Chapter 9: 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:

9.1 Describe the three-process modeling approach for FSM design.
9.2 Design a VHDL model for a FSM from a state diagram.

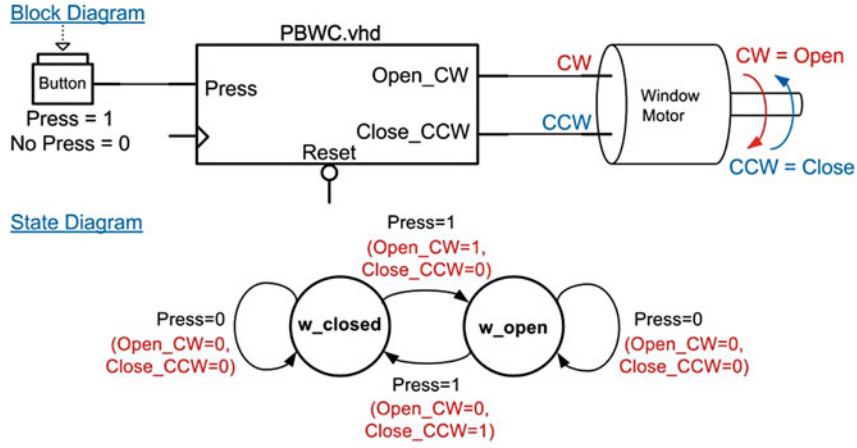
9.1 The FSM Design Process and a Push-Button Window Controller Example

The most common modeling practice for FSMs is to create a new *user-defined* type that can take on the descriptive state names from the state diagram. Two signals are then created of this type, *current_state* and *next_state*. Once these signals are created, all of the functional blocks in the state machine can use the descriptive state names in their conditional signal assignments. The synthesizer will automatically assign the state codes based on the most effective use of the target technology (e.g., binary, gray code, one-hot). Within the VHDL state machine model, three processes 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 a push-button window controller example. Example 9.1 gives the overview of the design objectives for this example and the state diagram describing the behavior to be modeled in VHDL.

B. J. LaMeres, *Quick Start Guide to VHDL*, https://doi.org/10.1007/978-3-030-04516-6_9

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

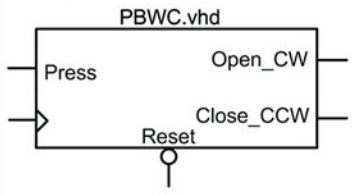
The window controller will send the appropriate control signals to a motor to open or close it whenever a button is pressed. The system must keep track whether the window is open or closed in order to send the correct signal, thus a state machine is needed. The block diagram and state diagram for this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



Example 9.1
Push-button window controller in VHDL: design description

Let's begin by defining the entity. 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 9.2 shows the VHDL entity definition for this example.

Example: Push-Button Window Controller in VHDL – Entity Definition




```
library IEEE;
use IEEE.std_logic_1164.all;

entity PBWC is
  port (Clock, Reset       : in  std_logic;
        Press              : in  std_logic;
        Open_CW, Close_CCW : out std_logic);
end entity;
```

Example 9.2
Push-button window controller in VHDL: entity definition

9.1.1 Modeling the States with User-Defined, Enumerated Data Types

Now we begin designing the finite state machine in VHDL using behavioral modeling constructs. The first step is to create a new user-defined, enumerated data type that can take on values that match the descriptive state names we've chosen in the state diagram (i.e., w_closed and w_open). This is accomplished by declaring a new type before the begin statement in the architecture with the keyword *type*. For this example, we will create a new type called *State_Type* and explicitly enumerate the values that it can take on. This type can now be used in future signal declarations. We then create two new signals called *current_state* and *next_state* of type State_Type. These two signals will be used throughout the VHDL

model in order to provide a high-level, readable description of the FSM behavior. The following syntax shows how to declare the new type and declare the current_state and next_state signals.

```
type  State_Type is (w_closed, w_open);
signal current_state, next_state : State_Type;
```

9.1.2 The State Memory Process

Now we model the state memory of the FSM using a process. This process 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 process 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 process will simply update current_state with next_state on every rising edge of the clock. The process model is very similar to the model of a D-flip-flop. This is as expected since this process 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:

```
STATE_MEMORY : process (Clock, Reset)
  begin
    if (Reset = '0') then
        current_state <= w_closed;
    elsif (Clock'event and Clock='1') then
        current_state <= next_state;
    end if;
end process;
```

