Chapter 9: Behavioral Modeling of Sequential Logic

In this chapter, we will look at modeling sequential logic using the more sophisticated behavioral modeling techniques presented in Chap. 8. We will begin by looking at modeling sequential storage devices. Next, we will look at the behavioral modeling of finite-state machines. Finally, we will look at register transfer level, or RTL modeling. The goal of this chapter is to provide an understanding of how hardware description languages can be used to create behavioral models of synchronous digital systems.

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

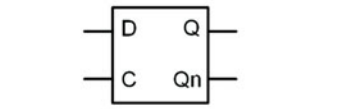
9.1 Design a Verilog behavioral model for a sequential logic storage device.
9.2 Describe the process for creating a Verilog behavioral model for a finite-state machine.
9.3 Design a Verilog behavioral model for a finite-state machine.
9.4 Design a Verilog behavioral model for a counter.
9.5 Design a Verilog register transfer level (RTL) model of a synchronous digital system.

9.1 Modeling Sequential Storage Devices in Verilog

9.1.1 D-Latch

Let's begin with the model of a simple D-Latch. Since the outputs of this sequential storage device are not updated continuously, its behavior is modeled using a procedural assignment. Since we want to create a synthesizable model of sequential logic, non-blocking assignments are used. In the sensitivity list, we need to include the C input since it controls when the D-Latch is in track or store mode. We also need to include the D input in the sensitivity list because during the track mode, the output Q will be assigned the value of D, so any change on D needs to trigger the procedural assignments. The use of an if-else statement is used to model the behavior during track mode (C = 1). Since the behavior is not explicitly stated for when C = 0, the outputs will hold their last value, which allows us to simply omit the else portion of the if statement to complete the model. Example 9.1 shows the behavioral model for a D-Latch.

Example: Behavioral Model of a D-Latch in Verilog



C	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	0	0	1	Track
1	1	1	0	Track

```
module dlatch (output reg  Q, Qn,
               input  wire C, D);

  always @ (C or D)
    if (C == 1'b1)
      begin
        Q  <= D;
        Qn <= ~D;
      end

endmodule
```

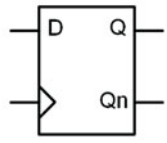
Example 9.1
Behavioral model of a D-latch in verilog

B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with Verilog*,
https://doi.org/10.1007/978-3-030-13605-5_9

9.1.2 D-Flip-Flop

The rising edge behavior of a D-flip-flop is modeled using a (posedge Clock) Boolean condition in the sensitivity list of a procedural block. Example 9.2 shows the behavioral model for a rising edge-triggered D-flip-flop with both Q and Qn outputs.

Example: Behavioral Model of a D-Flip-Flop in Verilog



Clk	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	X	Last Q	Last Qn	Store
↓	0	0	1	Update
↓	1	1	0	Update

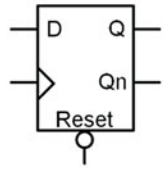
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, D);

  always @ (posedge Clock)
     begin
        Q  <= D;
        Qn <= ~D;
     end

endmodule
```

Example 9.2
Behavioral model of a D-flip-flop in verilog

9.1.3 D-Flip-Flop with Asynchronous Reset

D-flip-flops typically have a reset line to initialize their outputs to known states (e.g., Q = 0, Qn = 1). Resets are asynchronous, meaning whenever they are asserted, assignments to the outputs take place immediately. If a reset was *synchronous*, the outputs would only update on the next rising edge of the clock. This behavior is undesirable because if there is a system failure, there is no guarantee that a clock edge will ever occur. Thus, the reset may never take place. Asynchronous resets are more desirable not only to put the D-flip-flops into a known state at start-up, but also to recover from a system failure that may have impacted the clock signal. In order to model this asynchronous behavior, the reset signal is included in the sensitivity list. This allows both clock and the reset transitions to trigger the procedural block. The edge sensitivity of the reset can be specified using posedge (active HIGH) or negedge (active LOW). Within the block an if-else statement is used to determine whether the reset has been asserted or a rising edge of the clock has occurred. The if-else statement first checks whether the reset input has been asserted since it has the highest priority. If it has, it makes the appropriate assignments to the outputs (Q = 0, Qn = 1). If the reset has not been asserted, the *else* clause is executed, which corresponds to a rising edge of clock (Q <= D, Qn <= ~D). No other assignments are listed in the block; thus, the outputs are only updated on a transition of the reset or clock. At all other times, the outputs remain at their current value, thus modeling the store behavior of the D-flip-flop. Example 9.3 shows the behavioral model for a rising edge-triggered D-flip-flop with an asynchronous, active LOW reset.

Example: Behavioral Model of a D-Flip-Flop with Asynchronous Reset in Verilog



$\overline{R}$	Clk	D	Q	Qn	
0	X	X	0	1	Reset
1	0	X	Last Q	Last Qn	Store
1	1	X	Last Q	Last Qn	Store
1	↓	0	0	1	Update
1	↓	1	1	0	Update

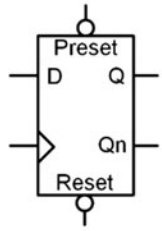
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, D);

  always @ (posedge Clock or negedge Reset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else
      begin
        Q  <= D;
        Qn <= ~D;
      end

endmodule
```

Example 9.3
Behavioral model of a D-flip-flop with asynchronous reset in verilog

9.1.4 D-Flip-Flop with Asynchronous Reset and Preset

A D-flip-flop with both an asynchronous reset and asynchronous preset is handled in a similar manner as the D-flip-flop in the prior section. The preset input is included in the sensitivity list in order to trigger the block whenever a transition occurs on either the clock, reset, or preset inputs. The edge sensitivity keywords are used to dictated whether the preset is active HIGH or LOW. Nested if-else statements are used to first check whether a reset has occurred; then whether a preset has occurred; and finally, whether a rising edge of the clock has occurred. Example 9.4 shows the model for a rising edge-triggered D-flip-flop with asynchronous, active LOW reset and preset.

Example: Behavioral Model of a D-Flip-Flop with Asynchronous Reset and Preset in Verilog



R	P	Clk	D	Q	Qn	
0	X	X	X	0	1	Reset
1	0	X	X	1	0	Preset
1	1	0	X	Last Q	Last Qn	Store
1	1	1	X	Last Q	Last Qn	Store
1	1	↓	0	0	1	Update
1	1	↓	1	1	0	Update

