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:

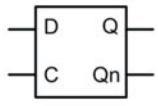
9.1 Design a VHDL behavioral model for a sequential logic storage device.
9.2 Describe the process for creating a VHDL behavioral model for a finite state machine.
9.3 Design a VHDL behavioral model for a finite state machine.
9.4 Design a VHDL behavioral model for a counter.
9.5 Design a VHDL register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

9.1 Modeling Sequential Storage Devices in VHDL

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 process. Since we want to create a synthesizable model, we use a sensitivity list to trigger the process instead of wait statements. 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 continuously assigned the value of D so any change on D needs to trigger the process. The use of an if/then 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 end the if/then statement to complete the model. Example 9.1 shows the behavioral model for a D-Latch.

Example: Behavioral Model of a D-Latch in VHDL



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

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

entity Dlatch is
    port    (C, D     : in   std_logic;
             Q, Qn    : out  std_logic);
end entity;

architecture Dlatch_arch of Dlatch is

  begin

     D_LATCH : process (C, D)
       begin
          if (C = '1') then
              Q <= D; Qn <= not D;
          end if;
     end process;

end architecture;
```

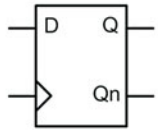
Example 9.1
Behavioral model of a D-Latch in VHDL

B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*,
https://doi.org/10.1007/978-3-030-12489-2_9

9.1.2 D-Flip-Flop

The rising edge behavior of a D-Flip-Flop is modeled using a (Clock'event and Clock = '1') Boolean condition within a process. The (rising_edge(Clock)) function can also be used for type std_logic. 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 VHDL



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

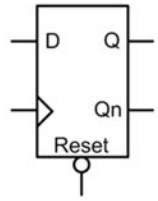
```
library IEEE;
use IEEE.std_logic_1164.all;

entity Dflipflop is
    port    (Clock          : in   std_logic;
             D              : in   std_logic;
             Q, Qn          : out  std_logic);
end entity;

architecture Dflipflop_arch of Dflipflop is

  begin

    D_FLIP_FLOP : process (Clock)
      begin
          if (Clock'event and Clock='1') then
             Q <= D;    Qn <= not D;
          end if;
    end process;

end architecture;
```

Example 9.2
Behavioral model of a D-Flip-Flop in VHDL

9.1.3 D-Flip-Flop with Asynchronous Reset

D-Flip-Flops typically have a reset line in order to initialize their outputs to a known state (e.g., Q = 0, Qn = 1). Resets are asynchronous, meaning that whenever they are asserted, assignments to the outputs take place immediately. If a reset was *synchronous*, the output assignments would only take place 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 placed in the sensitivity list. This allows both the clock and the reset inputs to trigger the process. Within the process, an if/then/elsif statement is used to determine whether the reset has been asserted or a rising edge of the clock has occurred. The if/then/elsif statement first checks whether the reset input has been asserted. If it has, it makes the appropriate assignments to the outputs (Q = 0, Qn = 1). If the reset has not been asserted, the *elsif* clause checks whether a rising edge of the clock has occurred using the (Clock'event and Clock = '1') Boolean condition. If it has, the outputs are updated accordingly (Q <= D, Qn <= not D). A final else statement is not included so that assignments to the outputs are not made under any other condition. This models 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 VHDL



$\overline{R}$	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

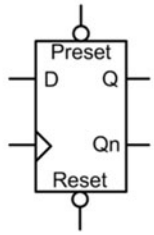
```
library IEEE;
use IEEE.std_logic_1164.all;

entity Dflipflop is
    port    (Clock          : in   std_logic;
             Reset          : in   std_logic;
             D              : in   std_logic;
             Q, Qn          : out  std_logic);
end entity;

architecture Dflipflop_arch of Dflipflop is

  begin

     D_FLIP_FLOP : process (Clock, Reset)
       begin
          if (Reset = '0') then
              Q <= '0'; Qn <= '1';
          elsif (Clock'event and Clock='1') then
              Q <= D;   Qn <= not D;
          end if;
     end process;

end architecture;
```

Example 9.3
Behavioral model of a D-Flip-Flop with asynchronous reset in VHDL

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 process whenever a transition occurs on either the clock, reset, or preset inputs. An if/then/elsif statement is 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 VHDL



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

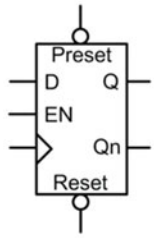
```
library IEEE;
use IEEE.std_logic_1164.all;

entity Dflipflop is
    port   (Clock           : in   std_logic;
            Reset, Preset   : in   std_logic;
            D               : in   std_logic;
            Q, Qn           : out  std_logic);
end entity;

architecture Dflipflop_arch of Dflipflop is

  begin

     D_FLIP_FLOP : process (Clock, Reset, Preset)
       begin
          if (Reset = '0') then
              Q <= '0'; Qn <= '1';
          elsif (Preset = '0') then
              Q <= '1'; Qn <= '0';
          elsif (Clock'event and Clock='1') then
              Q <= D;   Qn <= not D;
          end if;
     end process;

end architecture;
```

Example 9.4
Behavioral model of a D-Flip-Flop with asynchronous reset and preset in VHDL

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 process. Since action is only taken when there is a rising edge of the clock, a nested if/then statement is included beneath the *elsif (Clock'event and Clock = '1')* clause. 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 VHDL



$\overline{R}$	$\overline{P}$	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

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

entity Dflipflop is
    port   (Clock            : in   std_logic;
            Reset, Preset    : in   std_logic;
            D, EN            : in   std_logic;
            Q, Qn            : out  std_logic);
end entity;

architecture Dflipflop_arch of Dflipflop is

  begin

     D_FLIP_FLOP : process (Clock, Reset, Preset)
       begin
          if (Reset = '0') then
              Q <= '0'; Qn <= '1';
          elsif (Preset = '0') then
              Q <= '1'; Qn <= '0';
          elsif (Clock'event and Clock='1') then
              if (EN = '1') then
                 Q <= D;   Qn <= not D;
              end if;
          end if;
     end process;

end architecture;
```

A nested if/then statement is used to model the synchronous enable.

Example 9.5
Behavioral model of a D-Flip-Flop with synchronous enable in VHDL

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.

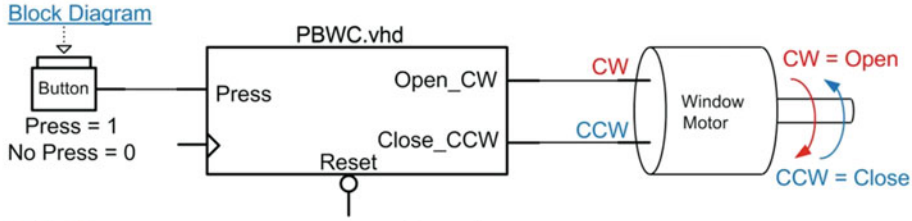
9.2 Modeling Finite State Machines in VHDL

Finite state machines can be easily modeled using the behavioral constructs from Chap. 8. 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 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 VHDL.

