the state memory. State memory is modeled using its own procedural block. This block models the behavior of the D-Flip-Flops in the FSM that are holding the current state on their Q outputs. Each time there is a rising edge of the clock, the current state is updated with the next state value present on the D inputs of the D-Flip-Flops. This block must also model the reset condition. For this example, we will have the state machine go to the *w_closed* state when Reset is asserted. At all other times, the block will simply update current_state with next_state on every rising edge of the clock. The block model is very similar to the model of a D-Flip-Flop. This is as expected since this block will synthesize into one or more D-Flip-Flops to hold the current state. The sensitivity list contains only Clock and Reset and assignments are only made to the signal current_state. The following syntax shows how to model the state memory of this FSM example.

```
always @ (posedge Clock or negedge Reset)
  begin: STATE_MEMORY
    if (!Reset)
        current_state <= w_closed;
    else
        current_state <= next_state;
  end
```

8.1.3 The Next State Logic Block

Now we model the next state logic of the FSM using a second procedural block. Recall that the next state logic is combinational logic; thus, we need to include all of the input signals that the circuit considers in the next state calculation in the sensitivity list. The current_state signal will always be included in the sensitivity list of the next state logic block in addition to any inputs to the system. For this example, the system has one other input called *Press*. This block makes assignments to the next_state signal. It is common to use a case statement to separate out the assignments that occur at each state. At each state within the case statement, an if-else statement is used to model the assignments for different input conditions on Press. The following syntax shows how to model the next state logic of this FSM example. Notice that we include a *default* clause in the case statement to ensure that the state machine has a path back to the reset state in the case of an unexpected fault.

```
always @ (current_state or Press)
  begin: NEXT_STATE_LOGIC
    case (current_state)
      w_closed : if (Press ==1'b1) next_state = w_open;  else next_state = w_closed;
      w_open   : if (Press == 1'b1) next_state = w_closed; else next_state = w_open;
      default  : next_state = w_closed;
    endcase
  end
```

8.1.4 The Output Logic Block

Now we model the output logic of the FSM using a third procedural block. Recall that output logic is combinational logic; thus, we need to include all of the input signals that this circuit considers in the output assignments. The current_state will always be included in the sensitivity list. If the FSM is a Mealy machine, then the system inputs will also be included in the sensitivity list. If the machine is a Moore machine, then only the current_state will be present in the sensitivity list. For this example, the FSM is a Mealy machine, so the input Press needs to be included in the sensitivity list. Note that this block only makes assignments to the outputs of the machine (Open_CW and Close_CCW). The following syntax shows how to model the output logic of this FSM example. Again, we include a *default* clause to ensure that the state machine has explicit output behavior in the case of a fault.

```
always @ (current_state or Press)
  begin: OUTPUT_LOGIC
    case (current_state)
      w_closed : if (Press == 1'b1)
                   begin
                     Open_CW  = 1'b1;
                     Close_CCW = 1'b0;
                   end
                 else
                   begin
                     Open_CW  = 1'b0;
                     Close_CCW = 1'b0;
                   end
      w_open   : if (Press == 1'b1)
                   begin
                     Open_CW  = 1'b0;
                     Close_CCW = 1'b1;
                   end
                 else
                   begin
                     Open_CW  = 1'b0;
                     Close_CCW = 1'b0;
                   end
      default  : begin
                   Open_CW  = 1'b0;
                   Close_CCW = 1'b0;
                 end
    endcase
  end
```

Putting this all together yields a behavioral model for the FSM that can be simulated and synthesized. Example 8.3 shows the entire model for this example.