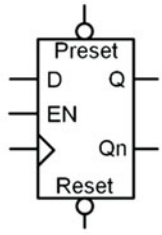
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, Preset, D);

  always @ (posedge Clock or negedge Reset or negedge Preset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else if (!Preset)
      begin
        Q  <= 1'b1;
        Qn <= 1'b0;
      end
    else
      begin
        Q  <= D;
        Qn <= ~D;
      end

endmodule
```

Example 9.4
Behavioral model of a D-flip-flop with asynchronous reset and preset in verilog

9.1.5 D-Flip-Flop with Synchronous Enable

An enable input is also a common feature of modern D-flip-flops. Enable inputs are synchronous, meaning that when they are asserted, action is only taken on the rising edge of the clock. This means that the enable input is not included in the sensitivity list of the always block. Since enable is only considered when there is a rising edge of the clock, the logic for the enable is handled in a nested if-else statement that is included in the section that models the behavior for when a rising edge of clock is detected. Example 9.5 shows the model for a D-flip-flop with a synchronous enable (EN) input. When EN = 1, the D-flip-flop is enabled, and assignments are made to the outputs only on the rising edge of the clock. When EN = 0, the D-flip-flop is disabled and assignments to the outputs are not made. When disabled, the D-flip-flop effectively ignores rising edges on the clock and the outputs remain at their last values.

Example: Behavioral Model of a D-Flip-Flop with Synchronous Enable in Verilog



$\overline{R}$	$\overline{P}$	Clk	EN	D	Q	Qn	
0	X	X	X	X	0	1	Reset
1	0	X	X	X	1	0	Preset
1	1	0	X	X	Last Q	Last Qn	Store
1	1	1	X	X	Last Q	Last Qn	Store
1	1	↓	0	X	Last Q	Last Qn	Disabled (ignore clock)
1	1	↓	1	0	0	1	Update
1	1	↓	1	1	1	0	Update

```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, Preset, D, EN);

  always @ (posedge Clock or negedge Reset or negedge Preset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else if (!Preset)
      begin
        Q  <= 1'b1;
        Qn <= 1'b0;
      end
    else
      if (EN)
        begin
          Q  <= D;
          Qn <= ~D;
        end

endmodule
```

Since EN is not listed in the sensitivity list it does not trigger the block when it transitions. This "if" statement is only reached if there is a rising edge of the clock. This models an enable that is synchronous to the clock.

Example 9.5
Behavioral model of a D-flip-flop with synchronous enable in verilog

Concept Check

CC9.1 Why is the D input not listed in the sensitivity list of a D-flip-flop?

A) To simplify the behavioral model.

B) To avoid a setup time violation if D transitions too closely to the clock.

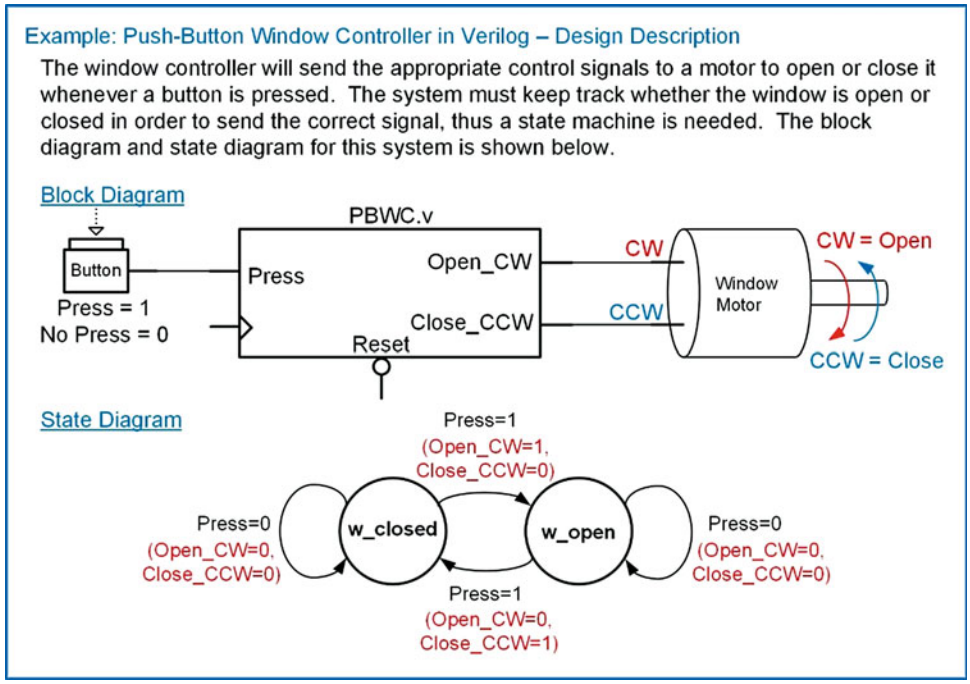
C) Because a rising edge of clock is needed to make the assignment.

D) Because the outputs of the D-flip-flop are not updated when D changes.

9.2 Modeling Finite-State Machines in Verilog

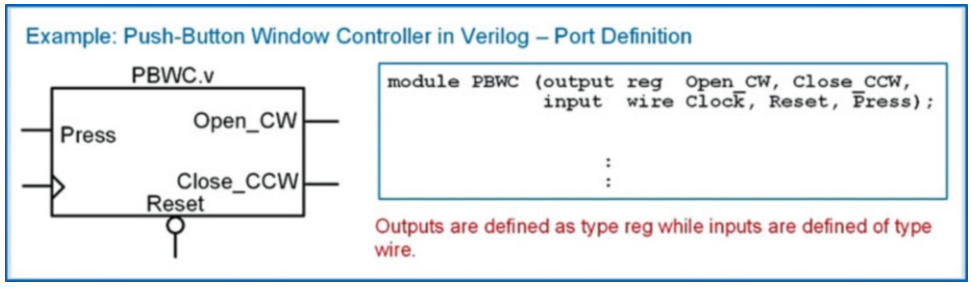
Finite-state machines can be easily modeled using the behavioral constructs from Chap. 8. The most common modeling practice for FSMs is to declare two signals of type reg that are called *current_state* and *next_state*. Then a parameter is declared for each descriptive state name in the state diagram. 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 9.6 gives the overview of the design objectives for this example and the state diagram describing the behavior to be modeled in Verilog.



Example 9.6
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 9.7 shows the port definitions for this example. Note that outputs are declared as type reg, while inputs are declared as type wire.



Example 9.7
Push-button window controller in verilog: port definition

9.2.1 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;
```

9.2.2 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
```

9.2.3 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
```

9.2.4 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 9.8 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.

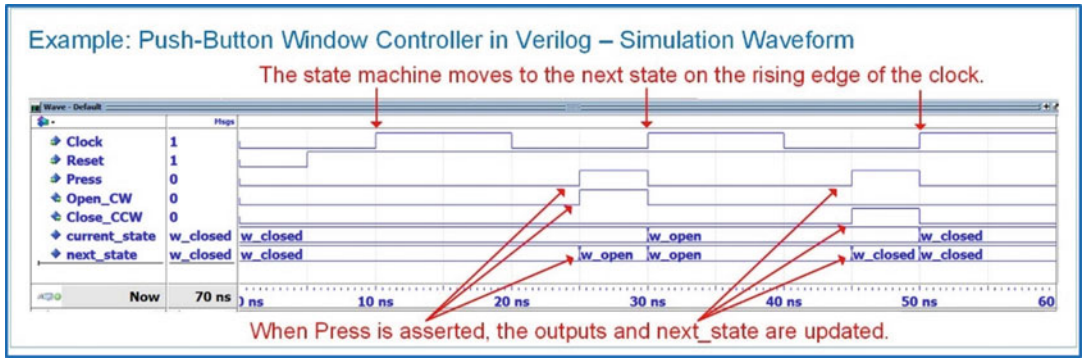
State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

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 9.8
Push-button window controller in verilog: full model

Example 9.9 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.