9.1.3 The Next State Logic Process

Now we model the next state logic of the FSM using a second process. 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 process in addition to any inputs to the system. For this example, the system has one other input called *Press*. This process 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/then 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 *when others* clause to ensure that the state machine has a path back to the reset state in the case of an unexpected fault.

```
NEXT_STATE_LOGIC : process (current_state, Press)
  begin
    case (current_state) is
       when w_closed => if (Press = '1') then
                           next_state <= w_open;
                        else
                           next_state <= w_closed;
                        end if;
       when w_open  => if (Press = '1') then
                           next_state <= w_closed;
                        else
                           next_state <= w_open;
                        end if;
       when others  => next_state <= w_closed;
    end case;
  end process;
```

9.1.4 The Output Logic Process

Now we model the output logic of the FSM using a third process. 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 process 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 *when others* clause to ensure that the state machine has explicit output behavior in the case of a fault.

```
OUTPUT_LOGIC : process (current_state, Press)
  begin
    case (current_state) is
      when w_closed => if (Press = '1') then
                           Open_CW <= '1'; Close_CCW <= '0';
                       else
                           Open_CW <= '0'; Close_CCW <= '0';
                       end if;

      when w_open   => if (Press = '1') then
                           Open_CW <= '0'; Close_CCW <= '1';
                       else
                           Open_CW <= '0'; Close_CCW <= '0';
                       end if;

      when others   => Open_CW <= '0'; Close_CCW <= '0';
    end case;
end process;
```

Putting this all together in the VHDL architecture yields a functional model for the FSM that can be simulated and synthesized. Once again, it is important to keep in mind that since we did not explicitly assign the state codes, the synthesizer will automatically assign the codes based on the most efficient use of the target technology. Example 9.3 shows the entire architecture for this example.

Example: Push-Button Window Controller in VHDL – Architecture

```
architecture PBWC_arch of PBWC is

 type   State_Type is (w_closed, w_open);
 signal current_state, next_state : State_Type;

begin
-------------------------------------------------------
  STATE_MEMORY : process (Clock, Reset)
    begin
      if (Reset = '0') then
        current_state <= w_closed;
      elsif (Clock'event and Clock='1') then
        current_state <= next_state;
      end if;
  end process;
-------------------------------------------------------
  NEXT_STATE_LOGIC : process (current_state, Press)
    begin
      case (current_state) is
        when w_closed => if (Press = '1') then
                            next_state <= w_open;
                         else
                            next_state <= w_closed;
                         end if;
        when w_open   => if (Press = '1') then
                            next_state <= w_closed;
                         else
                            next_state <= w_open;
                         end if;
        when others   => next_state <= w_closed;
      end case;
  end process;
-------------------------------------------------------
  OUTPUT_LOGIC : process (current_state, Press)
    begin
      case (current_state) is
        when w_closed => if (Press = '1') then
                            Open_CW <= '1'; Close_CCW <= '0';
                         else
                            Open_CW <= '0'; Close_CCW <= '0';
                         end if;
        when w_open   => if (Press = '1') then
                            Open_CW <= '0'; Close_CCW <= '1';
                         else
                            Open_CW <= '0'; Close_CCW <= '0';
                         end if;
        when others   => Open_CW <= '0'; Close_CCW <= '0';
      end case;
  end process;

end architecture;
```

Declaration of user defined type for the signals current_state and next_state.

The first process is used to model the state memory.

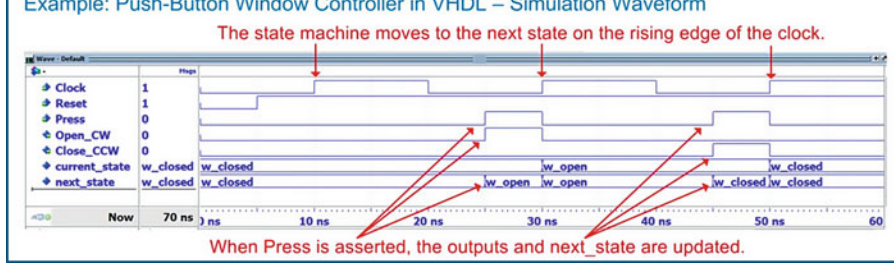
The second process is used to model the next state logic.