Example: Push-Button Window Controller in Verilog – Full Model

```
module PBWC (output reg  Open_CW, Close_CCW,
             input  wire Clock, Reset, Press);

  reg       current_state, next_state;
  parameter w_closed = 1'b0,
            w_open   = 1'b1;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= w_closed;
      else
        current_state <= next_state;
    end

  always @ (current_state or Press)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        w_closed : if (Press == 1'b1)
                     next_state = w_open;
                   else
                     next_state = w_closed;
        w_open   : if (Press == 1'b1)
                     next_state = w_closed;
                   else
                     next_state = w_open;
        default  : next_state = w_closed;
      endcase
    end

  always @ (current_state or Press)
     begin: OUTPUT_LOGIC
       case (current_state)
         w_closed : if (Press == 1'b1)
                      begin
                        Open_CW   = 1'b1;
                        Close_CCW = 1'b0;
                      end
                    else
                      begin
                        Open_CW   = 1'b0;
                        Close_CCW = 1'b0;
                      end
         w_open   : if (Press == 1'b1)
                      begin
                        Open_CW   = 1'b0;
                        Close_CCW = 1'b1;
                      end
                    else
                      begin
                        Open_CW   = 1'b0;
                        Close_CCW = 1'b0;
                      end
          default : begin
                        Open_CW   = 1'b0;
                        Close_CCW = 1'b0;
                    end
      endcase
    end

endmodule
```

Declaration of state variables and state encoding.

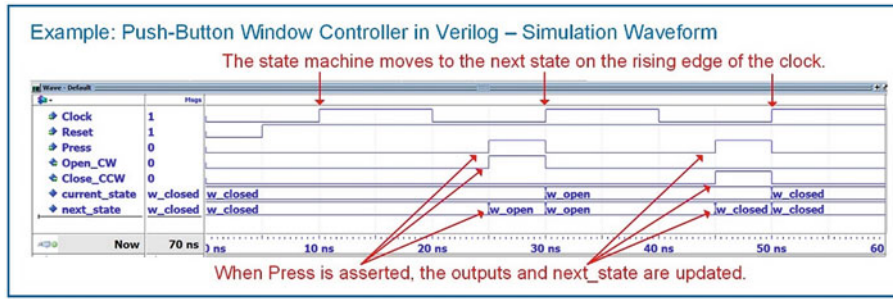
State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

Next state logic block. This is combinational logic so blocking assignments are used. This block only makes assignments to the signal "next_state".

Output logic block. This is combinational logic so blocking assignments are used. Since this is a Mealy machine, the current state and input are listed in the sensitivity list. This block only makes assignments to the outputs "Open_CW" and "Close_CCW".

Example 8.3
Push-button window controller in Verilog—full model

Example 8.4 shows the simulation waveform for this state machine. This functional simulation was performed using ModelSim-Altera Starter Edition 10.1d. A macro file was used to display the current and next state variables using their parameter names instead of their state codes. This allows the functionality of the FSM to be more easily observed. This approach will be used for the rest of the FSM examples in this book.

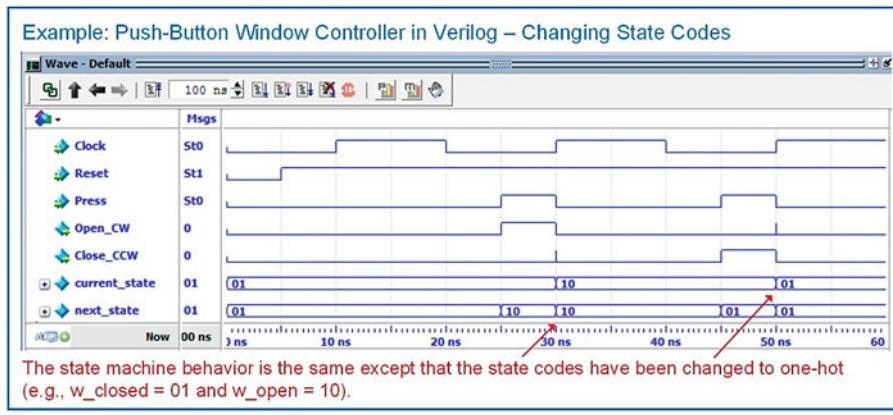


Example 8.4
Push-button window controller in Verilog—simulation waveform

8.1.5 Changing the State Encoding Approach

In the prior example we had two states that were encoded as: w_closed = 1'b0; w_open_1'b1. This encoding technique is considered *binary* and takes 1-bit; however, a *one-hot* could be adopted that would require 2-bits. The way that state variables and state codes are assigned in Verilog makes is straightforward to change the state codes. The only consideration that must be made is expanding the size of the current_state and next_state variables to accommodate the new state codes. The following example shows how the state encoding would look if a *one-hot* approach was used (w_closed = 2'b01; w_open_2'b10). Note that the state variables now must be two bits wide. This means the state variables need to be declared as type reg[1:0]. Example 8.5 shows the resulting simulation waveforms. The simulation waveform shows the value of the state codes instead of the state names.

```
reg [1:0]  current_state, next_state;
parameter  w_closed = 2'b01,
           w_open   = 2'b10;
```