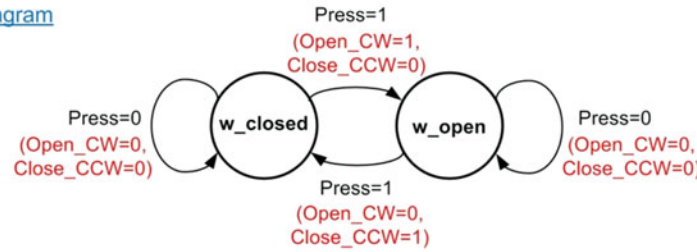
Example: Push-Button Window Controller in VHDL – Design Description

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.

Block Diagram

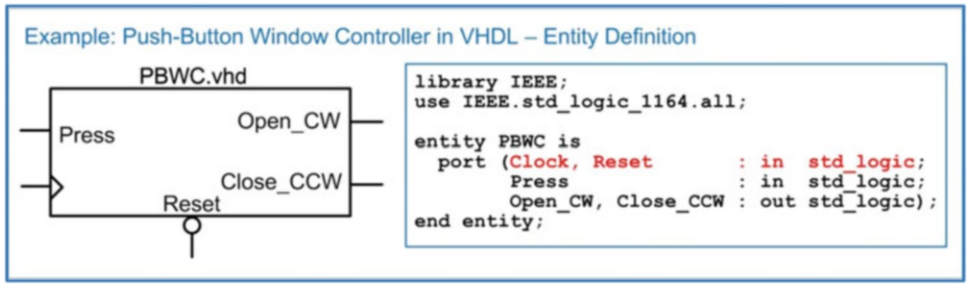


State Diagram



Example 9.6
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 9.7 shows the VHDL entity definition for this example.



Example 9.7
Push-button window controller in VHDL – entity definition

9.2.1 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;
```

9.2.2 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;
```

9.2.3 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;
```

9.2.4 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 9.8 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.

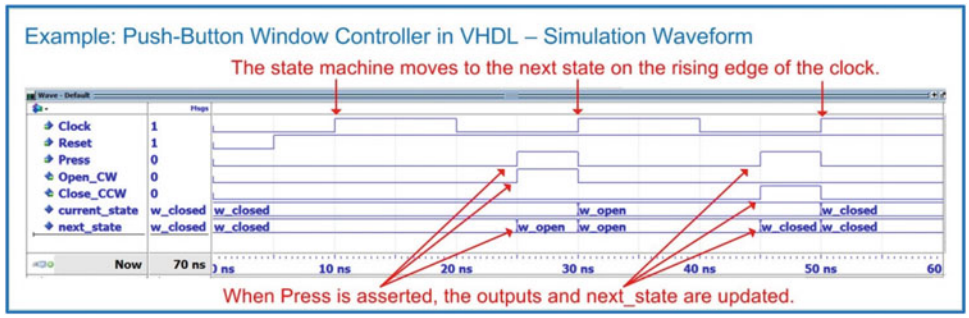
The first process is used to model the state memory.

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

The third process is used to model the output logic.

Example 9.8
Push-button window controller in VHDL – architecture

Example 9.9 shows the simulation waveform for this state machine. This functional simulation was performed using ModelSim-Altera Starter Edition 10.1d.

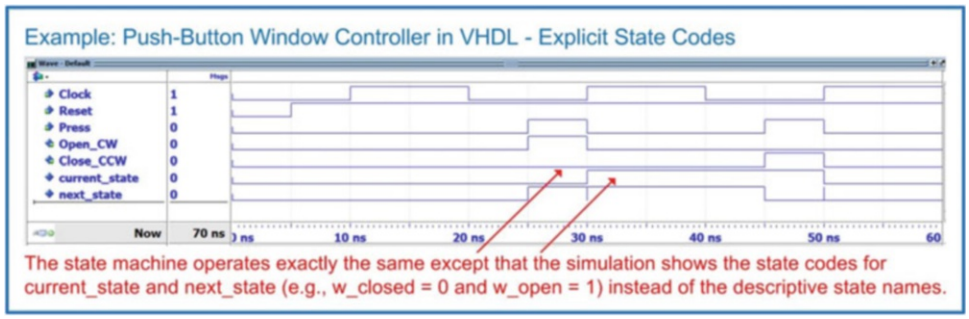


Example 9.9
Push-button window controller in VHDL – simulation waveform

9.2.5 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 9.10 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 9.10
Push-button window controller in VHDL – explicit 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 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.

9.3 FSM Design Examples in VHDL

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

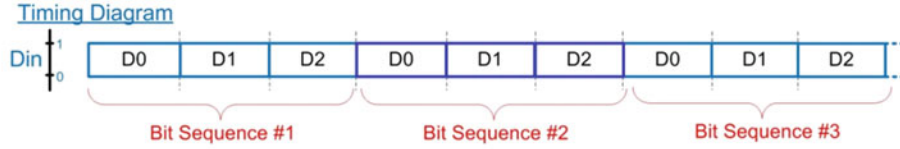
9.3.1 Serial Bit Sequence Detector in VHDL

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 9.11 shows the design description and entity definition for this state machine.

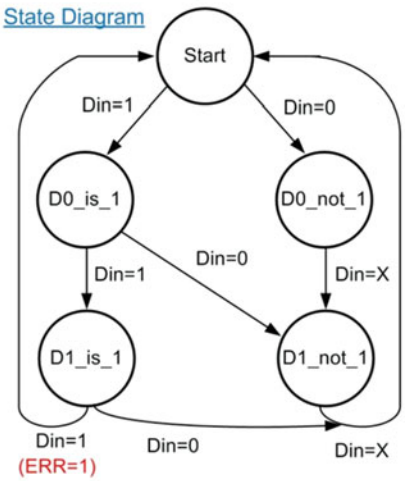
Example: Serial Bit Sequence Detector in VHDL – Design Description and Entity Definition

This circuit will monitor an incoming serial bit stream . The information in the bit stream represents data in groups of 3-bits. The code "111" represents that an error has occurred in the transmitter. The FSM will monitor the incoming bit stream and assert a signal called "ERR" if the sequence "111" is detected. At all other times ERR=0.

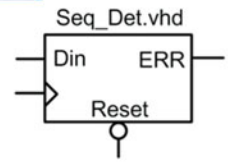
Timing Diagram



State Diagram



Entity Definition




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

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

Example 9.11
Serial bit sequence detector in VHDL – design description and entity definition

Example 9.12 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;
```

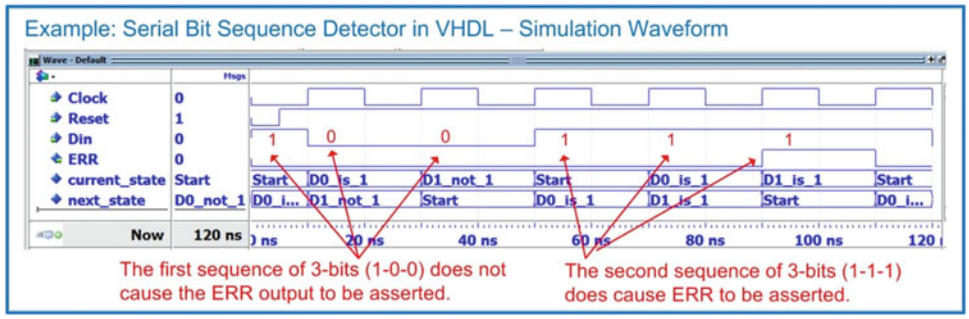
Declaration of user defined type for the signals current_state and next_state.