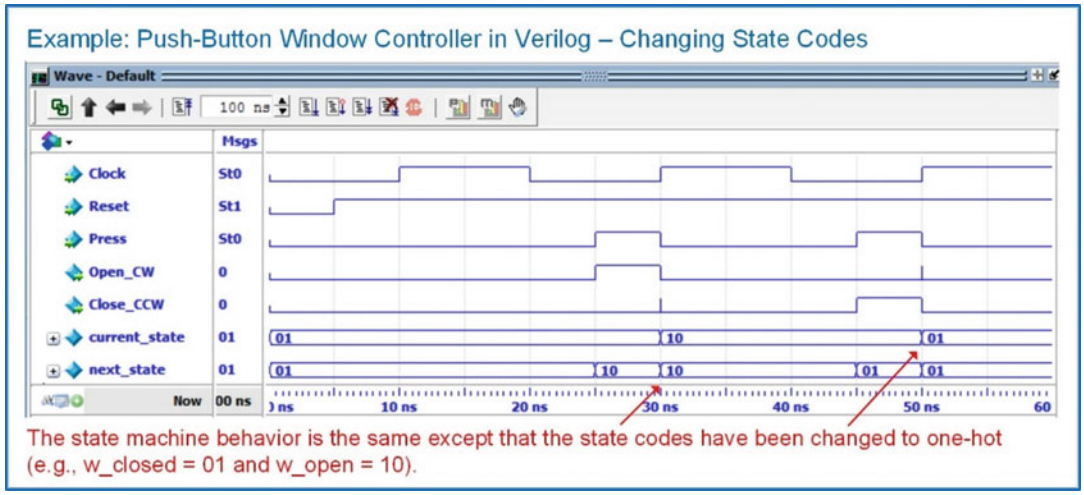


Example 9.9
Push-button window controller in verilog: simulation waveform

9.2.5 Changing the State Encoding Approach

In the prior example, we only had two states and they were encoded as w_closed = 1'b0; w_open_1'b1. This encoding technique is considered *binary*; however, a *gray code* approach would yield the same codes since the width of the variables were only one bit. 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 9.10 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 9.10
Push-button window controller in verilog: changing state codes

CONCEPT CHECK

CC9.2 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.

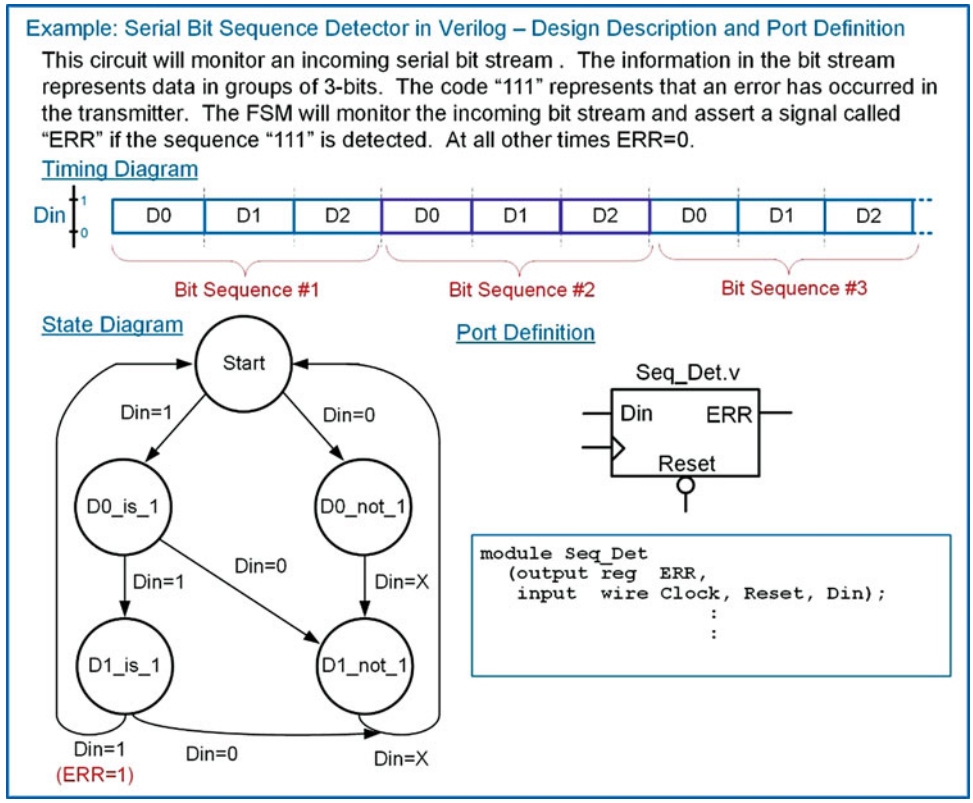
D) Because each of the three sub-systems of a FSM has unique inputs and outputs that should be handled using dedicated processes.

9.3 FSM Design Examples in Verilog

This section presents a set of example finite-state machine designs using the behavioral modeling constructs of Verilog. These examples are the same state machines that were presented in Chap. 7.

9.3.1 Serial Bit Sequence Detector in Verilog

Let's look at the design of the serial bit sequence detector finite-state machine from Chap. 7 using the behavioral modeling constructs of Verilog. Example 9.11 shows the design description and port definition for this state machine.



Example 9.11
Serial bit sequence detector in verilog: design description and port definition

Example 9.12 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.

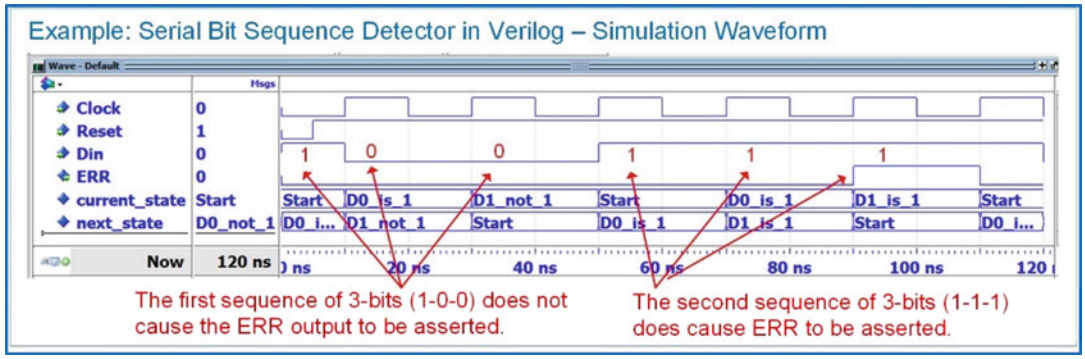
State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

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 9.12
Serial bit sequence detector in verilog: full model