Example 8.5
Push-button window controller in Verilog—changing state codes

CONCEPT CHECK

CC8.1 Why is it always a good design approach to model a generic finite state machine using three processes?

(A) For readability.

(B) So that it is easy to identify whether the machine is a Mealy or Moore.

(C) So that the state memory process can be reused in other FSMs.

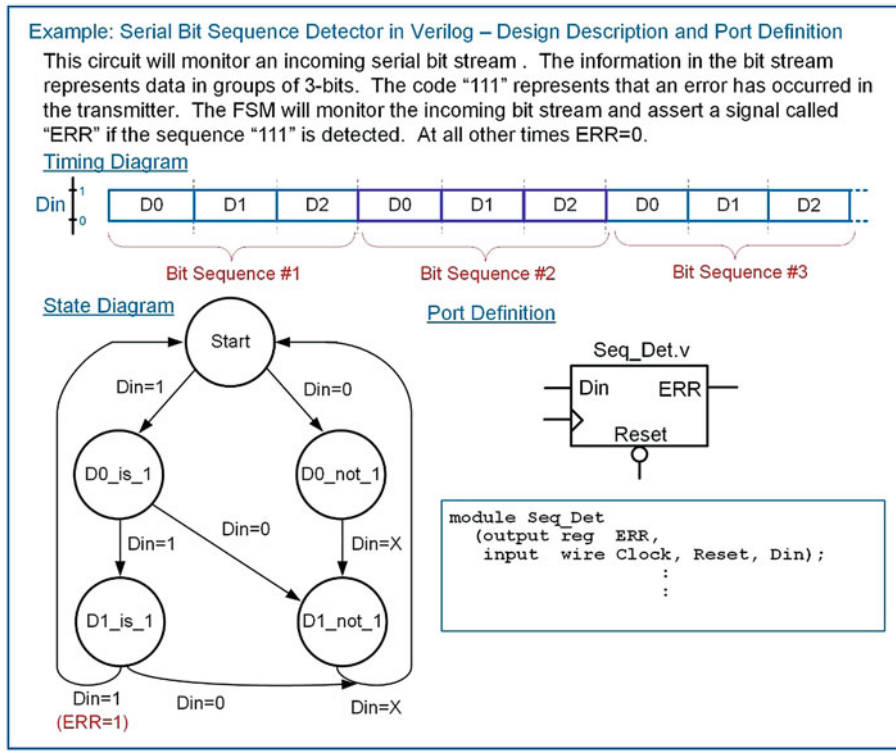
(D) Because each of the three subsystems of a FSM has unique inputs and outputs that should be handled using dedicated processes.

8.2 FSM Design Examples

This section presents a set of example finite state machine designs using the behavioral modeling constructs of Verilog.

8.2.1 Serial Bit Sequence Detector in Verilog

Let's look at the design of the serial bit sequence detector finite state machine using the behavioral modeling constructs of Verilog. Example 8.6 shows the design description and port definition for this state machine.



Example 8.6
Serial bit sequence detector in Verilog—design description and port definition

Example 8.7 shows the full model for the serial bit sequence detector. Notice that the states are encoded in binary, which requires three bits for the variables current_state and next_state.