The third process is used to model the output logic.

Example 9.3
Push-button window controller in VHDL: architecture

Example 9.4 shows the simulation waveform for this state machine. This functional simulation was performed using ModelSim-Altera Starter Edition 10.1d.

Example: Push-Button Window Controller in VHDL – Simulation Waveform

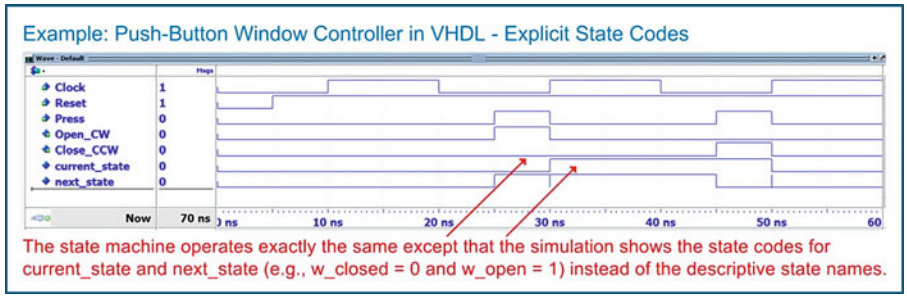


Example 9.4
Push-button window controller in VHDL: simulation waveform

9.1.5 Explicitly Defining State Codes with Subtypes

In the prior example, we did not have control over the state variable encoding. While the previous example is the most common way to model FSMs, there are situations where we would like to assign the state variable codes manually. This is accomplished using a *subtype* and constants. A subtype is simply a constrained type, meaning that it defines a subset of values that an existing type can take on. For example, we could create a subtype to constrain the std_logic data type to only allow values of 0 and 1 and *not* the values of U, X, Z, W, L, H, and -. This would not be considered a new type since it is simply a constraint put upon the existing std_logic type. A subtype defines the constraint and has a unique name that can be used to declare other signals. To use this approach for manually encoding the states of a FSM, we first declare a new subtype called *State_Type* that is simply a version of the existing type std_logic. We then create constants to represent the descriptive state names in the state diagram. These constants are given the type State_Type and a specific value. The value given is the state code we wish to assign to the particular state name. Finally, the current_state and next_state signals are declared of type State_Type. In this way, we can use the same VHDL processes as in the previous example that use the descriptive state names from the state diagram. The following is the VHDL syntax for manually assigning the state codes using subtypes. This syntax would replace the State_Type declaration in the previous example. Example 9.5 shows the resulting simulation waveforms.

```
subtype State_Type is std_logic;
constant w_open  : State_Type := '0';
constant w_closed : State_Type := '1';
signal  current_state, next_state : State_Type;
```



Example 9.5
Push-button window controller in VHDL: explicit state codes

Concept Check

CC9.1 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

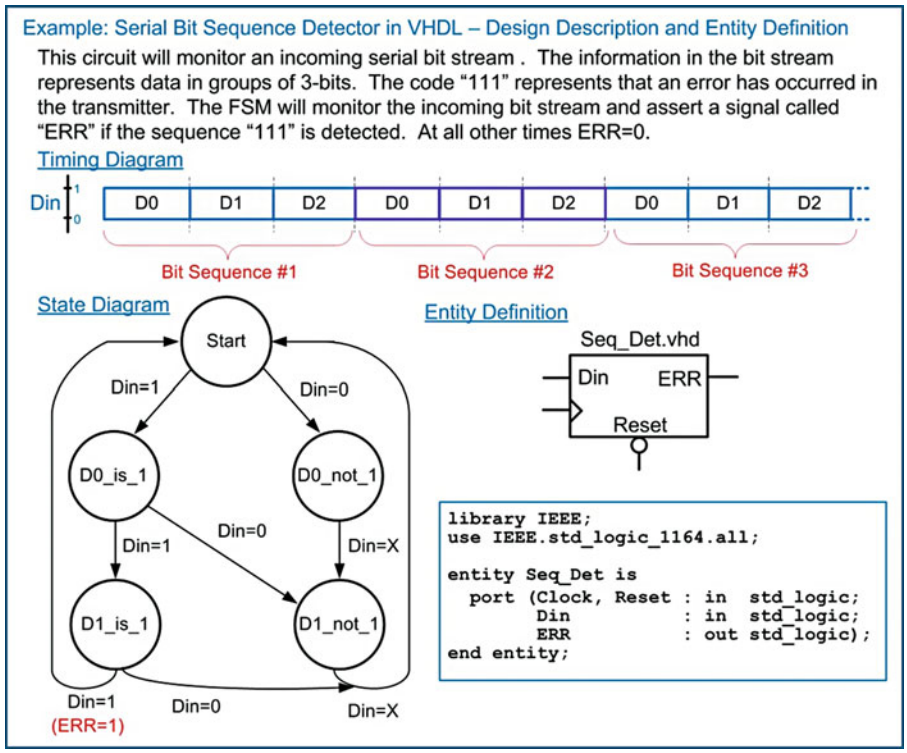
(C) So that the state memory process can be re-used in other FSMs

