Chapter 4: Structural Design and Hierarchy

This chapter describes how to accomplish hierarchy within VHDL using lower-level sub-systems. Structural design in VHDL refers to including lower-level sub-systems within a higher-level system in order to produce the desired functionality. A purely structural VHDL design would not contain any behavioral modeling in the architecture such as signal assignments but instead just contain the instantiation and interconnections of other sub-systems.

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

4.1 Instantiate and map the ports of a lower-level component in VHDL.
4.2 Design a VHDL model for a system that uses hierarchy.

4.1 Components

4.1.1 Component Instantiation

A sub-system is called a **component** in VHDL. For any component that is going to be used in an architecture, it must be declared before the begin statement. Refer to Sect. 2.2.3.3 for the syntax of declaring a component. A specific component only needs to be declared once. After the begin statement, it can be used as many times as necessary. Each component is executed concurrently.

The term *instantiation* refers to the *use* or *inclusion* of the component in the VHDL system. When a component is instantiated, it needs to be given a unique identifying name. This is called the *instance name*. To instantiate a component, the instance name is given first, followed by a colon and then the component name. The last part of instantiating a component is connecting signals to its ports. The way in which signals are connected to the ports of the component is called the **port map**. The syntax for instantiating a component is as follows:

```
instance_name : <component name>
  port map (<port connections>);
```

4.1.2 Port Mapping

There are two techniques to connect signals to the ports of the component, *explicit port mapping* and *positional port mapping*.

4.1.2.1 Explicit Port Mapping

In explicit port mapping, the name of each port of the component is given, followed by the connection indicator =**>**, followed by the signal it is connected to. The port connections can be listed in any order since the details of the connection (i.e., port name to signal name) are explicit. Each connection name is separated by a comma. The syntax for explicit port mapping is as follows:

```
instance_name : <component name>
  port map (port1 => signal1, port2 => signal2, ...);
```

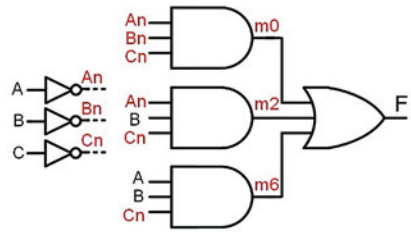

B. J. LaMeres, *Quick Start Guide to VHDL*, https://doi.org/10.1007/978-3-030-04516-6_4

Example 4.1 shows how to design a VHDL model of a combinational logic circuit using structural VHDL and explicit port mapping. Note that this example again uses the same truth table as in Examples 3.1, 3.10, and 3.16 to illustrate a comparison between approaches.

Example: Modeling Logic using Structural VHDL (Explicit Port Mapping)

Implement the following truth table with structural VHDL using lower level sub-systems for the basic gates. We will assume that VHDL designs have been completed for the inverter, AND gate, and OR gate. The entities for these designs are provided.

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0




```
entity INV1 is
 port (A : in  bit;
       F : out bit);
end entity;
```

```
entity AND3 is
 port (A,B,C : in  bit;
       F     : out bit);
end entity;
```

```
entity OR3 is
 port (A,B,C : in  bit;
       F     : out bit);
end entity;
```

The basic gate designs can be declared as components in our system and then instantiated in order to describe the sum of products logic diagram above.

```
entity SystemX is
   port (A, B, C  :  in  bit;
         F        :  out bit);
end entity;

architecture SystemX_arch of SystemX is

  signal  An, Bn, Cn : bit;  -- declare signals
  signal  m0, m2, m6 : bit;

  component INV1             -- declare INV1
     port (A : in  bit;
           F : out bit);
  end component;

  component AND3             -- declare AND3
     port (A,B,C : in  bit;
           F     : out bit);
  end component;

  component OR3
     port (A,B,C : in  bit;  -- declare OR3
           F     : out bit);
  end component;

begin

  U1 : INV1 port map (A=>A, F=>An);
  U2 : INV1 port map (A=>B, F=>Bn);
  U3 : INV1 port map (A=>C, F=>Cn);

  U4 : AND3 port map (A=>An, B=>Bn, C=>Cn, F=>m0);
  U5 : AND3 port map (A=>An, B=>B,  C=>Cn, F=>m2);
  U6 : AND3 port map (A=>A,  B=>B,  C=>Cn, F=>m6);

  U7 : OR3  port map (A=>m0, B=>m2, C=>m6, F=>F);

end architecture;
```

The entity is named SystemX.