Example: Serial Bit Sequence Detector in Verilog – Full Model

```
module Seq_Det (output reg  ERR,
                input  wire Clock, Reset, Din);

  reg [2:0] current_state, next_state;
  parameter Start    = 3'b000,
            D0_is_1  = 3'b001,
            D1_is_1  = 3'b010,
            D0_not_1 = 3'b011,
            D1_not_1 = 3'b100;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= Start;
      else
        current_state <= next_state;
    end

  always @ (current_state or Din)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        Start    : if (Din == 1'b1)
                     next_state = D0_is_1;
                   else
                     next_state = D0_not_1;
        D0_is_1  : if (Din == 1'b1)
                     next_state = D1_is_1;
                   else
                     next_state = D1_not_1;
        D1_is_1  : next_state = Start;
        D0_not_1 : next_state = D1_not_1;
        D1_not_1 : next_state = Start;
        default  : next_state = Start;
      endcase
    end

  always @ (current_state or Din)
    begin: OUTPUT_LOGIC
      case (current_state)
        D1_is_1  : if (Din == 1'b1)
                     ERR = 1'b1;
                   else
                     ERR = 1'b0;

        default  : ERR = 1'b0;
      endcase
    end

endmodule
```

Declaration of state variables and state encoding.

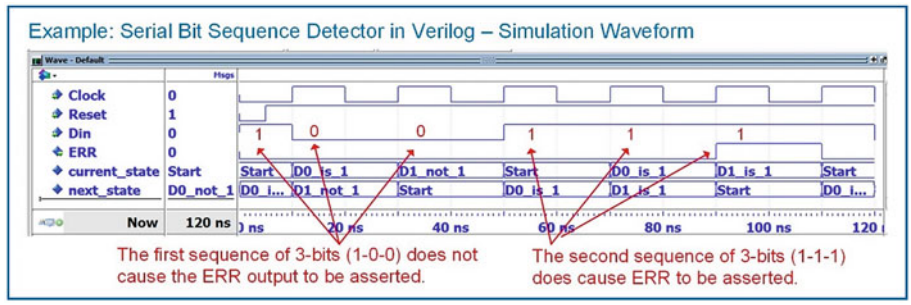
State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

Next state logic block. This is combinational logic so blocking assignments are used. This block only makes assignments to the signal "next_state".

Output logic block. This is combinational logic so blocking assignments are used. Since there is only one condition where the output "ERR" is asserted, the default clause can be used for all other conditions.

Example 8.7
Serial bit sequence detector in Verilog—full model

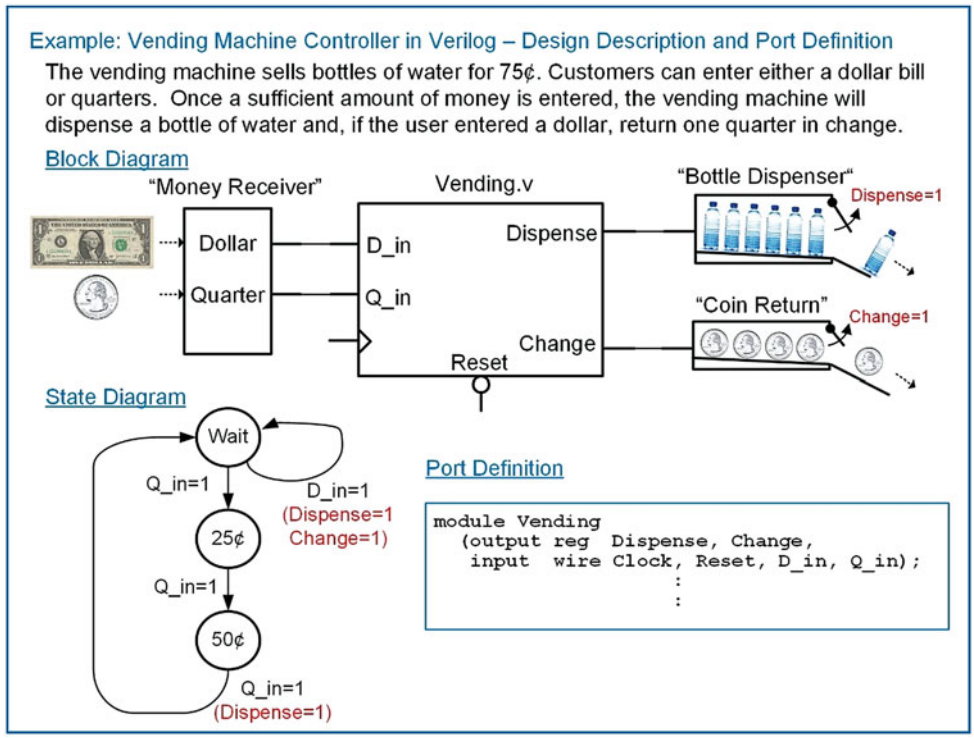
Example 8.8 shows the functional simulation waveform for this design.