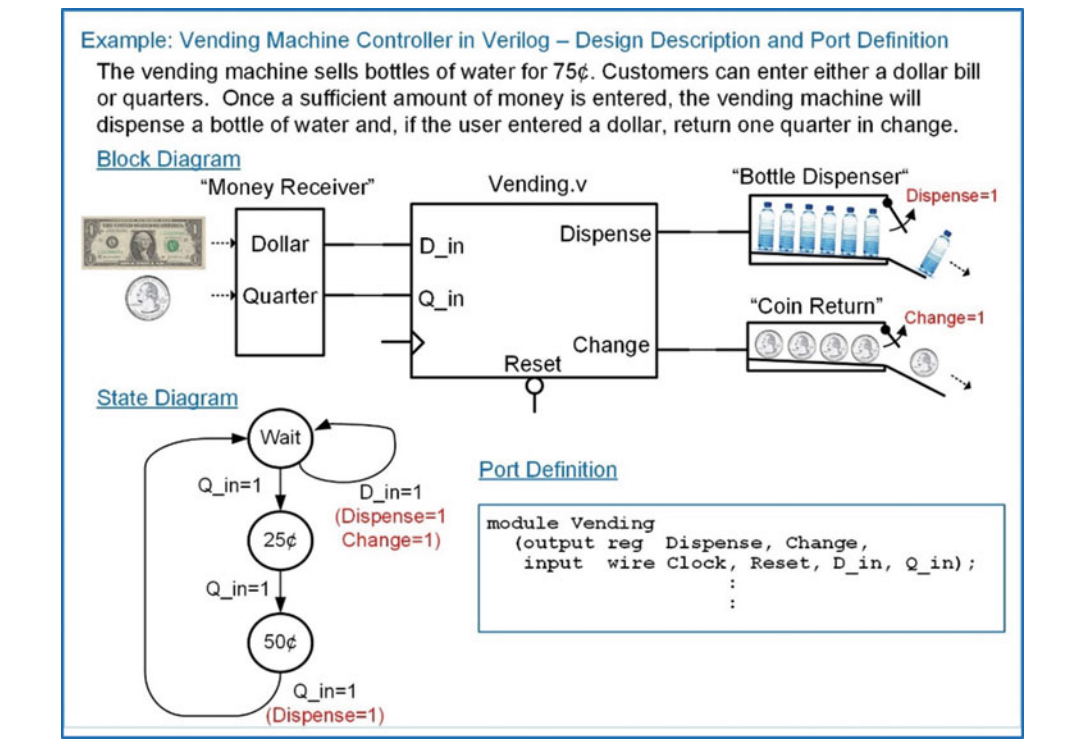
Example 9.13 shows the functional simulation waveform for this design.



Example 9.13
Serial bit sequence detector in verilog: simulation waveform

9.3.2 Vending Machine Controller in Verilog

Let's now look at the design of the vending machine controller from Chap. 7 using the behavioral modeling constructs of Verilog. Example 9.14 shows the design description and port definition.



Example 9.14
Vending machine controller in verilog: design description and port definition

Example 9.15 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.

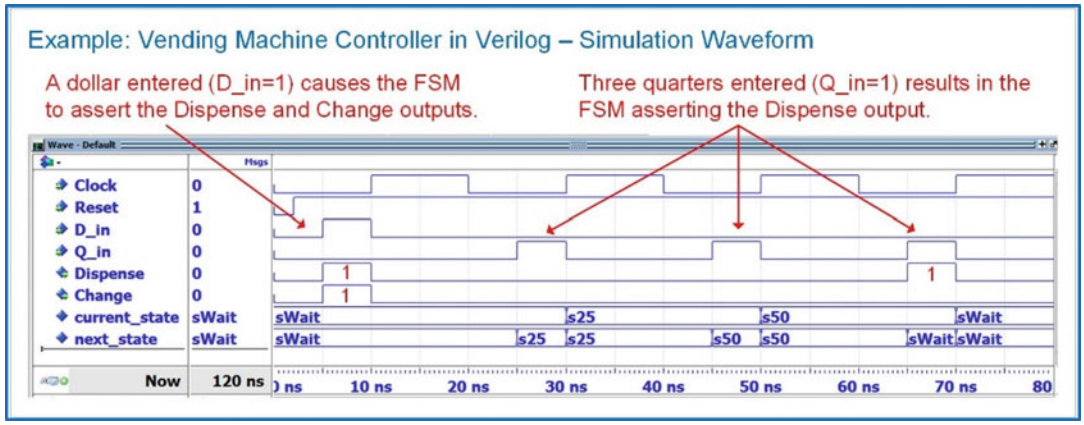
State memory block.

Next state logic block.

Output logic block.

Example 9.15
Vending machine controller in verilog: full model

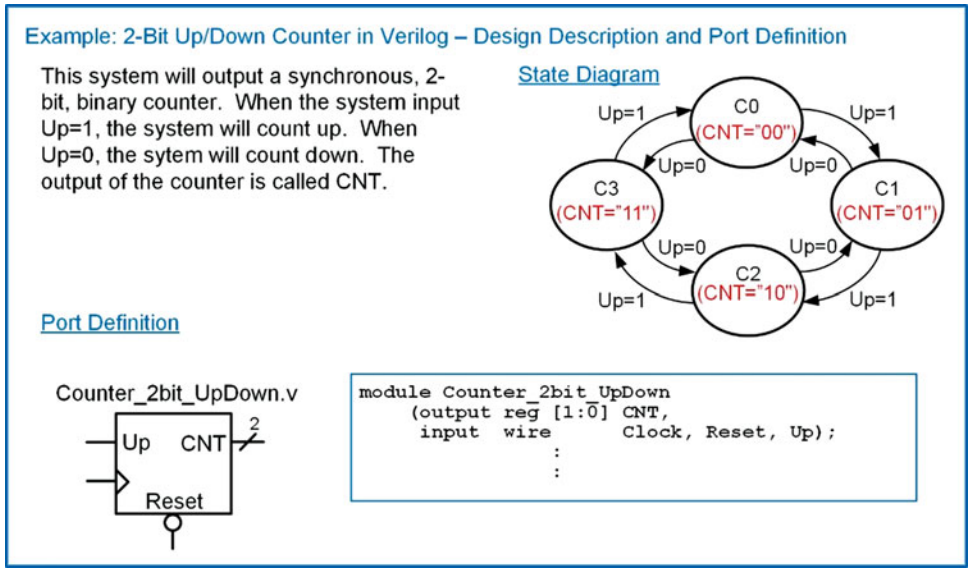
Example 9.16 shows the resulting simulation waveform for this design.



Example 9.16
Vending machine controller in verilog: simulation waveform

9.3.3 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 9.17 shows the design description and port definition for the 2-bit, binary up/down counter FSM from Chap. 7.



Example 9.17
2-bit up/down counter in verilog: design description and port definition

Example 9.18 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.

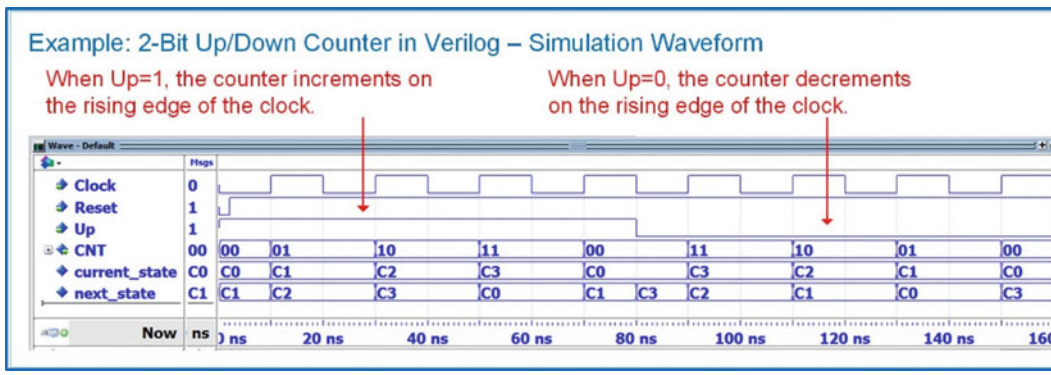
Next state logic block.