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 9.12
Serial bit sequence detector in VHDL – architecture

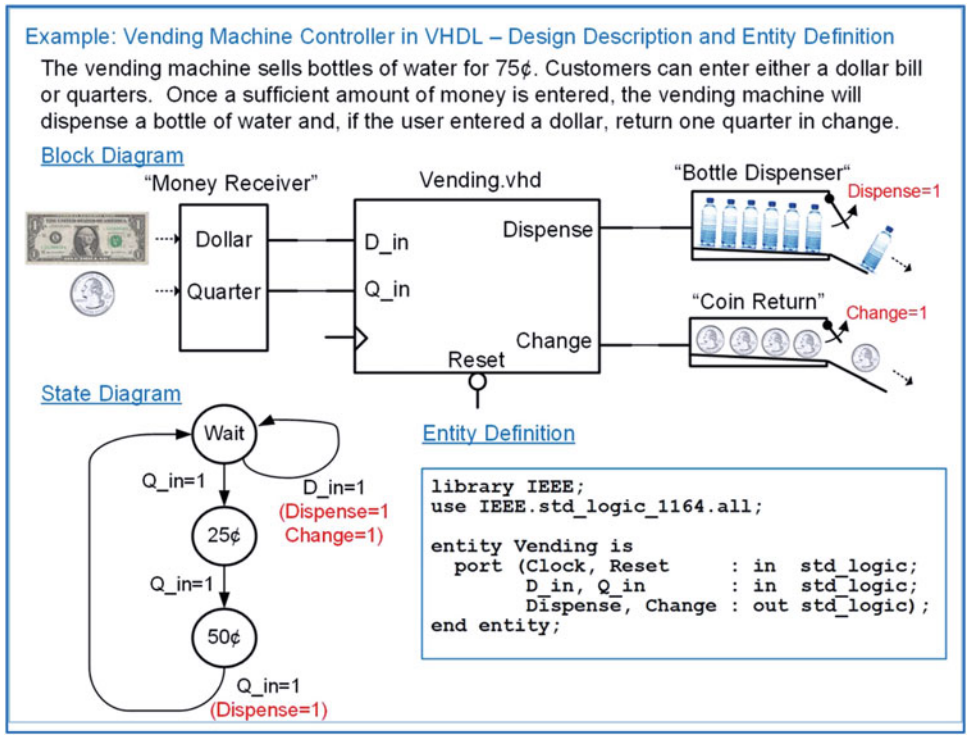
Example 9.13 shows the functional simulation waveform for this design.



Example 9.13
Serial bit sequence detector in VHDL – simulation waveform

9.3.2 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 9.14 shows the design description and entity definition.



Example 9.14
Vending machine controller in VHDL – design description and entity definition

Example 9.15 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

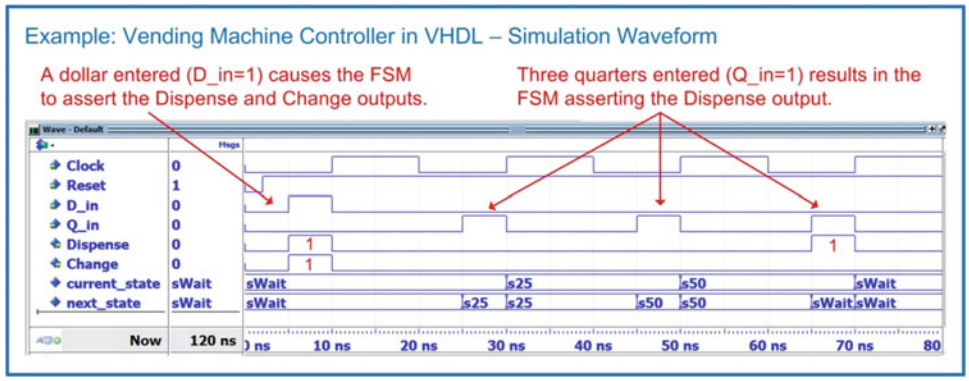
```
architecture Vending_arch of Vending is

 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 Mealy machine so both the current state and the system inputs are present in the sensitivity list.

Example 9.15
Vending machine controller in VHDL – architecture

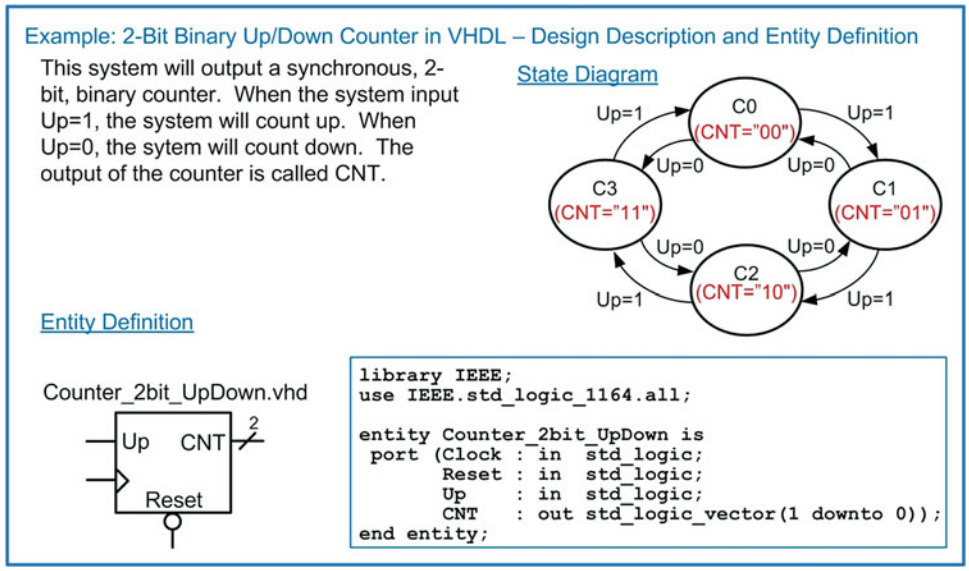
Example 9.16 shows the resulting simulation waveform for this design.



Example 9.16
Vending machine controller in VHDL – simulation waveform

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



Example 9.17
2-bit binary up/down counter in VHDL – design description and entity definition

Example 9.18 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.

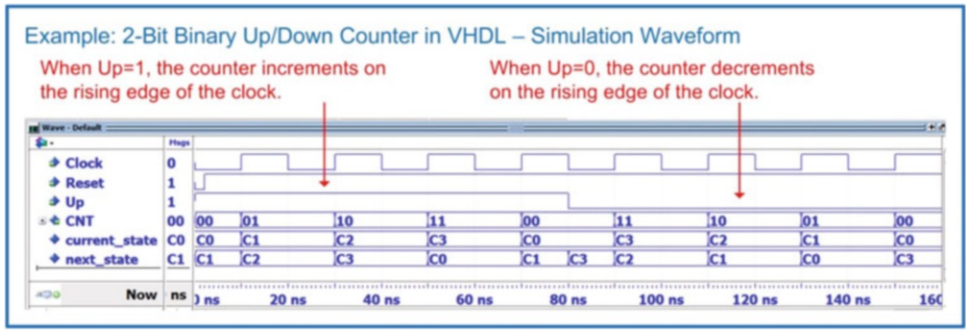
Example: 2-Bit Binary Up/Down Counter in VHDL – Architecture (Three Process Model)

```
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 9.18
2-bit binary up/down counter in VHDL – architecture (three-process model)

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



Example 9.19
2-bit binary up/down counter in VHDL – simulation waveform

CONCEPT CHECK

CC9.3 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.