Example 8.8
Serial bit sequence detector in Verilog—simulation waveform

8.2.2 Vending Machine Controller in Verilog

Let's now look at the design of the vending machine controller using the behavioral modeling constructs of Verilog. Example 8.9 shows the design description and port definition.



Example 8.9
Vending machine controller in Verilog—design description and port definition

Example 8.10 shows the full model for the vending machine controller. In this model, the descriptive state names Wait, 25¢, and 50¢ cannot be used directly. This is because Verilog user-defined names cannot begin with a number. Instead, the letter "s" is placed in front of the state names in order to make them legal Verilog names (i.e., sWait, s25, s50).

Example: Vending Machine Controller in Verilog – Full Model

```
module Vending (output reg  Dispense, Change,
                input  wire Clock, Reset, D_in, Q_in);

  reg [1:0] current_state, next_state;
  parameter sWait = 2'b00, s25 = 2'b01, s50 = 2'b10;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= sWait;
      else
        current_state <= next_state;
    end

  always @ (current_state or D_in or Q_in)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        sWait    : if (Q_in == 1'b1)
                      next_state = s25;
                   else
                      next_state = sWait;
        s25      : if (Q_in == 1'b1)
                      next_state = s50;
                   else
                      next_state = s25;
        s50      : if (Q_in == 1'b1)
                      next_state = sWait;
                   else
                      next_state = s50;
        default  : next_state = sWait;
      endcase
    end

   always @ (current_state or D_in or Q_in)
     begin: OUTPUT_LOGIC
       case (current_state)
         sWait    : if (D_in == 1'b1)
                       begin
                         Dispense = 1'b1; Change = 1'b1;
                       end
                    else
                       begin
                         Dispense = 1'b0; Change = 1'b0;
                       end
         s25      : begin
                       Dispense = 1'b0; Change = 1'b0;
                    end
         s50      : if (Q_in == 1'b1)
                       begin
                         Dispense = 1'b1; Change = 1'b0;
                       end
                    else
                       begin
                         Dispense = 1'b0; Change = 1'b0;
                       end
         default  : begin
                       Dispense = 1'b0; Change = 1'b0;
                    end
       endcase
     end
endmodule
```

State variables and state encoding.

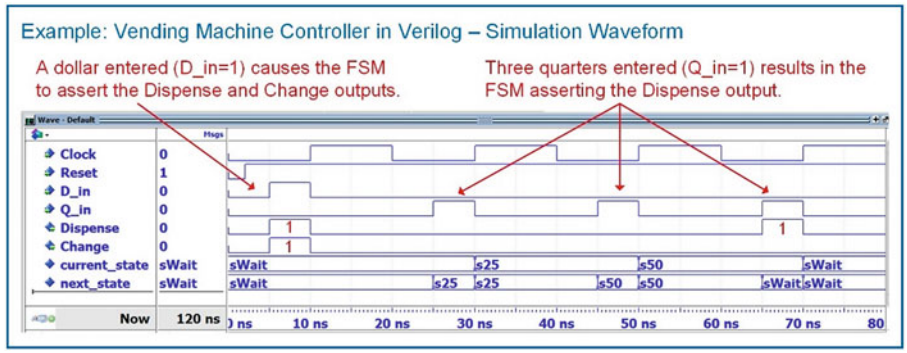
State memory block.

Next state logic block.

Output logic block.