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

Example 9.18
2-bit up/down counter in verilog: full model (three-block approach)

Example 9.19 shows the resulting simulation waveform for this counter finite-state machine.

Example: 2-Bit Up/Down Counter in Verilog – Simulation Waveform



Example 9.19
2-bit up/down counter in verilog: simulation waveform

Concept Check

CC9.3 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.

9.4 Modeling Counters in Verilog

Counters are a special case of finite-state machines because they move linearly through their discrete states (either forward or backward) and typically are implemented with state-encoded outputs. Due to this simplified structure and widespread use in digital systems, Verilog allows counters to be modeled using a single-procedural block with arithmetic operators (i.e., + and −). This enables a more compact model and allows much wider counters to be implemented in a practical manner.

9.4.1 Counters in Verilog Using a Single-Procedural Block

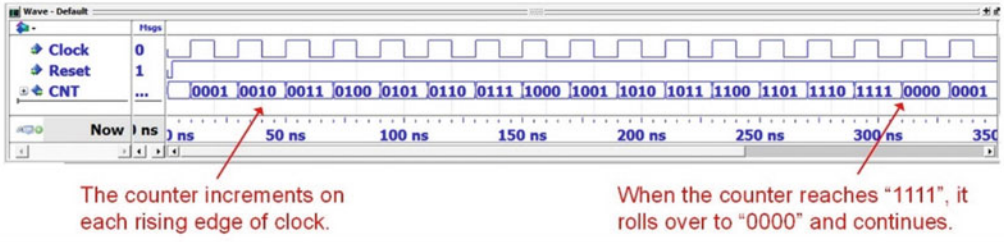
Let's look at how we can model a 4-bit, binary up counter with an output called *CNT*. We want to model this counter using the "+" operator to avoid having to explicitly define a state code for each state as in the three-block modeling approach to FSMs. The "+" operator works on the type reg, so the counting behavior can simply be modeled using CNT <= CNT + 1. The procedural block also needs to handle the reset condition. Both the Clock and Reset signals are listed in the sensitivity list. Within the block, an if-else statement is used to handle both the reset and increment behaviors. Example 9.20 shows the Verilog model and simulation waveform for this counter. When the counter reaches its maximum value of "1111," it rolls over to "0000" and continues counting because it is declared to only contain 4-bits.

Example: Binary Counter using a Single Procedural Block in Verilog

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire      Clock, Reset);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        CNT <= CNT + 1;
    end

endmodule
```

A single procedural block handles the reset condition and the increment behavior.

Using decimal format for numbers makes the model more readable.



The counter increments on each rising edge of clock.

When the counter reaches "1111", it rolls over to "0000" and continues.

Example 9.20
Binary counter using a single-procedural block in verilog

9.4.2 Counters with Range Checking

When a counter needs to have a maximum range that is different from the maximum binary value of the count vector (i.e., $<2^n-1$), then the procedural block needs to contain *range checking* logic. This can be modeled by inserting a nested if-else statement beneath of the else clause that handles the behavior for when the counter receives a rising clock edge. This nested if-else first checks whether the count has reached its maximum value. If it has, it is reset back to it minimum value. If it hasn't, the counter is incremented as usual. Example 9.21 shows the Verilog model and simulation waveform for a counter with a minimum count value of 0_{10} and a maximum count value of 10_{10}. This counter still requires 4-bits to be able to encode 10_{10}.

Example: Binary Counter with Range Checking in Verilog

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire      Clock, Reset);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (CNT == 10)
          CNT <= 0;
        else
          CNT <= CNT + 1;
    end

endmodule
```

A nested if-else statement checks if the counter has reached its maximum value. If it has, it is reset back to zero. If it hasn't, it increments.

Once the counter reaches 10, it is set back to 0. In this waveform, the radix of the counter is formatted as unsigned decimal.

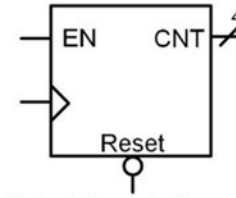
Example 9.21
Binary counter with range checking in verilog

9.4.3 Counters with Enables in Verilog

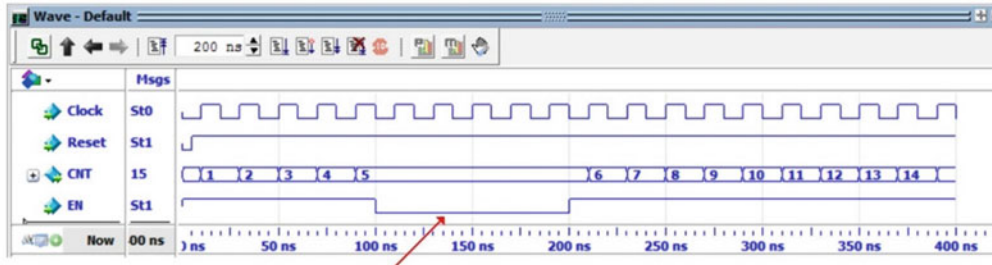
Including an *enable* in a counter is a common technique to prevent the counter from running continuously. When the enable is asserted, the counter will increment on the rising edge of the clock as usual. When the enable is de-asserted, the counter will simply hold its last value. Enable lines are synchronous, meaning that they are only evaluated on the rising edge of the clock. As such, they are modeled using a nested if-else statement within the main if-else statement checking for a rising edge of the clock. Example 9.22 shows an example model for a 4-bit counter with enable.

Example: Binary Counter with Enable in Verilog

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire      Clock, Reset, EN);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (EN)
          CNT <= CNT + 1;
    end

endmodule
```



The EN is synchronous to the clock, so its logic is nested beneath the portion of the main if-else clause that handles the behavior when the counter receives a rising edge of clock.



When the counter is NOT enabled, it will hold its last value.

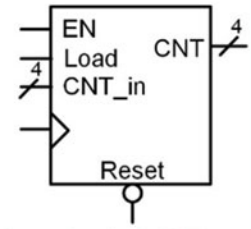
Example 9.22
Binary counter with enable in verilog

9.4.4 Counters with Loads

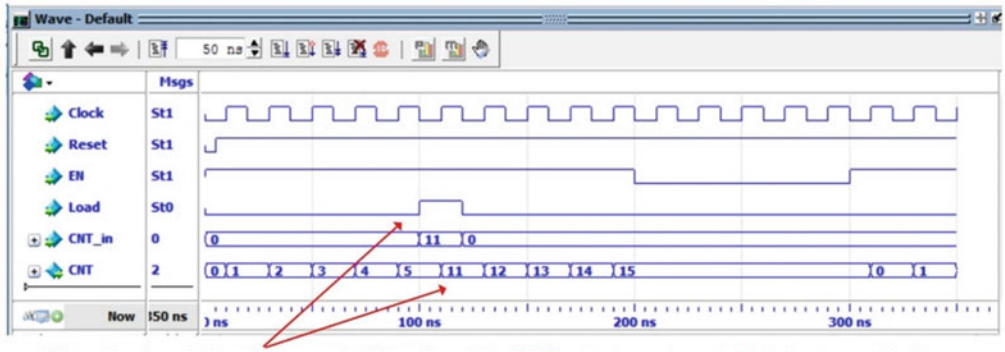
A counter with a *load* has the ability to set the counter to a specified value. The specified value is provided on an input port (i.e., CNT_in) with the same width as the counter output (CNT). A synchronous load input signal (i.e., Load) is used to indicate when the counter should set its value to the value present on CNT_in. Example 9.23 shows an example model for a 4-bit counter with load capability.

Example: Binary Counter with Load in Verilog