9.4 Modeling Counters in VHDL

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

9.4.1 Counters in VHDL Using the Type UNSIGNED

Let's look at how we can model a 4-bit, binary up counter with an output called *CNT*. First, we want to model this counter using the "+" operator. Recall that the "+" operator is not defined in the std_logic_1164 package. We need to include the numeric_std package in order to add this capability. Within the numeric_std package, the "+" operator is only defined for types signed and unsigned (and not for std_logic_vector), so the output CNT will be declared as type unsigned. Next, we want to implement the counter using a signal assignment in the form *CNT* <= *CNT* + *1*; however, since CNT is an output port, it cannot be used as an argument (right hand side) in an operation. We will need to create an internal signal to implement the counter functionality (i.e., CNT_tmp). Since a signal does not contain directionality, it can be used as both the target and an argument of an operation. Outside of the counter process, a concurrent signal assignment is used to continuously assign CNT_tmp to CNT in order to drive the output of the system. This means that we need to create the internal signal CNT_tmp of type unsigned also to

support this assignment. Example 9.20 shows the VHDL 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 defined to only contain 4-bits.

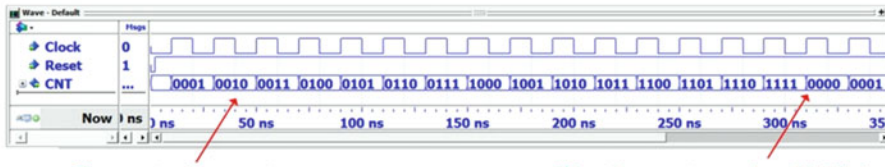
Example: 4-Bit Binary Up Counter in VHDL Using the Type UNSIGNED

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

entity Counter_4bit_Up is
 port (Clock, Reset : in  std_logic;
       CNT          : out unsigned(3 downto 0));
end entity;

architecture Counter_4bit_Up_arch of Counter_4bit_Up is

 signal CNT_tmp : unsigned(3 downto 0);

 begin

  COUNTER : process (Clock, Reset)
    begin
      if (Reset = '0') then
        CNT_tmp <= "0000";
      elsif (Clock'event and Clock='1') then
        CNT_tmp <= CNT_tmp + 1;
      end if;
  end process;

  CNT <= CNT_tmp;

end architecture;
```

The numeric_std package is needed to include the "+" operator. This operator only works on types signed/unsigned, so we will define the output CNT as type unsigned.

An internal signal is needed to support assignments in the form C <= C+1; because a port cannot be used as an argument in a signal assignment.

A concurrent signal assignment is used to continually assign CNT_tmp to CNT.



The counter increments on each rising edge of clock.

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

Example 9.20
4-bit binary up counter in VHDL using the type UNSIGNED

9.4.2 Counters in VHDL Using the Type INTEGER

Another common technique to model counters with a single process is to use the type integer. The numeric_std package supports the "+" operator for type integer. It also contains a conversion between the types integer and unsigned/signed. This means a process can be created to model the counter functionality with integers, and then the result can be converted and assigned to the output of the system of type unsigned. One thing that must be considered when using integers is that they are defined as 32-bit, two's complement numbers. This means that if a counter is defined to use integers and the maximum range of the counter is not explicitly controlled, the counter will increment through the entire range of 32-bit values it can take on. There are a variety of ways to explicitly bound the size of an integer counter. The first is to use an if/then clause within the process to check for the upper limit desired in the counter. For example, if we wish to create a 4-bit binary counter, we will check if the integer counter has reached 15 each time through the process. If it has, we will reset it to zero. Synthesizers will recognize that the integer counter is never allowed to exceed 15 (or "1111" for an unsigned counter) and remove the

unused bits of the integer type during implementation (i.e., the remaining 28-bits). Example 9.21 shows the VHDL model and simulation waveform for this implementation of the 4-bit counter using integers.

Example: 4-Bit Binary Up Counter in VHDL Using the Type INTEGER

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

entity Counter_4bit_Up is
 port (Clock, Reset : in  std_logic;
       CNT          : out unsigned(3 downto 0));
end entity;

architecture Counter_4bit_Up_arch of Counter_4bit_Up is

 signal CNT_int : integer;

 begin

  COUNTER : process (Clock, Reset)
    begin
      if (Reset = '0') then
        CNT_int <= 0;
      elsif (Clock'event and Clock='1') then

        if (CNT_int = 15) then
          CNT_int <= 0;
        else
          CNT_int <= CNT_int + 1;
        end if;

      end if;
  end process;

  CNT <= to_unsigned(CNT_int, 4);

end architecture;
```

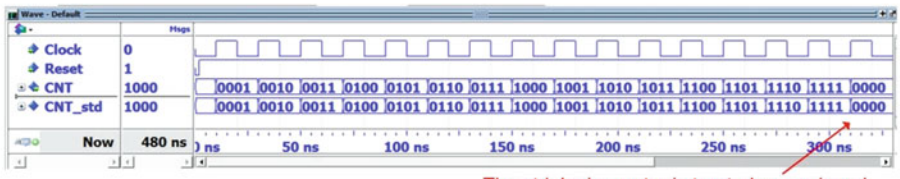
The numeric_std package contains the "+" operator for type integer and a conversion from type integer to type unsigned.

In this example, the output port is defined to be of type unsigned.

An internal signal of type integer is declared to model the counter functionality.

A nested if/then statement checks to see if the integer counter has reached its maximum value.

A concurrent assignment between the internal counter and the output port is made that contains the conversion between type integer and unsigned. The 4 in this function represents the number of unsigned bits to convert the integer into.



The std_logic_vector is treated as unsigned and will roll over once it gets to "1111".

Example 9.21
4-bit binary up counter in VHDL using the type INTEGER

9.4.3 Counters in VHDL Using the Type STD_LOGIC_VECTOR

It is often desired to have the ports of a system be defined of type std_logic/std_logic_vector for compatibility with other systems. One technique to accomplish this and also model the counter behavior internally using std_logic_vector is through inclusion of the numeric_std_unsigned package. This package allows the use of std_logic_vector when declaring the ports and signals within the design and treats them as unsigned when performing arithmetic and comparison functions. Example 9.22 shows the VHDL model and simulation waveform for this alternative implementation of the 4-bit counter.