Internal signals are needed to connect the sub-systems.

The three lower level sub-systems are declared as components in SystemX.

The components are instantiated and connected using explicit port mapping in order to describe the behavior of the logic diagram.

NOT's

AND's

OR

Example 4.1
Modeling logic using structural VHDL (explicit port mapping)

4.1.2.2 Positional Port Mapping

In positional port mapping, the names of the ports of the component are not explicitly listed. Instead, the signals are listed in the same order that the ports of the component were defined. Each signal name is separated by a comma. This approach requires less text to describe but can also lead to misconnections due to mismatches in the order of the signals being connected. The syntax for positional port mapping is as follows:

```
instance_name : <component name>
  port map (signal1, signal2, ...);
```

Example 4.2 shows how to create the same structural VHDL model as in Example 4.1, but using positional port mapping instead.

Example: Modeling Logic using Structural VHDL (Positional Port Mapping)

In positional port mapping the port names are not listed in the component instantiation. Instead, the signals are simply listed in the same order as the ports were defined. The signal listed first will be connected to the port defined first. The signal listed second will be connected to the port defined second, etc.

Explicit Port Mapping

```
begin

  U1 : INV1 port map (A=>A, F=>An);
  U2 : INV1 port map (A=>B, F=>Bn);
  U3 : INV1 port map (A=>C, F=>Cn);

  U4 : AND3 port map (A=>An, B=>Bn, C=>Cn, F=>m0);
  U5 : AND3 port map (A=>An, B=>B,  C=>Cn, F=>m2);
  U6 : AND3 port map (A=>A,  B=>B,  C=>Cn, F=>m6);

  U7 : OR3  port map (A=>m0, B=>m2, C=>m6, F=>F);
```

Positional Port Mapping of Same System

```
begin

  U1 : INV1 port map (A, An);
  U2 : INV1 port map (B, Bn);
  U3 : INV1 port map (C, Cn);

  U4 : AND3 port map (An, Bn, Cn, m0);
  U5 : AND3 port map (An, B,  Cn, m2);
  U6 : AND3 port map (A,  B,  Cn, m6);

  U7 : OR3  port map (m0, m2, m6, F);
```

Example 4.2
Modeling logic using structural VHDL (positional port mapping)

Concept Check

CC4.1 Does the use of components model concurrent functionality? Why?

(A) No. Since the lower-level behavior of the component being instantiated may contain non-concurrent behavior, it is not known what functionality will be modeled.

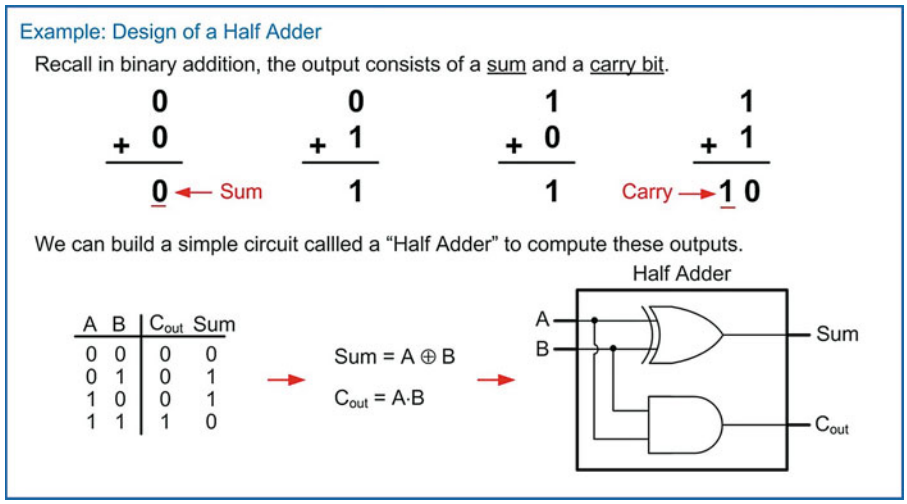
(B) Yes. The components are treated like independent sub-systems whose behavior runs in parallel just as if separate parts were placed in a design.

