Chapter 8: Modeling Sequential Storage and Registers

In this chapter, we will look at modeling sequential storage devices. We begin by looking at modeling scalar storage devices such as D-latches and D-flip-flops and then move into multiple-bit storage models known as registers.

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

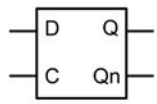
8.1 Design a VHDL model for a single-bit sequential logic storage device.
8.2 Design a VHDL model for a register.

8.1 Modeling Scalar Storage Devices

8.1.1 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 8.1 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;
```

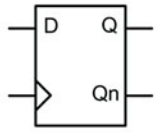
Example 8.1
Behavioral model of a D-latch in VHDL

B. J. LaMeres, *Quick Start Guide to VHDL*, https://doi.org/10.1007/978-3-030-04516-6_8

8.1.2 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 8.2 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

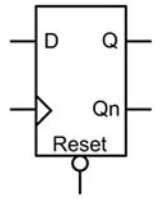
```
library IEEE;
use IEEE.std_logic_1164.all;

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 8.2
Behavioral model of a D-flip-flop in VHDL

8.1.3 D-Flip-Flop with Asynchronous Resets

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 startup 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 8.3 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

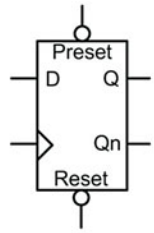
```
library IEEE;
use IEEE.std_logic_1164.all;

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 8.3
Behavioral model of a D-flip-flop with asynchronous reset in VHDL

8.1.4 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 8.4 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

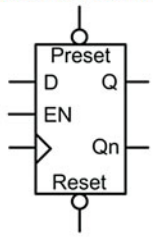
```
library IEEE;
use IEEE.std_logic_1164.all;

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 8.4
Behavioral model of a D-flip-flop with asynchronous reset and preset in VHDL

8.1.5 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 8.5 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 8.5
Behavioral model of a D-flip-flop with synchronous enable in VHDL

Concept Check

CC8.1 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

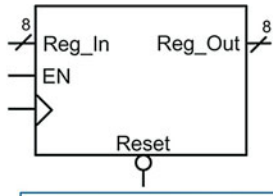
8.2 Modeling Registers

8.2.1 Registers with Enables

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. Register transfer level (RTL) 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. Example 8.6 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

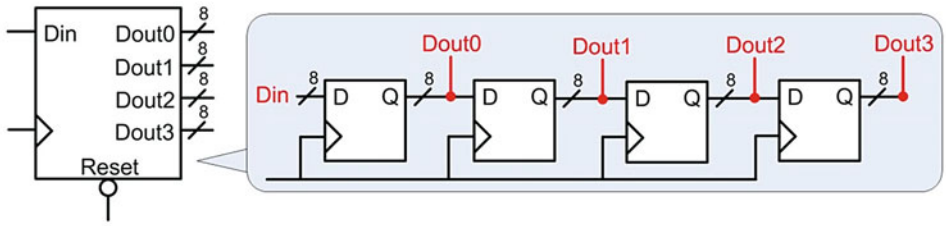
```
library IEEE;
use IEEE.std_logic_1164.all;

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 8.6
RTL model of an 8-bit register in VHDL

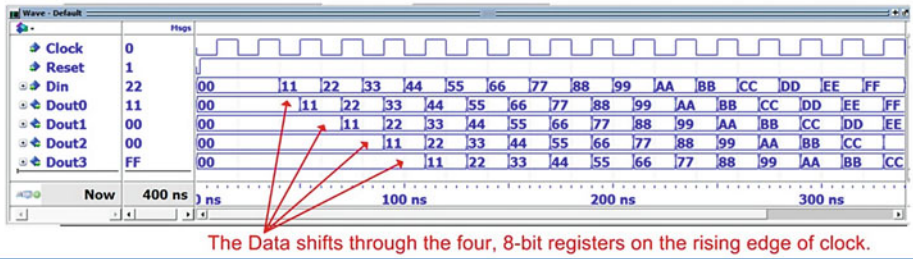
8.2.2 Shift Registers

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 8.7 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;
```



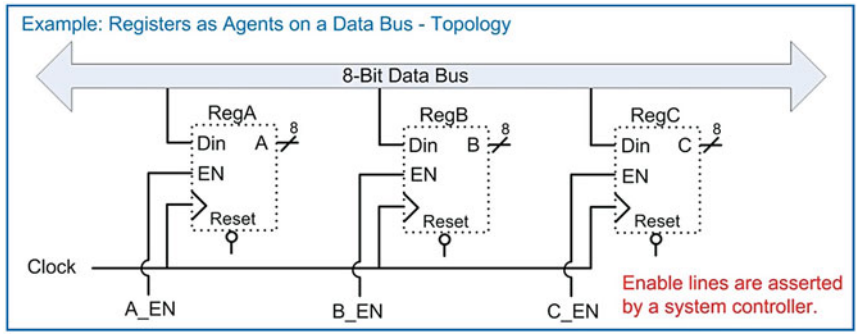
The Data shifts through the four, 8-bit registers on the rising edge of clock.