```
module Counter_4bit_Up (output reg [3:0]  CNT,
                        input  wire       Clock, Reset, EN, Load,
                        input  wire [3:0] CNT_in);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (EN)
          if (Load)
            CNT <= CNT_in;
          else
            CNT <= CNT + 1;
    end

endmodule
```



A nested if-else statement is used to load CNT with CNT_in when the Load signal is asserted and the counter receives a rising edge of clock.



When the Load signal is asserted, it will update CNT with the value of CNT_in (e.g., "11_{10}").

Example 9.23
Binary counter with load in verilog

Concept Check

CC9.4 If a counter is modeled using only one procedural block in Verilog, is it still a finite-state machine? Why or why not?

A) Yes. It is just a special case of a FSM that can easily be modeled using one block. Synthesizers will recognize the single block model as a FSM.

B) No. Using only one block will synthesize into combinational logic. Without the ability to store a state, it is not a finite-state machine.

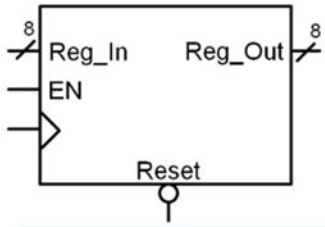
9.5 RTL Modeling

Register transfer level modeling refers to a level of design abstraction in which vector data is moved and operated on in a synchronous manner. This design methodology is widely used in data path modeling and computer system design.

9.5.1 Modeling Registers in Verilog

The term *register* describes a group of D-Flip-Flops running off of the same clock, reset, and enable inputs. Data is moved in and out of the bank of D-flip-flops as a vector. Logic operations can be made on the vectors and are latched into the register on a clock edge. A register is a higher level of abstraction that allows vector data to be stored without getting into the details of the lower-level implementation of the D-flip-flops and combinational logic. Example 9.24 shows an RTL model of an 8-bit, synchronous register. This circuit has an active LOW, asynchronous reset that will cause the 8-bit output *Reg_Out* to go to 0 when it is asserted. When the reset is not asserted, the output will be updated with the 8-bit input *Reg_In* if the system is enabled (EN = 1), and there is a rising edge on the clock. If the register is disabled (EN = 0), the input clock is ignored. At all other times, the output holds its last value.

Example: RTL Model of an 8-Bit Register in Verilog



$\overline{R}$	Clk	EN	Reg_Out	
0	X	X	x"00"	Reset
1	X	0	Last Reg_Out	Disabled (ignore clock)
1	0	1	Last Reg_Out	Store
1	1	1	Last Reg_Out	Store
1	↓	1	Reg_In	Update

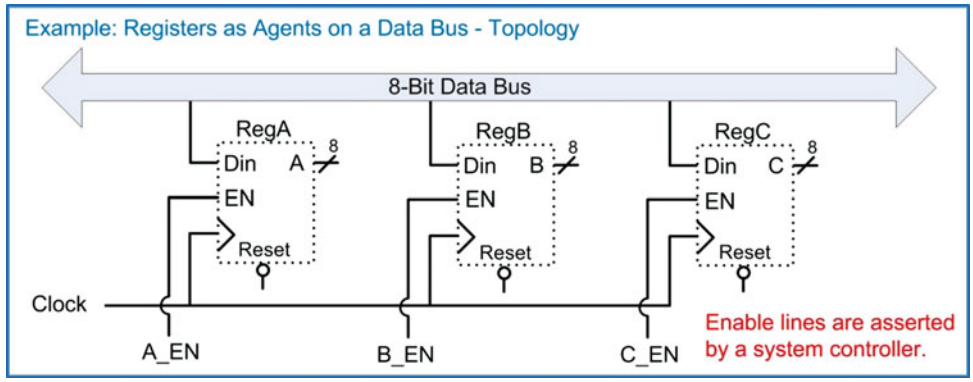
```
module RegX (output reg [7:0]  Reg_Out,
             input  wire       Clock, Reset, EN,
             input  wire [7:0] Reg_In);

  always @ (posedge Clock or negedge Reset)
    begin: REGISTER
      if (!Reset)
        Reg_Out <= 8'h00;
      else
        if (EN)
          Reg_Out <= Reg_In;
    end

endmodule
```

When Enabled (EN=1), the register will latch in the input value on the rising edge of clock.

Example 9.24
RTL model of an 8-bit register in verilog

9.5.2 Registers as Agents on a Data Bus

One of the powerful topologies that registers can easily model is a multi-drop bus. In this topology, multiple registers are connected to a data bus as receivers, or *agents*. Each agent has an enable line that controls when it latches information from the data bus into its storage elements. This topology is synchronous, meaning that each agent and the driver of the data bus is connected to the same clock signal. Each agent has a dedicated, synchronous enable line that is provided by a system controller elsewhere in the design. Example 9.25 shows this multi-drop bus topology. In this example system, three registers (A, B, and C) are connected to a data bus as receivers. Each register is connected to the same clock and reset signals. Each register has its own dedicated enable line (A_EN, B_EN, and C_EN).



Example 9.25
Registers as agents on a data bus: system topology

This topology can be modeled using RTL abstraction by treating each register as a separate procedural block. Example 9.26 shows how to describe this topology with an RTL model in Verilog. Notice that the three procedural blocks modeling the A, B, and C registers are nearly identical to each other except for the signal names they use.

Example: Registers as Agents on a Data Bus – RTL Model in Verilog

```
module MultiDropBus
   (output reg [7:0]  A, B, C,
    input  wire       Clock, Reset,
    input  wire [7:0] Data_Bus,
    input  wire       A_EN, B_EN, C_EN);

  always @ (posedge Clock or negedge Reset)
    begin: A_REG
      if (!Reset)
        A <= 8'h00;
      else
        if (A_EN ==1)
          A <= Data_Bus;
    end

  always @ (posedge Clock or negedge Reset)
    begin: B_REG
      if (!Reset)
        B <= 8'h00;
      else
        if (B_EN ==1)
          B <= Data_Bus;
    end

  always @ (posedge Clock or negedge Reset)
    begin: C_REG
      if (!Reset)
        C <= 8'h00;
      else
        if (C_EN ==1)
          C <= Data_Bus;
    end