Example: 4-Bit Binary Up Counter in VHDL Using the Type STD_LOGIC_VECTOR (1)

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

entity Counter_4bit_Up is
 port (Clock, Reset : in  std_logic;
       CNT          : out std_logic_vector(3 downto 0));
end entity;

architecture Counter_4bit_Up_arch of Counter_4bit_Up is

 signal CNT_std : std_logic_vector(3 downto 0);

 begin

  COUNTER : process (Clock, Reset)
    begin
      if (Reset = '0') then
        CNT_std <= "0000";
      elsif (Clock'event and Clock='1') then
        CNT_std <= CNT_std + 1;
      end if;
  end process;

  CNT <= CNT_std;

end architecture;
```

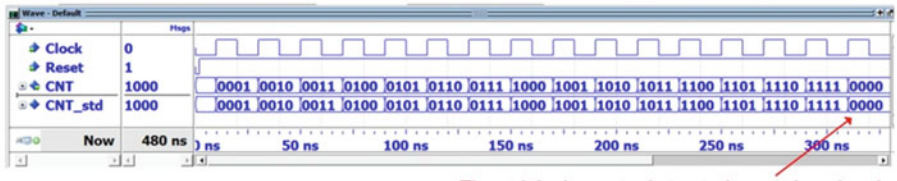
Including this package will treat all std_logic_vector types as unsigned numbers.

The output port is defined to be of type std_logic_vector.

The internal signal to model the counter behavior is declared as type std_logic_vector.

No boundary checking is needed since the 4-bit std_logic_vector will simply roll over.

No type conversion is needed since the internal signal and output port are of type std_logic_vector.



The std_logic_vector is treated as unsigned and will roll over once it gets to "1111".

Example 9.22
4-bit binary up counter in VHDL using the type STD_LOGIC_VECTOR (1)

If it is designed to have an output type of std_logic_vector and use an integer in modeling the behavior of the counter, and then a double conversion can be used. In the following example, the counter behavior is modeled using an integer type with range checking. A concurrent signal assignment is used at the end of the architecture in order to convert the integer to type std_logic_vector. This is accomplished by first converting the type integer to unsigned and then converting the type unsigned to std_logic_vector. Example 9.23 shows the VHDL model and simulation waveform for this alternative implementation of the 4-bit counter.

Example: 4-Bit Binary Up Counter in VHDL Using the Type STD_LOGIC_VECTOR (2)

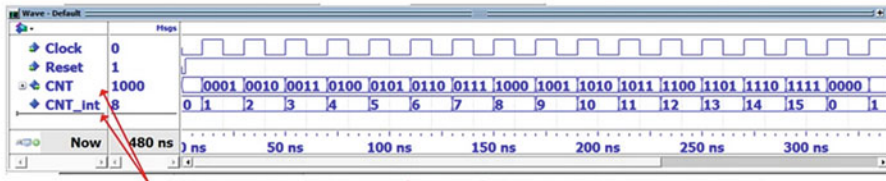
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Counter_4bit_Up is
 port (Clock, Reset : in  std_logic;
       CNT          : out std_logic_vector(3 downto 0));
end entity;

architecture Counter_4bit_Up_arch of Counter_4bit_Up is

 signal CNT_int : integer range 0 to 15;

 begin

  COUNTER : process (Clock, Reset)
    begin
      if (Reset = '0') then
        CNT_int <= 0;
      elsif (Clock'event and Clock='1') then

          if (CNT_int = 15) then
            CNT_int <= 0;
          else
            CNT_int <= CNT_int + 1;
          end if;

      end if;
  end process;

  CNT <= std_logic_vector( to_unsigned(CNT_int, 4) );

end architecture;
```

The output port is defined to be of type std_logic_vector.

The internal signal to model the counter behavior is declared as type integer. In this declaration, the integer range is also specified. This is unnecessary since the process will check for the maximum counter value but is commonly included for readability.

Range checking is required when using the type integer.

A double type conversion is used to change the integer to std_logic_vector.



In this example, the output CNT is of type std_logic_vector while the counter behavior is modeled using type integer.

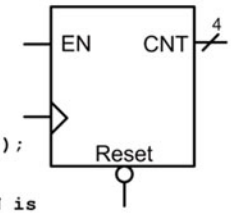
Example 9.23
4-bit binary up counter in VHDL using the type STD_LOGIC_VECTOR (2)

9.4.4 Counters with Enables in VHDL

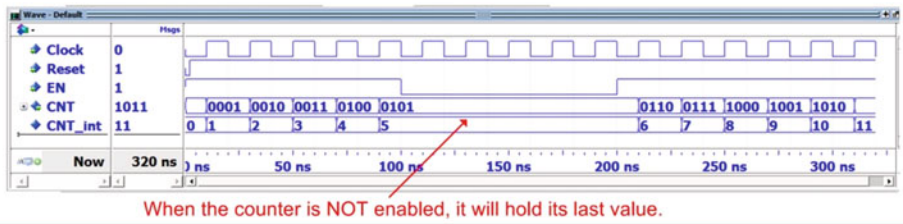
Including an enable in a counter is a common technique to prevent the counter from running continuously. When the enable is asserted, the counter will increment on the rising edge of the clock as usual. When the enable is de-asserted, the counter will simply hold its last value. Enable lines are synchronous, meaning that they are only evaluated on the rising edge of the clock. As such, they are modeled using a nested if/then statement within the if/then statement checking for a rising edge of the clock. Example 9.24 shows an example model for a 4-bit counter with enable.

Example: 4-Bit Binary Up Counter with Enable in VHDL

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

entity Counter_4bit_wEN is
 port (Clock, Reset : in  std_logic;
       EN           : in  std_logic;
       CNT          : out std_logic_vector(3 downto 0));
end entity;

architecture Counter_4bit_wEN_arch of Counter_4bit_wEN is

 signal CNT_int : integer range 0 to 15;

 begin

  COUNTER : process (Clock, Reset)
    begin
      if (Reset = '0') then
        CNT_int <= 0;
      elsif (Clock'event and Clock='1') then

        if (EN='1') then
            if (CNT_int = 15) then
              CNT_int <= 0;
            else
              CNT_int <= CNT_int + 1;
            end if;
        end if;

     end if;
  end process;