4.2 Structural Design Examples: Ripple Carry Adder

This section gives an example of a structural design that implements a simple binary adder.

4.2.1 Half Adders

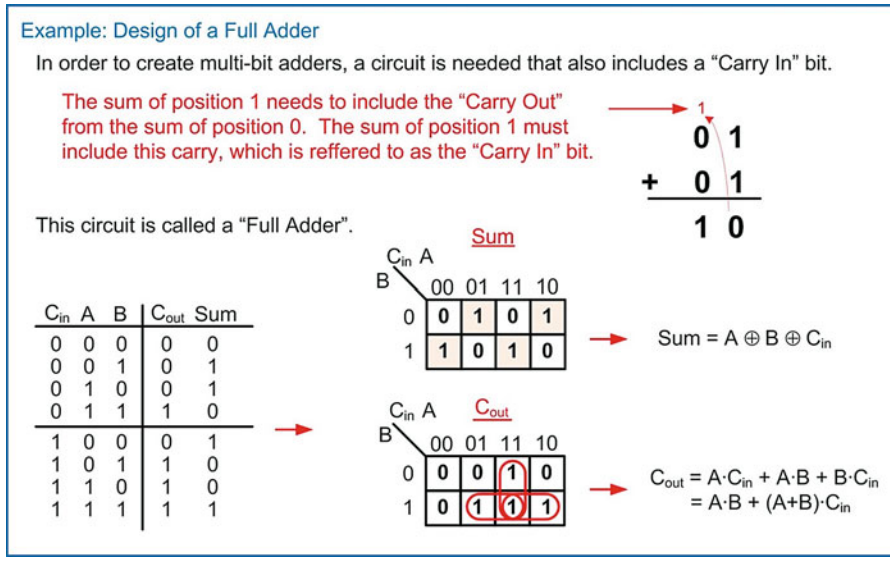
When creating a binary adder, it is desirable to design incremental sub-systems that can be reused. This reduces design effort and minimizes troubleshooting complexity. The most basic component in the adder is called a *half adder*. This circuit computes the sum and carry out on two input arguments. The reason it is called a half adder instead of a full adder is because it does not accommodate a *carry in* during the computation; thus it does not provide all the necessary functionality required for the positional adder. Example 4.3 shows the design of a half adder. Notice that two combinational logic circuits are required in order to produce the sum (the XOR gate) and the carry out (the AND gate). These two gates are in parallel to each other; thus the delay through the half adder is due to only one level of logic.



Example 4.3
Design of a half adder

4.2.2 Full Adders

A full adder is a circuit that still produces a sum and carry out but considers three inputs in the computations (A, B, and C_{in}). Example 4.4 shows the design of a full adder using the classical design approach. This step is shown to illustrate why it is possible to reuse half adders to create the full adder. In order to do this, it is necessary to have the minimal sum of products logic expression.



Example 4.4
Design of a full adder

As mentioned before, it is desirable to reuse design components as we construct more complex systems. One such design reuse approach is to create a full adder using two half adders. This is straightforward for the sum output since the logic is simply two cascaded XOR gates (Sum = A⊕B⊕Cin). The carry out is not as straightforward. Notice that the expression for Cout derived in Example 4.4 contains the term (A + B). If this term could be manipulated to use an XOR gate instead, it would allow the full adder to take advantage of existing circuitry in the system. Figure 4.1 shows a derivation of an equivalency that allows (A + B) to be replaced with (A⊕B) in the Cout logic expression.