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

9.2 FSM Design Examples

9.2.1 Serial Bit Sequence Detector in VHDL

Let's look at the design of the serial bit sequence detector finite state machine from Chap. 7 using the behavioral modeling constructs of VHDL. Example 9.6 shows the design description and entity definition for this state machine.



Example 9.6
Serial bit sequence detector in VHDL: design description and entity definition

Example 9.7 shows the architecture for the serial bit sequence detector. In this example, a user-defined type is created to model the descriptive state names in the state diagram.

Example: Serial Bit Sequence Detector in VHDL – Architecture

```
architecture Seq_Det_arch of Seq_Det is

 type   State_Type is (Start, D0_is_1, D1_is_1, D0_not_1, D1_not_1);
 signal current_state, next_state : State_Type;

 begin

-------------------------------------------------------------
  STATE_MEMORY : process (Clock, Reset)
    begin
      if (Reset = '0') then
        current_state <= Start;
      elsif (Clock'event and Clock='1') then
        current_state <= next_state;
      end if;
  end process;
-------------------------------------------------------------
  NEXT_STATE_LOGIC : process (current_state, Din)
    begin
      case (current_state) is
        when Start    => if (Din = '1') then
                           next_state <= D0_is_1;
                         else
                           next_state <= D0_not_1;
                         end if;
        when D0_is_1  => if (Din = '1') then
                           next_state <= D1_is_1;
                         else
                           next_state <= D1_not_1;
                         end if;
        when D1_is_1  => next_state <= Start;
        when D0_not_1 => next_state <= D1_not_1;
        when D1_not_1 => next_state <= Start;
        when others   => next_state <= Start;
      end case;
  end process;
-------------------------------------------------------------
  OUTPUT_LOGIC : process (current_state, Din)
    begin
      case (current_state) is
        when D1_is_1  => if (Din = '1') then
                           ERR <= '1';
                         else
                           ERR <= '0';
                         end if;
        when others   => ERR <= '0';
      end case;
  end process;

end architecture;
```

Declaration of user defined type for the signals current_state and next_state.

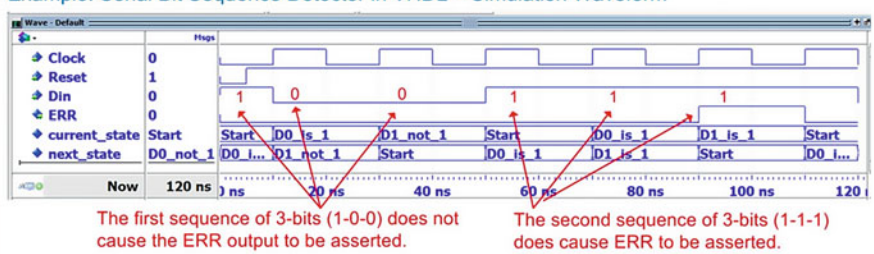
Note that in this example there are states decisions that don't require if/then statements.

This is a Meally machine so both the current state and the system inputs are present in the sensitivity list.

Example 9.7
Serial bit sequence detector in VHDL: architecture

Example 9.8 shows the functional simulation waveform for this design.