  CNT <= std_logic_vector( to_unsigned(CNT_int, 4) );

end architecture;
```



A nested if/then statement is used in order to check if the counter is enabled on each edge of the clock. If EN='1', the counter will increment as usual. If EN='0', the counter will simply hold its last value.



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

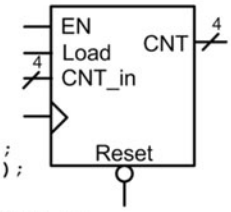
Example 9.24
4-bit binary up counter with enable in VHDL

9.4.5 Counters with Loads

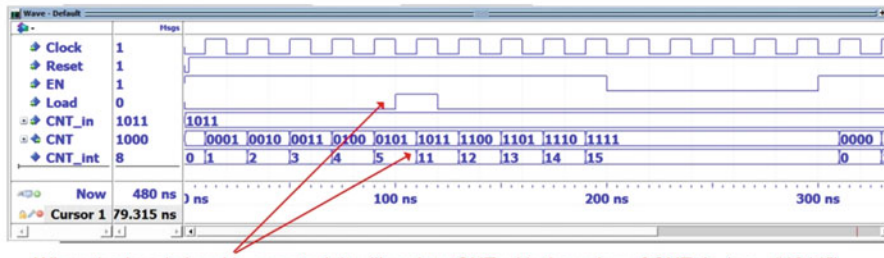
A counter with a *load* has the ability to set the counter to a specified value. The specified value is provided on an input port (i.e., CNT_in) with the same width as the counter output (CNT). A synchronous load input signal (i.e., Load) is used to indicate when the counter should set its value to the value present on CNT_in. Example 9.25 shows an example model for a 4-bit counter with load capability.

Example: 4-Bit Binary Up Counter with Load in VHDL

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

entity Counter_4bit_wLoad is
 port (Clock, Reset : in  std_logic;
       EN           : in  std_logic;
       Load         : in  std_logic;
       CNT_in       : in  std_logic_vector(3 downto 0);
       CNT          : out std_logic_vector(3 downto 0));
end entity;

architecture Counter_4bit_wLoad_arch of Counter_4bit_wLoad is

 signal CNT_int : integer range 0 to 15;

 begin

  COUNTER : process (Clock, Reset)
    begin
      if (Reset = '0') then
        CNT_int <= 0;
      elsif (Clock'event and Clock='1') then

        if (Load = '1') then
           CNT_int <= to_integer( unsigned(CNT_in) );
        else

            if (EN='1') then
                if (CNT_int = 15) then
                  CNT_int <= 0;
                else
                  CNT_int <= CNT_int + 1;
                end if;
            end if;

       end if;

     end if;
  end process;

  CNT <= std_logic_vector( to_unsigned(CNT_int, 4) );

end architecture;
```



A nested if/then statement is used to load CNT with CNT_in when the Load signal is asserted. Since CNT_int is of type integer and CNT_in is of type std_logic_vector, a type conversion is needed. Once again, two conversions are used since there is not a direct conversion between std_logic_vector and integer.



When the Load signal is asserted, it will update CNT with the value of CNT_in (e.g., "1011").

Example 9.25
4-bit binary up counter with load in VHDL

Concept Check

CC9.4 If a counter is modeled using only one process in VHDL, 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 process. Synthesizers will recognize the single process model as a FSM.

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

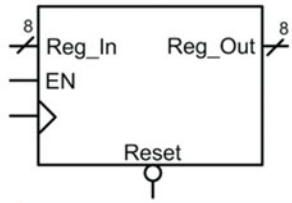
9.5 RTL Modeling

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 VHDL

The term *register* describes a circuit that operates in a similar manner as a D-Flip-Flop with the exception that the input and output data are vectors. This circuit is implemented with a set of D-Flip-Flops all connected to the same clock, reset, and enable inputs. 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-Flop components. Example 9.26 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 VHDL



$\overline{R}$	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

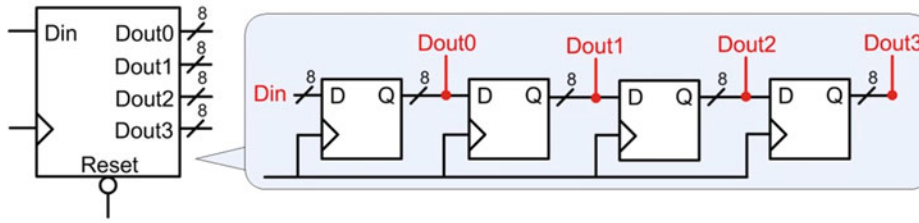
```
library IEEE;
use IEEE.std_logic_1164.all;

entity reg is
    port   (Clock          : in   std_logic;
            Reset          : in   std_logic;
            Reg_In         : in   std_logic_vector(7 downto 0);
            EN             : in   std_logic;
            Reg_Out        : out  std_logic_vector(7 downto 0));
end entity;

architecture reg_arch of reg is

 begin

  Reg_Proc : process (Clock, Reset)
    begin
      if (Reset = '0') then
         Reg_Out <= x"00";
      elsif (Clock'event and Clock='1') then
          if (EN = '1') then
            Reg_Out <= Reg_In;
          end if;
      end if;
   end process;

end architecture;
```

Example 9.26
RTL model of an 8-bit register in VHDL

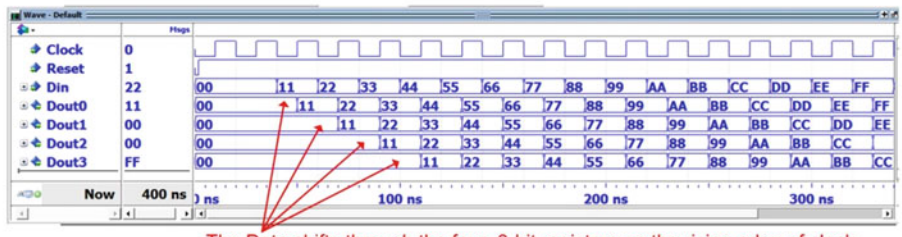
9.5.2 Shift Registers in VHDL

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 9.27 shows an RTL model for a 4-stage, 8-bit shift register. In the simulation waveform, the data is shown in hexadecimal format.

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




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

entity Shift_Register is
 port (Clock, Reset : in  std_logic;
       Din          : in  std_logic_vector(7 downto 0);
       Dout0, Dout1 : out std_logic_vector(7 downto 0);
       Dout2, Dout3 : out std_logic_vector(7 downto 0));
end entity;

architecture Shift_Register_arch of Shift_Register is

 signal D0, D1, D2, D3 : std_logic_vector(7 downto 0);

 begin

  SHIFT : process (Clock, Reset)
    begin
      if (Reset = '0') then
        D0 <= x"00"; D1 <= x"00"; D2 <= x"00"; D3 <= x"00";
      elsif (Clock'event and Clock='1') then
        D0 <= Din; D1 <= D0; D2 <= D1; D3 <= D2;
      end if;
  end process;