Example 8.10
Vending machine controller in Verilog—full model

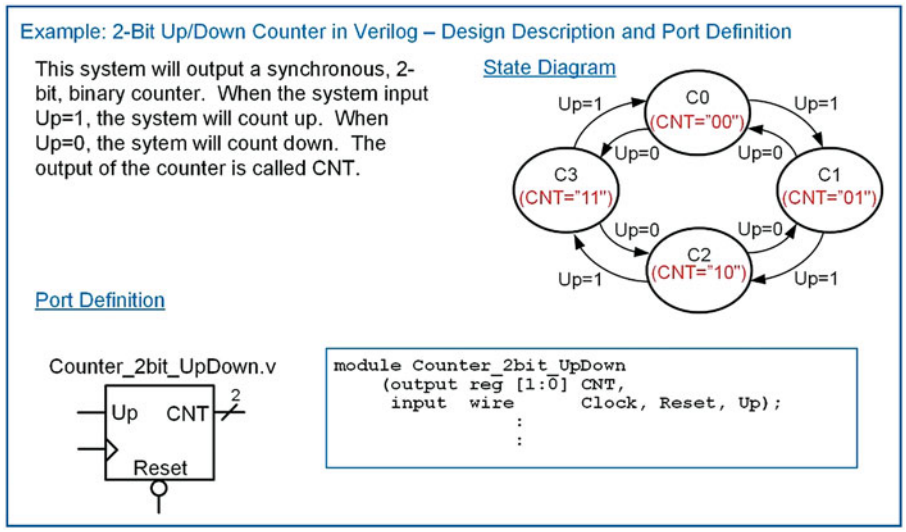
Example 8.11 shows the resulting simulation waveform for this design.



Example 8.11
Vending machine controller in Verilog—simulation waveform

8.2.3 2-Bit, Binary Up/Down Counter in Verilog

Let's now look at how a simple counter can be implemented using the three-block behavioral modeling approach in Verilog. Example 8.12 shows the design description and port definition for the 2-bit, binary up/down counter FSM from Chap. 7.



Example 8.12
2-Bit up/down counter in Verilog—design description and port definition

Example 8.13 shows the full model for the 2-bit up/down counter using the three-block modeling approach. Since a counter's outputs only depend on the current state, counters are Moore machines. This simplifies the output logic block since it only needs to contain the current state in its sensitivity list.

Example: 2-Bit Up/Down Counter in Verilog – Full Model (Three Block Approach)

```
module Counter_2bit_UpDown (output reg [1:0] CNT,
                            input  wire      Clock, Reset, Up);

  reg [1:0] current_state, next_state;
  parameter C0 = 2'b00,
            C1 = 2'b01,
            C2 = 2'b10,
            C3 = 2'b11;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= C0;
      else
        current_state <= next_state;
    end

  always @ (current_state or Up)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        C0      : if (Up == 1'b1) next_state = C1; else next_state = C3;
        C1      : if (Up == 1'b1) next_state = C2; else next_state = C0;
        C2      : if (Up == 1'b1) next_state = C3; else next_state = C1;
        C3      : if (Up == 1'b1) next_state = C0; else next_state = C2;
        default : next_state = C0;
      endcase
    end

  always @ (current_state)
    begin: OUTPUT_LOGIC
      case (current_state)
        C0      : CNT = 2'b00;
        C1      : CNT = 2'b01;
        C2      : CNT = 2'b10;
        C3      : CNT = 2'b11;
        default : CNT = 2'b00;
      endcase
    end

endmodule
```

State variables and state encoding.

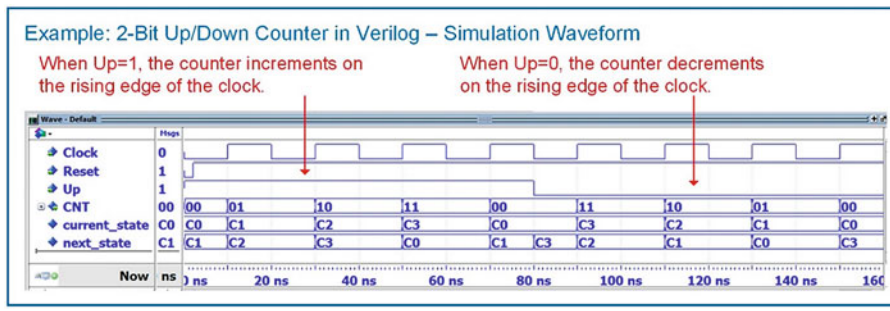
State memory block.