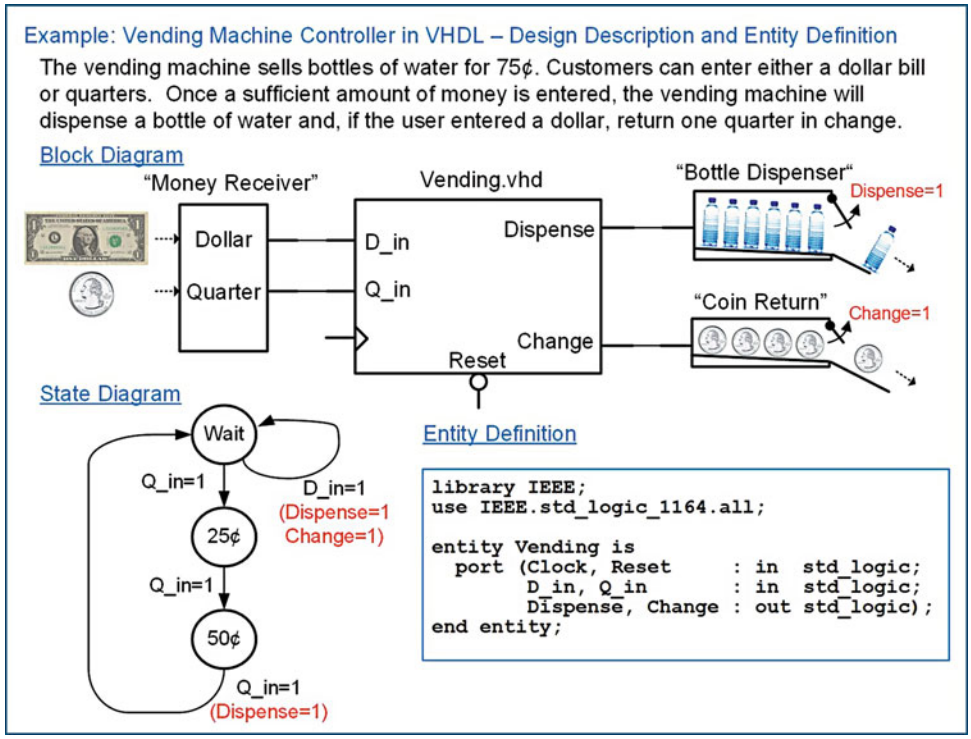
Example: Serial Bit Sequence Detector in VHDL – Simulation Waveform



Example 9.8
Serial bit sequence detector in VHDL: simulation waveform

9.2.2 Vending Machine Controller in VHDL

Let's now look at the design of the vending machine controller from Chap. 7 using the behavioral modeling constructs of VHDL. Example 9.9 shows the design description and entity definition.



Example 9.9
Vending machine controller in VHDL: design description and entity definition

Example 9.10 shows the VHDL architecture for the vending machine controller. In this model, the descriptive state names Wait, 25¢, and 50¢ cannot be used directly. This is because Wait is a VHDL keyword and 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 VHDL names (i.e., sWait, s25, s50).

Example: Vending Machine Controller in VHDL – Architecture

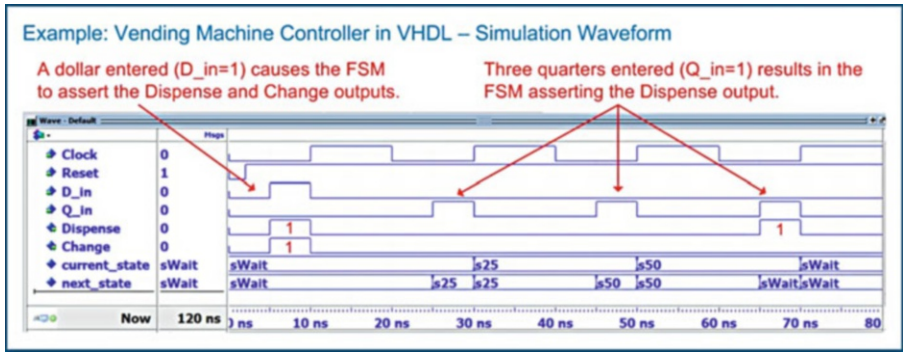
```
architecture Vending_arch of Vending is

 type   State_Type is (sWait, s25, s50);
 signal current_state, next_state : State_Type;

 begin

-------------------------------------------------------
  STATE_MEMORY : process (Clock, Reset)
    begin
      if (Reset = '0') then
        current_state <= sWait;
      elsif (Clock'event and Clock='1') then
        current_state <= next_state;
      end if;
  end process;

-------------------------------------------------------
  NEXT_STATE_LOGIC : process (current_state, D_in, Q_in)
    begin
      case (current_state) is
        when sWait  => if (Q_in = '1') then
                         next_state <= s25;
                       else
                         next_state <= sWait;
                       end if;
        when s25    => if (Q_in = '1') then
                         next_state <= s50;
                       else
                         next_state <= s25;
                       end if;
        when s50    => if (Q_in = '1') then
                         next_state <= sWait;
                       else
                         next_state <= s50;
                       end if;
        when others => next_state <= sWait;
      end case;
  end process;

-------------------------------------------------------
  OUTPUT_LOGIC : process (current_state, D_in, Q_in)
    begin
      case (current_state) is
        when sWait  => if (D_in = '1') then
                          Dispense <= '1'; Change <='1';
                       else
                          Dispense <= '0'; Change <='0';
                       end if;
        when s25    => Dispense <= '0'; Change <='0';
        when s50    => if (Q_in = '1') then
                          Dispense <= '1'; Change <='0';
                       else
                          Dispense <= '0'; Change <='0';
                       end if;
        when others => Dispense <= '0'; Change <='0';
      end case;
  end process;

end architecture;
```