  Dout3 <= D3; Dout2 <= D2; Dout1 <= D1; Dout0 <= D0;

end architecture;
```

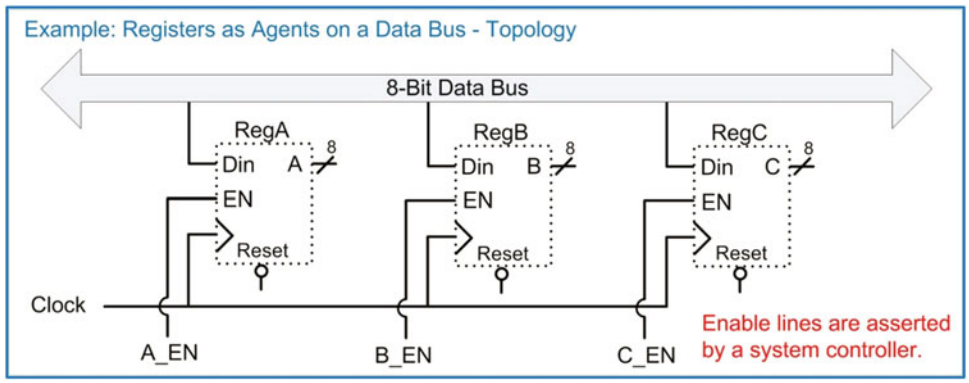


The Data shifts through the four, 8-bit registers on the rising edge of clock.

Example 9.27
RTL model of a 4-stage, 8-bit shift register in VHDL

9.5.3 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 9.28 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 9.28
Registers as agents on a data bus – system topology

This topology can be modeled using RTL abstraction by treating each register as a separate process. Example 9.29 shows how to describe this topology with an RTL model in VHDL. Notice that the three processes modeling the A, B, and C registers are nearly identical to each other with the exception of the signal names they use.

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

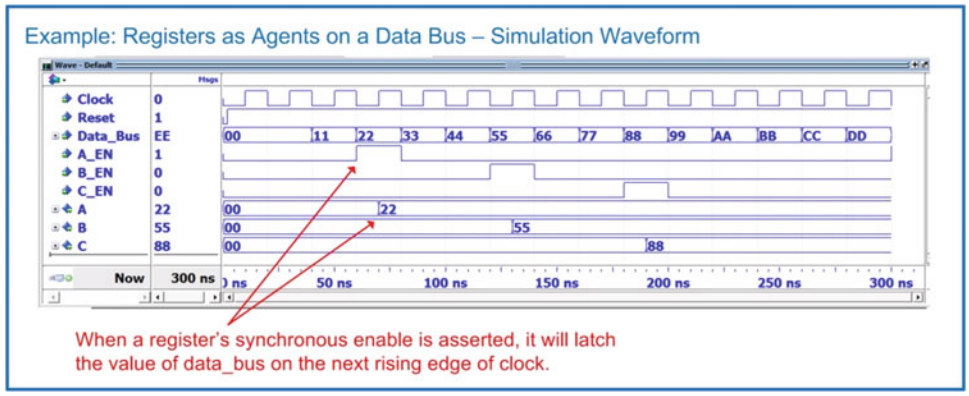
```
library IEEE;
use IEEE.std_logic_1164.all;

entity MultiDropBus is
 port (Clock, Reset      : in  std_logic;
       Data_Bus          : in  std_logic_vector(7 downto 0);
       A_EN, B_EN, C_EN  : in  std_logic;
       A, B, C           : out std_logic_vector(7 downto 0));
end entity;

architecture MultiDropBus_arch of MultiDropBus is

 begin
 ----------------------------------------
  A_REG : process (Clock, Reset)
    begin
      if (Reset = '0') then
        A <= x"00";
      elsif (Clock'event and Clock='1') then
          if (A_EN = '1') then
            A <= Data_Bus;
          end if;
      end if;
  end process;
  ----------------------------------------
  B_REG : process (Clock, Reset)
    begin
      if (Reset = '0') then
        B <= x"00";
      elsif (Clock'event and Clock='1') then
          if (B_EN = '1') then
            B <= Data_Bus;
          end if;
      end if;
  end process;
  ----------------------------------------
  C_REG : process (Clock, Reset)
    begin
      if (Reset = '0') then
        C <= x"00";
      elsif (Clock'event and Clock='1') then
          if (C_EN = '1') then
            C <= Data_Bus;
          end if;
      end if;
  end process;