Next state logic block.

Output logic block. Note that since this is a Moore machine only the current state is listed in the sensitivity list.

Example 8.13
2-Bit up/down counter in Verilog—full model (three-block approach)

Example 8.14 shows the resulting simulation waveform for this counter finite state machine.



Example 8.14
2-Bit up/down counter in Verilog—simulation waveform

Concept Check

CC8.2 The procedural block for the state memory is nearly identical for all finite state machines with one exception. What is it?

(A) The sensitivity list may need to include a preset signal.

(B) Sometimes it is modeled using an SR latch storage approach instead of with D-flip-flop behavior.

(C) The name of the reset state will be different.

(D) The current_state and next_state signals are often swapped.

Summary

- Generic finite state machines are modeled using three separate procedural blocks that describe the behavior of the next state logic, the state memory, and the output logic. Separate blocks are used because each of the three functions in a FSM are dependent on different input signals.
- In Verilog, descriptive state names can be created for a FSM using parameters. Two signals are first declared called *current_state* and *next_state* of type reg. Then a parameter is defined for each unique state in the machine with the state name and desired state code. Throughout the rest of the model, the unique state names can be used as both assignments to current_state/next_state and as inputs in case and if-else statements. This approach allows the model to be designed using readable syntax while providing a synthesizable design.

Exercise Problems

Section 8.1: The FSM Design Process

8.1.1 What is the advantage of using *parameters* for the state when modeling a finite state machine?

8.1.2 What is the advantage of having to assign the state codes during the parameter declaration for the state names when modeling a finite state machine?

8.1.3 When using the three-procedural block behavioral modeling approach for FSMs, does the next state logic block model combinational or sequential logic?

8.1.4 When using the three-procedural block behavioral modeling approach for FSMs, does the state memory block model combinational or sequential logic?

8.1.5 When using the three-procedural block behavioral modeling approach for FSMs, does the output logic block model combinational or sequential logic?

8.1.6 When using the three-procedural block behavioral modeling approach for FSMs, what inputs are listed in the sensitivity list of the next state logic block?

8.1.7 When using the three-procedural block behavioral modeling approach for FSMs, what inputs are listed in the sensitivity list of the state memory block?

8.1.8 When using the three-procedural block behavioral modeling approach for FSMs, what inputs are listed in the sensitivity list of the output logic block?

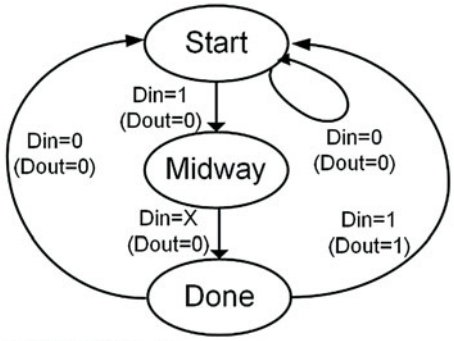
8.1.9 When using the three-procedural block behavioral modeling approach for FSMs, how can the signals listed in the sensitivity list of the output logic block immediately indicate whether the FSM is a Mealy or a Moore machine?

8.1.10 Why is it not a good design approach to combine the next state logic and output logic behavior into a single procedural block?

Section 8.2: FSM Design Examples

8.2.1 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.1. Use the port

definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in binary using the following state codes: Start = "00," Midway = "01," Done = "10."