Note that an "s" is added to the beginning of the state names since "Wait" is a VHDL keyword and names cannot start with a number.

This is a Meally machine so both the current state and the system inputs are present in the sensitivity list.

Example 9.10
Vending machine controller in VHDL: architecture

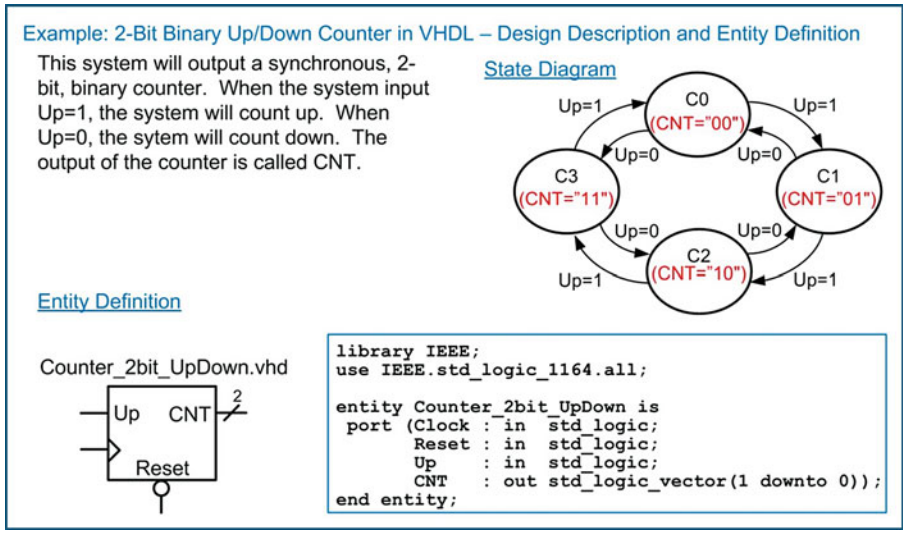
Example 9.11 shows the resulting simulation waveform for this design.



Example 9.11
Vending machine controller in VHDL: simulation waveform

9.2.3 2-Bit, Binary Up/Down Counter in VHDL

Let's now look at how a simple counter can be implemented using the three-process behavioral modeling approach in VHDL. Example 9.12 shows the design description and entity definition for the 2-bit, binary up/down counter FSM from Chap. 7.



Example 9.12
2-Bit binary up/down counter in VHDL: design description and entity definition

Example 9.13 shows the architecture for the 2-bit up/down counter using the three-process modeling approach. Since a counter's outputs only depend on the current state, counters are Moore machines. This simplifies the output logic process since it only needs to contain the current state in its sensitivity list.