end architecture;
```

Each register is modeled as a separate process. 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 9.29
Registers as agents on a data bus – RTL model in VHDL

Example 9.30 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 9.30
Registers as agents on a data bus – simulation waveform

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/then 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 process and a sensitivity list. The clock and reset signals are always listed by themselves in the sensitivity list. Within the process is an if/then statement. The first clause of the if/then statement handles the asynchronous reset condition while the second *elsif* clause handles the synchronous signal assignments.
- Edge sensitivity is modeled within a process using either the (*clock'event and clock = "1"*) syntax or an edge detection function provided by the STD_LOGIC_1164 package (i.e., *rising_edge()*).
- Most D-Flip-Flops and registers contain a synchronous *enable* line. This is modeled using a nested if/then statement within the main process if/then statement. The nested if/then goes beneath the clause for the synchronous signal assignments.
- 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 is 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 if it is desired to explicitly define the state codes.

- Counters are a special type of finite state machine that can be modeled using a single process. Only the clock, and reset signals are listed in the sensitivity list of the counter process.
- Registers are modeled in VHDL in a similar manner to a D-Flip-Flop with a synchronous enable. The only difference is that the inputs and outputs are n-bit vectors.

Exercise Problems

Section 9.1: Modeling Sequential Storage Devices in VHDL

9.1.1 How does a VHDL model for a D-Flip-Flop handle treating reset as the highest priority input?

9.1.2 For a VHDL 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 VHDL 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 VHDL model of a D-Flip-Flop with a synchronous enable (EN), why is the behavior of the enable modeled using a nested if/then statement under the clock edge clause rather than an additional elsif clause in the primary if/then statement?

Section 9.2: Modeling Finite State Machines in VHDL

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

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

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

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

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

9.2.6 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?

9.2.7 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?

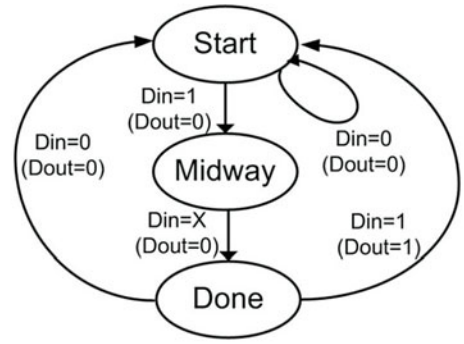
9.2.8 When using the three-process behavioral modeling approach for finite state machines, what inputs are listed in the sensitivity list of the output logic process?

9.2.9 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?

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

Section 9.3: FSM Design Examples in VHDL

9.3.1 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

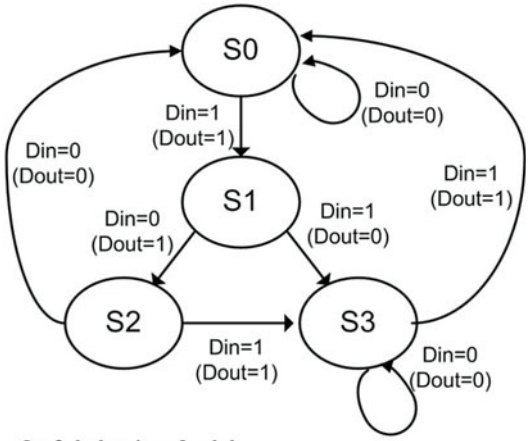
```
entity fsm1_behavioral is
    port   (Clock, Reset : in   std_logic;
            Din         : in   std_logic;
            Dout        : out  std_logic);
end entity;
```

Fig. 9.1
FSM 1 state diagram and entity

9.3.2 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".

9.3.3 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

9.3.4 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".

9.3.5 Design a VHDL behavioral model for a 4-bit serial bit sequence detector similar to Example 9.11. 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

9.3.6 Design a VHDL behavioral model for a 20-cent vending machine controller similar to Example 9.14. 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

9.3.7 Design a VHDL behavioral model for a finite state machine for a traffic light controller. Use the entity 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 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

Section 9.4: Modeling Counters in VHDL

9.4.1 Design a VHDL behavioral model for a 16-bit, binary up counter using a single process. The block diagram for the entity definition is shown in Fig. 9.6. In your model, declare Count_Out to be of type unsigned, and implement the internal counter functionality with a signal of type unsigned.

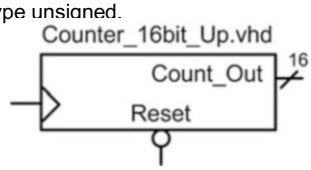


Fig. 9.6
16-bit binary up counter block diagram

9.4.2 Design a VHDL behavioral model for a 16-bit, binary up counter using a single process. The block diagram for the entity definition is shown in Fig. 9.6. In your model, declare Count_Out to be of type unsigned and implement the internal counter functionality with a signal of type integer.

9.4.3 Design a VHDL behavioral model for a 16-bit, binary up counter using a single process. The block diagram for the entity definition is shown in Fig. 9.6. In your model, declare Count_Out to be of type std_logic_vector and implement the internal counter functionality with a signal of type integer.

9.4.4 Design a VHDL behavioral model for a 16-bit, binary up counter with enable using a single process. The block diagram for the entity definition is shown in Fig. 9.7. In your model, declare Count_Out to be of type unsigned and implement the internal counter functionality with a signal of type unsigned.

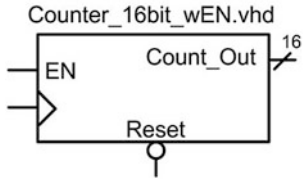


Fig. 9.7
16-bit binary up counter with enable block diagram

9.4.5 Design a VHDL behavioral model for a 16-bit, binary up counter with enable using a single process. The block diagram for the entity definition is shown in Fig. 9.7. In your model, declare Count_Out to be of type unsigned and implement the internal counter functionality with a signal of type integer.

9.4.6 Design a VHDL behavioral model for a 16-bit, binary up counter with enable using a single process. The block diagram for the entity definition is shown in Fig. 9.7. In your model, declare Count_Out to be of type std_logic_vector and implement the internal counter functionality with a signal of type integer.

9.4.7 Design a VHDL behavioral model for a 16-bit, binary up counter with enable and load using a single process. The block diagram for the entity definition is shown in Fig. 9.8. In your model, declare Count_Out to be of type unsigned and implement the internal counter functionality with a signal of type unsigned.

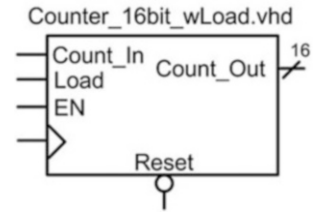


Fig. 9.8
16-bit binary up counter with load block diagram

9.4.8 Design a VHDL behavioral model for a 16-bit, binary up counter with enable and load using a single process. The block diagram for the entity definition is shown in Fig. 9.8. In your model, declare Count_Out to be of type unsigned and implement the internal counter functionality with a signal of type integer.

9.4.9 Design a VHDL behavioral model for a 16-bit, binary up counter with enable and load using a single process. The block diagram for the entity definition is shown in Fig. 9.8. In your model, declare Count_Out to be of type std_logic_vector and implement the internal counter functionality with a signal of type integer.

9.4.10 Design a VHDL behavioral model for a 16-bit, binary up/down counter using a single process. The block diagram for the entity definition is shown in Fig. 9.9. When Up = 1, the counter will increment. When Up = 0, the counter will decrement. In your model, declare Count_Out to be of type unsigned, and implement the internal counter functionality with a signal of type unsigned.

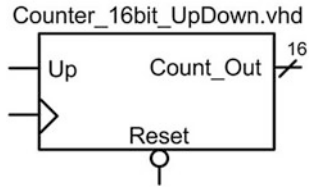


Fig. 9.9
16-bit binary up/down counter block diagram

9.4.11 Design a VHDL behavioral model for a 16-bit, binary up/down counter using a single process. The block diagram for the entity definition is shown in Fig. 9.9. When Up = 1, the counter will increment. When Up = 0, the counter will decrement. In your model, declare Count_Out to be of type unsigned, and implement the internal counter functionality with a signal of type integer.

9.4.12 Design a VHDL behavioral model for a 16-bit, binary up/down counter using a single process. The block diagram for the entity definition is shown in Fig. 9.9. When Up = 1, the counter will increment. When Up = 0, the counter will decrement. In your model, declare Count_Out to be of type std_logic_vector, and implement the internal counter functionality with a signal of type integer.

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 VHDL RTL model of a 32-bit, synchronous register. The block diagram for the entity definition is shown in Fig. 9.10. The register has a synchronous enable. The register should be modeled using a single process.

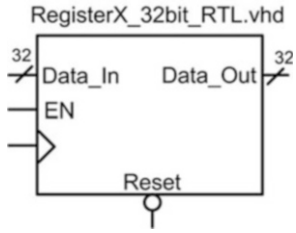


Fig. 9.10
32-bit register block diagram

9.5.4 Design a VHDL RTL model of an 8-stage, 16-bit shift register. The block diagram for the entity 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). Use std_logic or std_logic_vector for all ports.

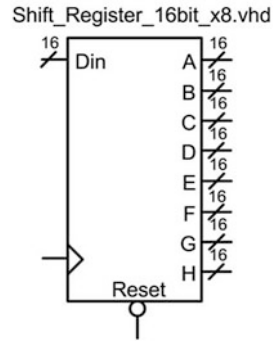


Fig. 9.11
16-bit shift register block diagram

9.5.5 Design a VHDL 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 process.

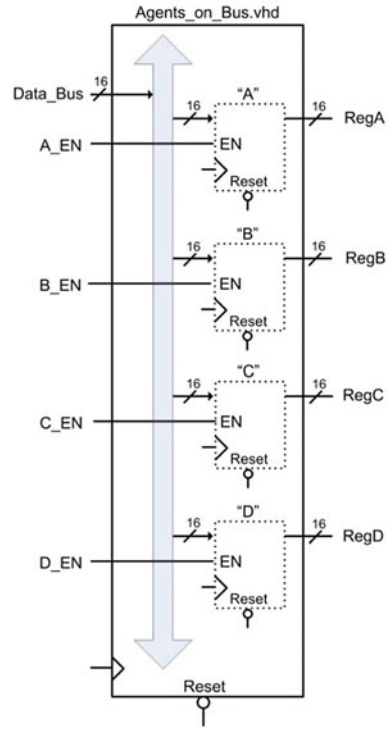


Fig. 9.12
Agents on a bus block diagram