Example 8.7
RTL model of a 4-stage, 8-bit shift register in VHDL

8.2.3 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 are 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 8.8 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: Registers as Agents on a Data Bus - Topology



Example 8.8
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 8.9 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

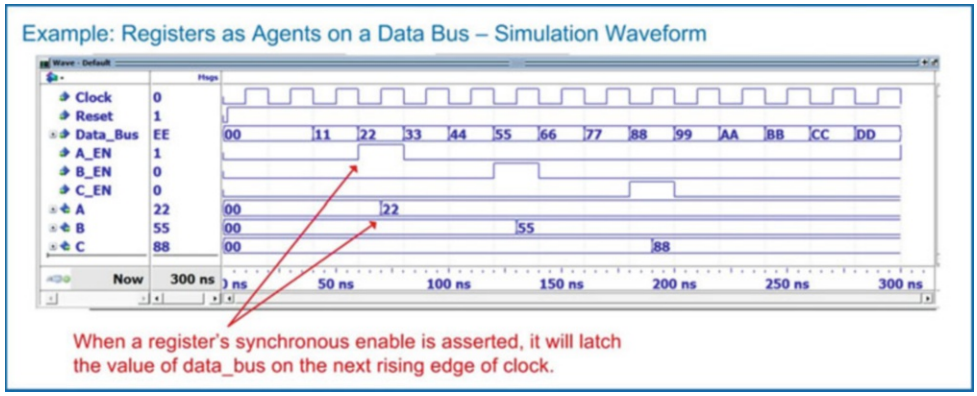
```
library IEEE;
use IEEE.std_logic_1164.all;

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 8.9
Registers as agents on a data bus: RTL model in VHDL

Example 8.10 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 8.10
Registers as agents on a data bus: simulation waveform

Concept Check

CC8.2 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.
- 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 8.1: Modeling Scalar Storage Devices

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

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

8.1.3 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?

8.1.4 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 8.2: Modeling Registers

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

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

8.2.3 Design a VHDL RTL model of a 32-bit, synchronous register. The block diagram for the entity definition is shown in Fig. 8.1. The register has a synchronous enable. The register should be modeled using a single process.

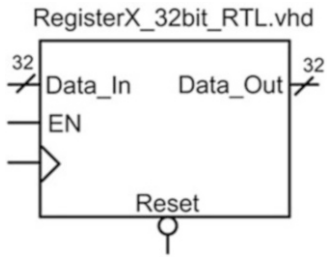


Fig. 8.1
32-Bit Register block diagram

8.2.4 Design a VHDL RTL model of an 8-stage, 16-bit shift register. The block diagram for the entity definition is shown in Fig. 8.2. 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.

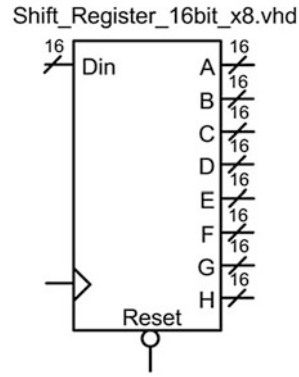


Fig. 8.2
16-Bit shift register block diagram

8.2.5 Design a VHDL RTL model of the multi-drop bus topology in Fig. 8.3. 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.

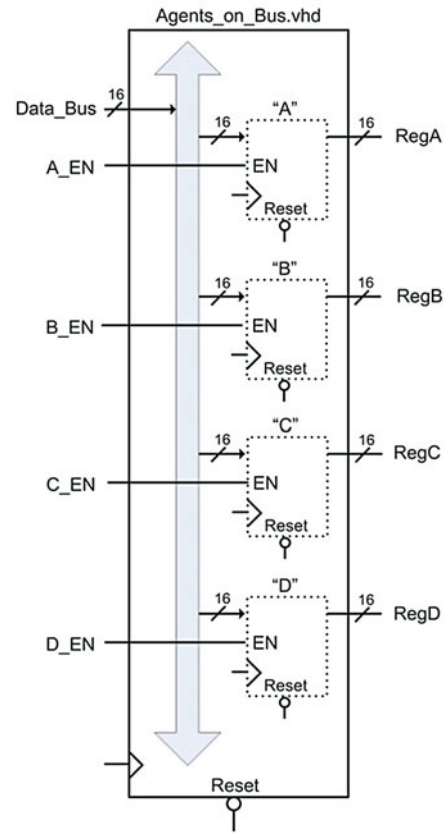


Fig. 8.3
Agents on a bus block diagram