endmodule
```

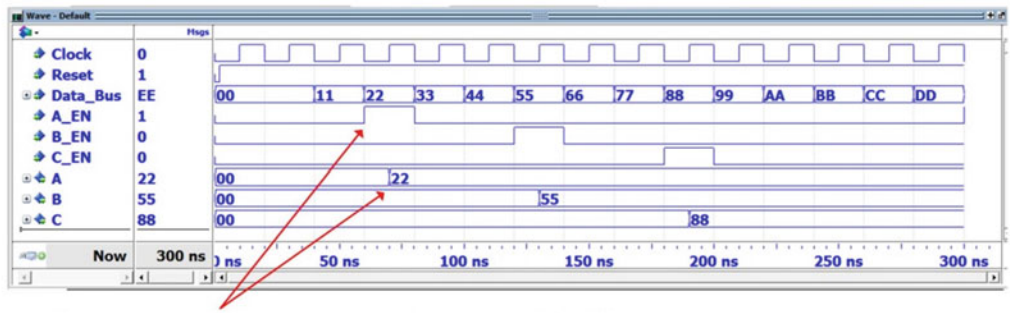
Each register is modeled as a separate block. The register has a synchronous enable that controls when it acquires data off of the data bus.

All registers are attached to the data bus as receivers.

Example 9.26
Registers as agents on a data bus: RTL model in verilog

Example 9.27 shows the resulting simulation waveform for this system. Each register is updated with the value on the data bus whenever its dedicated enable line is asserted.

Example: Registers as Agents on a Data Bus – Simulation Waveform



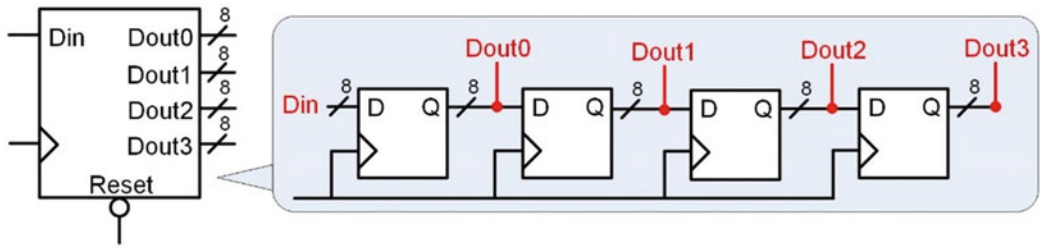
When a register's synchronous enable is asserted, it will latch the value of data_bus on the next rising edge of clock.

Example 9.27
Registers as agents on a data bus: simulation waveform

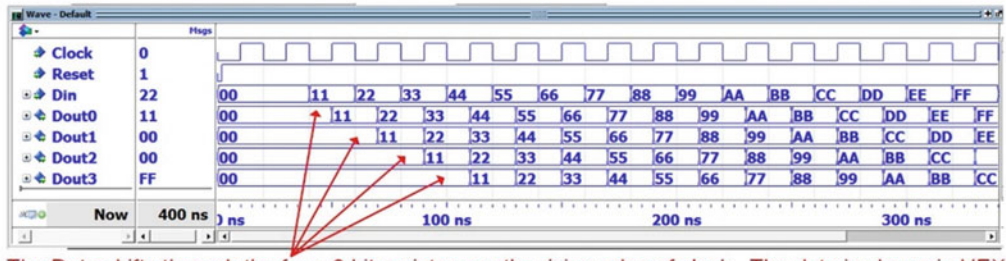
9.5.3 Shift Registers in Verilog

A shift register is a circuit which consists of multiple registers connected in series. Data is shifted from one register to another on the rising edge of the clock. This type of circuit is often used in serial-to-parallel data converters. Example 9.28 shows an RTL model for a 4-stage, 8-bit shift register.

Example: RTL Model of a 4-Stage, 8-Bit Shift Register in Verilog




```
module Shift_Register
    (output reg [7:0]  Dout0, Dout1, Dout2, Dout3,
     input  wire       Clock, Reset,
     input  wire [7:0] Din);

  always @ (posedge Clock or negedge Reset)
    begin: SHIFT_REGISTER
      if (!Reset)
        begin
          Dout0 <= 8'h00; Dout1 <= 8'h00; Dout2 <= 8'h00; Dout3 <= 8'h00;
        end
      else
        begin
          Dout0 <= Din;   Dout1 <= Dout0; Dout2 <= Dout1; Dout3 <= Dout2;
        end
    end