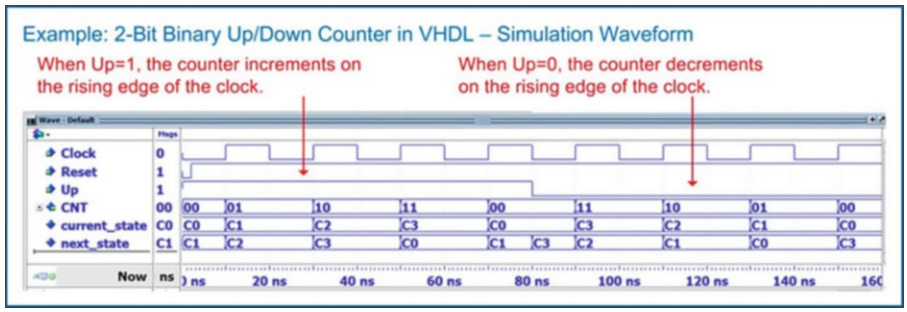
Example: 2-Bit Binary Up/Down Counter in VHDL – Architecture (Three Process Model)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Counter_2bit_UpDown is
 port (Clock, Reset : in  std_logic;
       Up           : in  std_logic;
       CNT          : out std_logic_vector(1 downto 0));
end entity;

architecture Counter_2bit_UpDown_arch of Counter_2bit_UpDown is
 type   State_Type is (C0, C1, C2, C3);
 signal current_state, next_state : State_Type;

 begin
-------------------------------------------------------
  STATE_MEMORY : process (Clock, Reset)
    begin
      if (Reset = '0') then
        current_state <= C0;
      elsif (Clock'event and Clock='1') then
        current_state <= next_state;
      end if;
  end process;
-------------------------------------------------------
  NEXT_STATE_LOGIC : process (current_state, Up)
    begin
      case (current_state) is
        when C0     => if (Up = '1') then
                         next_state <= C1;
                       else
                         next_state <= C3;
                       end if;
        when C1     => if (Up = '1') then
                         next_state <= C2;
                       else
                         next_state <= C0;
                       end if;
        when C2     => if (Up = '1') then
                         next_state <= C3;
                       else
                         next_state <= C1;
                       end if;
        when C3     => if (Up = '1') then
                         next_state <= C0;
                       else
                         next_state <= C2;
                       end if;
        when others => next_state <= C0;
      end case;
  end process;
-------------------------------------------------------
  OUTPUT_LOGIC : process (current_state)
    begin
      case (current_state) is
        when C0     => CNT <= "00";
        when C1     => CNT <= "01";
        when C2     => CNT <= "10";
        when C3     => CNT <= "11";
        when others => CNT <= "00";
      end case;
  end process;

end architecture;
```

A counter is a Moore machine so the output only depends on the current state.

Example 9.13
2-Bit binary up/down counter in VHDL: architecture (three process model)

Example 9.14 shows the resulting simulation waveform for this counter finite state machine.



Example 9.14
2-Bit binary up/down counter in VHDL: simulation waveform

Concept Check

CC9.2 The state memory process 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 processes that describe the behavior of the next state logic, the state memory, and the output logic. Separate processes are used because each of the three functions in a FSM are dependent on different input signals.
- In VHDL, descriptive state names can be created for a FSM with a user-defined, enumerated data type. The new type is first declared, and each of the descriptive state names is provided that the new data type can take on. Two signals are then created called *current_state* and *next_state* using the new data type. These two signals can then be assigned the descriptive state names of the FSM directly. This approach allows the synthesizer to assign the state codes arbitrarily.
- A subtype can be used when defining the state names if it is desired to explicitly define the state codes.

Exercise Problems

Section 9.1: The FSM Design Process

9.1.1 What is the advantage of using *user-defined, enumerated data types* for the states when modeling a finite state machine?

9.1.2 What is the advantage of using *subtypes* for the states when modeling a finite state machine?

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

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

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

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

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

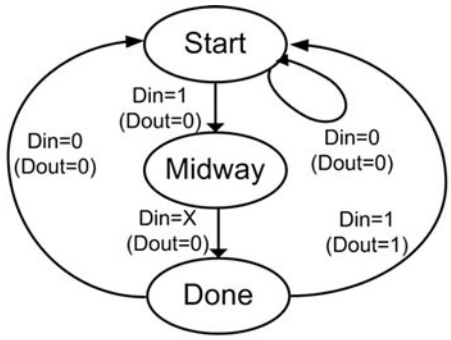
9.1.8 When using the three-process behavioral modeling approach for finite state machines, what inputs are listed in the sensitivity list of the output logic process?

9.1.9 When using the three-process behavioral modeling approach for finite state machines, how can the signals listed in the sensitivity list of the output logic process immediately tell whether the FSM is a Mealy or a Moore machine?

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

Section 9.2: FSM Design Examples

9.2.1 Design a VHDL behavioral model to implement the finite state machine described by the state diagram in Fig. 9.1. Use the entity definition provided in this figure for your design. Use the three-process approach to modeling FSMs described in this chapter for your design. Model the states in this machine with a user-defined enumerated type.



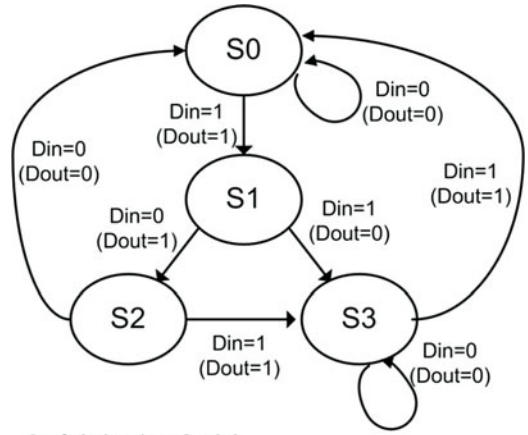
fsm1_behavioral.vhd