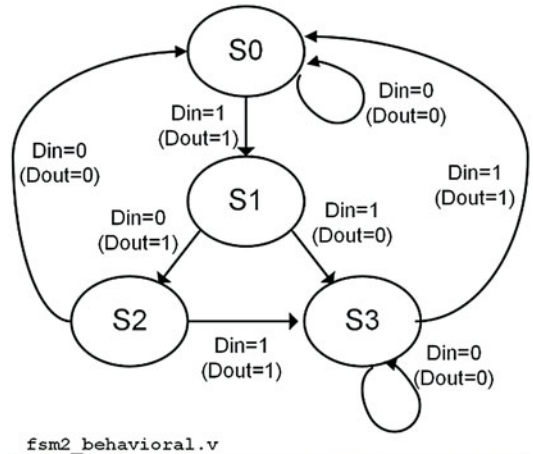
fsm1_behavioral.v

```
module fsm1_behavioral
   (output reg  Dout,
    input  wire Clock, Reset, Din);
              :
```

Fig. 8.1
FSM 1 State Diagram and Port Definition

8.2.2 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.1. Use the port definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in one-hot using the following state codes: Start = "001," Midway = "010," Done = "100."

8.2.3 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.2. Use the port definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in binary using the following state codes: S0 = "00," S1 = "01," S2 = "10," and S3 = "11."



fsm2_behavioral.v

```
module fsm2_behavioral
   (output reg  Dout,
    input  wire Clock, Reset, Din);
              :
```

Fig. 8.2
FSM 2 State Diagram and Port Definition

8.2.4 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.2. Use the port definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in one-hot using the following state codes: S0 = "0001," S1 = "0010," S2 = "0100," and S3 = "1000."

8.2.5 Design a Verilog behavioral model for a 4-bit serial bit sequence detector similar to Example 8.6. Use the port definition provided in Fig. 8.3. Use the three-block approach to modeling FSMs described in this chapter for your design. The input to your sequence detector is called *DIN* and the output is called *FOUND*. Your detector will assert FOUND anytime there is a 4-bit sequence of "0101." Model the states in this machine with parameters. Choose any state encoding approach you wish.

Seq_Det_behavioral.v

```
module Seq_Det_behavioral
   (output reg  FOUND,
    input  wire Clock, Reset,
    input  wire DIN);
              :
```

Fig. 8.3
Sequence Detector Port Definition

8.2.6 Design a Verilog behavioral model for a 20-¢ vending machine controller similar to Example 8.9. Use the port definition provided in Fig. 8.4. Use the three-block approach to modeling FSMs described in this chapter for your design. Your controller will take in nickels and dimes and dispense a product anytime the customer has entered 20¢. Your FSM has two inputs, *Nin* and *Din*. Nin is asserted whenever the customer enters a nickel while Din is asserted anytime the customer enters a dime. Your FSM has two outputs, *Dispense* and *Change*. Dispense is asserted anytime the customer has entered at least 20¢ and Change is asserted anytime the customer has entered more than 20¢ and needs a nickel in change. Model the states in this machine with parameters. Choose any state encoding approach you wish.

Vending_behavioral.v

```
module Vending_behavioral
   (output reg  Dispense, Change,
    input  wire Clock, Reset,
    input  wire Nin, Din);
              :
```

Fig. 8.4
Vending Machine Port Definition

8.2.7 Design a Verilog behavioral model for a finite state machine for a traffic light controller. Use the port definition provided in Fig. 8.5. This time, you will implement the functionality using the behavioral modeling techniques presented in this chapter. Your FSM will control a traffic light at the intersection of a busy highway and a seldom used side road. You will be designing the control signals for just the red, yellow, and green lights facing the highway. Under normal conditions, the highway has a green light. The side road has car detector that indicates when car pulls up by asserting a signal called *CAR*. When CAR is asserted, you will change the highway traffic light from green to yellow, and then from yellow to red. Once in the red position, a built-in timer will begin a countdown and provide your controller a signal called *TIMEOUT* when 15 s has passed. Once TIMEOUT is asserted, you will change the highway traffic light back to green. Your system will have three outputs *GRN*, *YLW*, and *RED*, which control when the highway facing traffic lights are on (1 = ON, 0 = OFF). Model the states in this machine with parameters. Choose any state encoding approach you wish.

tlc_behavioral.v

```
module tlc_behavioral
   (output reg  GRN, YLW, RED,
    input  wire Clock, Reset,
    input  wire CAR, TIMEOUT);
              :
```

Fig. 8.5
Traffic Light Controller Port Definition