endmodule
```



The Data shifts through the four, 8-bit registers on the rising edge of clock. The data is shown in HEX.

Example 9.28
RTL model of a 4-stage, 8-bit shift register in verilog

Concept Check

CC9.5 Does RTL modeling synthesize as combinational logic, sequential logic, or both? Why?

A) Combinational logic. Since only one process is used for each register, it will be synthesized using basic gates.

B) Sequential logic. Since the sensitivity list contains clock and reset, it will synthesize into only D-flip-flops.

C) Both. The model has a sensitivity list containing clock and reset and uses an if-else statement indicative of a D-flip-flop. This will synthesize a D-flip-flop to hold the value for each bit in the register. In addition, the ability to manipulate the inputs into the register (using either logical operators, arithmetic operators, or choosing different signals to latch) will synthesize into combinational logic in front of the D input to each D-flip-flop.

Summary

- A synchronous system is modeled with a procedural block and a sensitivity list. The clock and reset signals are always listed by themselves in the sensitivity list. Within the block is an if-else statement. The *if* clause of the statement handles the asynchronous reset condition, while the *else* clause handles the synchronous signal assignments.
- Edge sensitivity is modeled within a procedural block using the *(posedge Clock or negedge reset)* syntax in the sensitivity lists.
- Most D-flip-flops and registers contain a synchronous *enable* line. This is modeled using a nested if-else statement within the main procedural block's if-else statement. The nested if-else goes beneath the clause for the synchronous signal assignments.
- 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 is 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.
- Counters are a special type of finite-state machine that can be modeled using a single-procedural block. Only the clock and reset signals are listed in the sensitivity list of the counter block.
- Registers are modeled in Verilog in a similar manner to a D-flip-flop with a synchronous enable. The only difference is that the inputs and outputs are vectors.
- Register transfer level, or RTL, modeling provides a higher level of abstraction for moving and manipulating vectors of data in a synchronous manner.

Exercise Problems

Section 9.1: Modeling Sequential Storage Devices in Verilog

9.1.1 How does a Verilog model for a D-flip-flop handle treating reset as the highest priority input?

9.1.2 For a Verilog model of a D-flip-flop with a synchronous enable (EN), why isn't EN listed in the sensitivity list?

9.1.3 For a Verilog model of a D-flip-flop with a synchronous enable (EN), what is the impact of listing EN in the sensitivity list?

9.1.4 For a Verilog model of a D-flip-flop with a synchronous enable (EN), why is the behavior of the enable modeled using a nested if-else statement under the else clause handling the logic for the clock edge input?

Section 9.2: Modeling Finite-State Machines in Verilog

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

9.2.2 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?

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

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

9.2.5 When using the three-procedural block behavioral modeling approach for finite-state machines, does the output logic block model combinational or sequential logic?

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

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

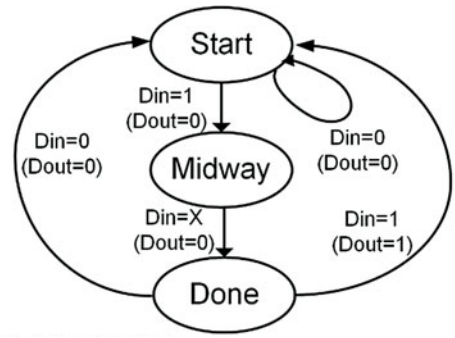
9.2.8 When using the three-procedural block behavioral modeling approach for finite-state machines, what inputs are listed in the sensitivity list of the output logic block?

9.2.9 When using the three-procedural block behavioral modeling approach for finite-state machines, 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?

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

Section 9.3: FSM Design Examples in Verilog

9.3.1 Design a Verilog behavioral model to implement the finite-state machine described by the state diagram in Fig. 9.1. 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."



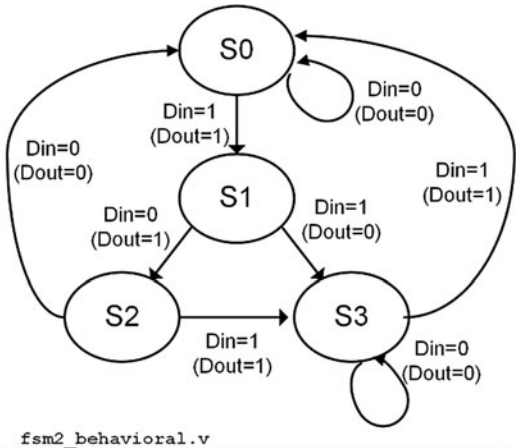
fsm1_behavioral.v

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

Fig. 9.1
FSM 1 state diagram and module definition

9.3.2 Design a Verilog behavioral model to implement the finite-state machine described by the state diagram in Fig. 9.1. 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."

9.3.3 Design a Verilog behavioral model to implement the finite-state machine described by the state diagram in Fig. 9.2. 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. 9.2
FSM 2 state diagram and module definition

9.3.4 Design a Verilog behavioral model to implement the finite-state machine described by the state diagram in Fig. 9.2. 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,"

9.3.5 Design a Verilog behavioral model for a 4-bit serial bit sequence detector similar to Example 9.11. Use the port definition provided in Fig. 9.3. 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. 9.3
Sequence detector module definition

9.3.6 Design a Verilog behavioral model for a 20-cent vending machine controller similar to Example 9.14. Use the port definition provided in Fig. 9.4. 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 cents. 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 cents and Change is asserted anytime the customer has entered more than 20 cents 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. 9.4
Vending machine module definition

9.3.7 Design a Verilog behavioral model for a finite-state machine for a traffic light controller. Use the port definition provided in Fig. 9.5. This is the same problem description as in exercise 7.4.15. 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 seconds 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. 9.5
Traffic light controller module definition

Section 9.4: Modeling Counters in Verilog

9.4.1 Design a Verilog behavioral model for a 16-bit, binary up counter using a single-procedural block. The block diagram for the port definition is shown in Fig. 9.6.

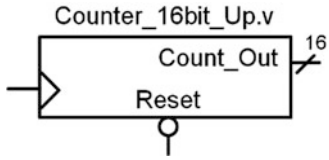


Fig. 9.6
16-bit binary up counter block diagram

9.4.2 Design a Verilog behavioral model for a 16-bit, binary up counter with range checking using a single-procedural block. The block diagram for the port definition is shown in Fig. 9.6. Your counter should count up to 60,000 and then start over at 0.

9.4.3 Design a Verilog behavioral model for a 16-bit, binary up counter with enable using a single-procedural block. The block diagram for the port definition is shown in Fig. 9.7.

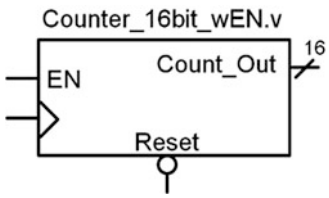


Fig. 9.7
16-bit binary up counter with enable block diagram

9.4.4 Design a Verilog behavioral model for a 16-bit, binary up counter with enable and load using a single-procedural block. The block diagram for the port definition is shown in Fig. 9.8.

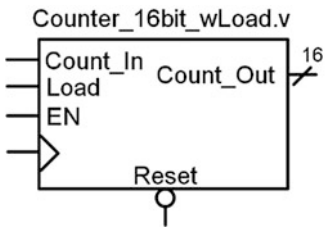


Fig. 9.8
16-bit binary up counter with load block diagram

9.4.5 Design a Verilog behavioral model for a 16-bit, binary up/down counter using a single-procedural block. The block diagram for the port definition is shown in Fig. 9.9. When Up = 1, the counter will increment. When Up = 0, the counter will decrement.

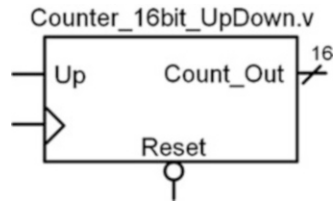


Fig. 9.9
16-bit binary up/down counter block diagram

Section 9.5: RTL Modeling

9.5.1 In *register transfer level* modeling, how does the width of the register relate to the number of D-flip-flops that will be synthesized?

9.5.2 In *register transfer level* modeling, how is the synchronous data movement managed if all registers are using the same clock?

9.5.3 Design a Verilog RTL model of a 32-bit, synchronous register. The block diagram for the port definition is shown in Fig. 9.10. The register has a synchronous enable. The register should be modeled using a single-procedural block.

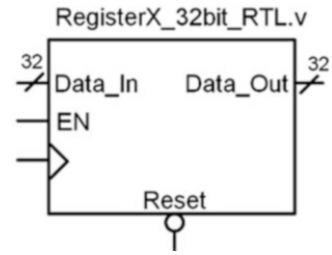


Fig. 9.10
32-bit register block diagram

9.5.4 Design a Verilog RTL model of an 8-stage, 16-bit shift register. The block diagram for the port definition is shown in Fig. 9.11. Each stage of the shift register will be provided as an output of the system (A, B, C, D, E, F, G, and H). The shift register should be modeled using a single-procedural block.

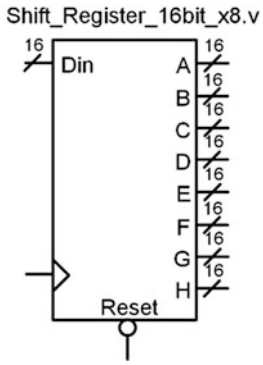


Fig. 9.11
16-bit shift register block diagram

9.5.5 Design a Verilog RTL model of the multi-drop bus topology in Fig. 9.12. Each of the 16-bit registers (RegA, RegB, RegC, and RegD) will latch the contents of the 16-bit data bus if their enable line is asserted. Each register should be modeled using an individual procedural block.

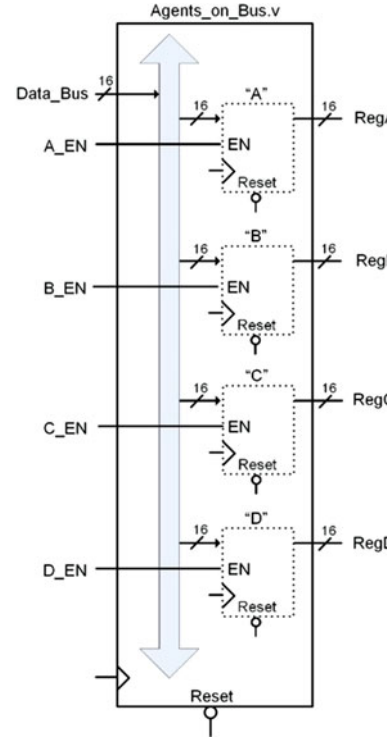


Fig. 9.12
Agents on a bus block diagram