```
entity fsm1_behavioral is
    port    (Clock, Reset : in   std_logic;
             Din          : in   std_logic;
             Dout         : out  std_logic);
end entity;
```

Fig. 9.1
FSM 1 state diagram and entity

9.2.2 Design a VHDL behavioral model to implement the finite state machine described by the state diagram in Fig. 9.1. Use the entity definition provided in this figure for your design. Use the three-process approach to modeling FSMs described in this chapter for your design. Explicitly assign binary state codes using VHDL subtypes. Use the following state codes: Start = "00," Midway = "01," and Done = "10.",

9.2.3 Design a VHDL behavioral model to implement the finite state machine described by the state diagram in Fig. 9.2. Use the entity definition provided in this figure for your design. Use the three-process approach to modeling FSMs described in this chapter for your design. Model the states in this machine with a user-defined enumerated type.



fsm2_behavioral.vhd

```
entity fsm2_behavioral is
    port    (Clock, Reset : in   std_logic;
             Din          : in   std_logic;
             Dout         : out  std_logic);
end entity;
```

Fig. 9.2
FSM 2 state diagram and entity

9.2.4 Design a VHDL behavioral model to implement the finite state machine described by the state diagram in Fig. 9.2. Use the entity definition provided in this figure for your design. Use the three-process approach to modeling FSMs described in this chapter for your design. Assign one-hot state codes using VHDL subtypes. Use the following state codes: S0 = "0001," S1 = "0010," S2 = "0100," and S3 = "1000."

9.2.5 Design a VHDL behavioral model for a 4-bit serial bit sequence detector similar to Example 9.6. Use the entity definition provided in Fig. 9.3. Use the three-process 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." For all other input sequences, the output is not asserted. Model the states in your machine with a user-defined enumerated type.

Seq_Det_behavioral.vhd

```
entity Seq_Det_behavioral is
  port (Clock, Reset : in  std_logic;
        DIN          : in  std_logic;
        FOUND        : out std_logic);
end entity;
```

Fig. 9.3
Sequence detector entity

9.2.6 Design a VHDL behavioral model for a 20-cent vending machine controller similar to Example 9.9. Use the entity definition provided in Fig. 9.4. Use the three-process 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 a user-defined enumerated type.

Vending_behavioral.vhd

```
entity Vending_behavioral is
  port (Clock, Reset      : in  std_logic;
        Nin, Din          : in  std_logic;
        Dispense, Change  : out std_logic);
end entity;
```

Fig. 9.4
Vending machine entity

9.2.7 Design a VHDL behavioral model for a finite state machine for a traffic light controller at the intersection of a busy highway and a seldom used side road. Use the entity definition provided in Fig. 9.5. 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 a user-defined enumerated type.

tlc_behavioral.vhd

```
entity tlc_behavioral is
  port (Clock, Reset    : in  std_logic;
        CAR, TIMEOUT    : in  std_logic;
        GRN, YLW, RED   : out std_logic);
end entity;
```

Fig. 9.5
Traffic light controller entity