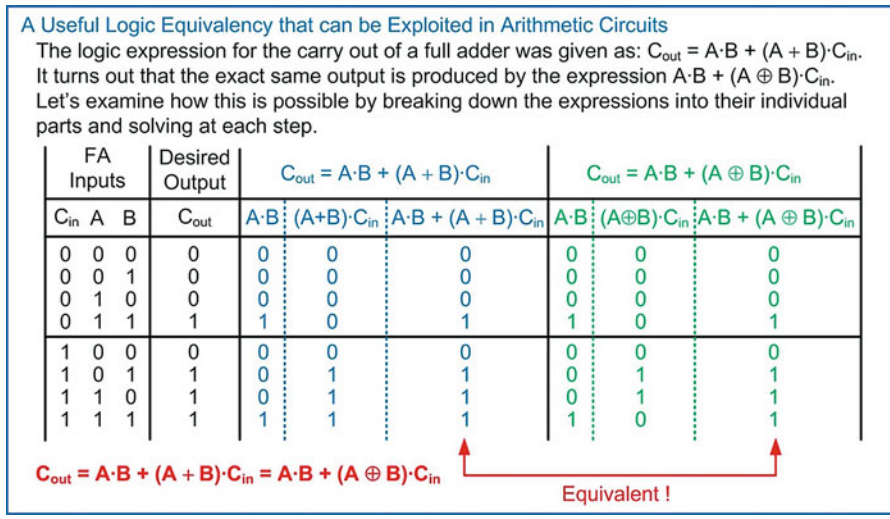
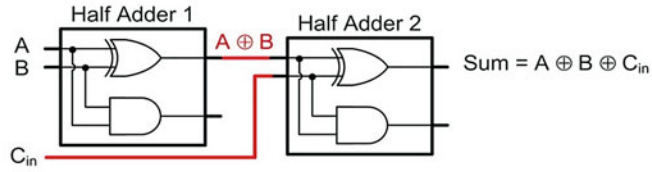


Fig. 4.1
A useful logic equivalency that can be exploited in arithmetic circuits

The ability to implement the carry out logic using the expression $C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$ allows us to implement a full adder with two half adders and the addition of a single OR gate. Example 4.5 shows this approach. In this new configuration, the sum is produced in two levels of logic, while the carry out is produced in three levels of logic.

Example – Design of a Full Adder Out of Two Half Adders

It is often desirable to create a full adder out of two half adders in order to re-use existing design components. The "Sum" of the full adder can be created by using two cascaded XOR gates provided by the half adders.



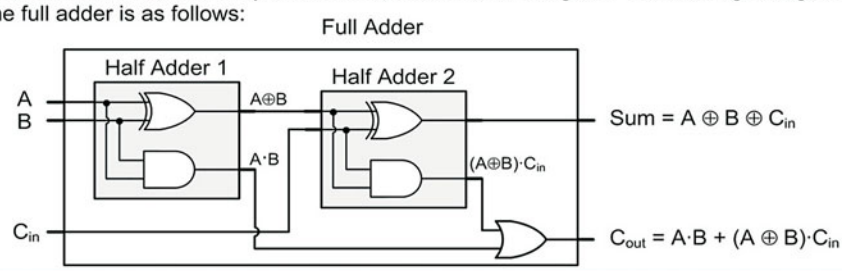
The expression for the "Carry Out" of the full adder is:

$$C_{out} = A \cdot B + (A + B) \cdot C_{in}$$

or

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

Notice that the carry out of Half Adder 1 produces the $A \cdot B$ term in this expression. Also notice that the carry out of Half Adder 2 produces the $(A \oplus B) \cdot C_{in}$ term. The only remaining logic needed to create the carry out of the full adder is an OR gate. The final logic diagram for the full adder is as follows:



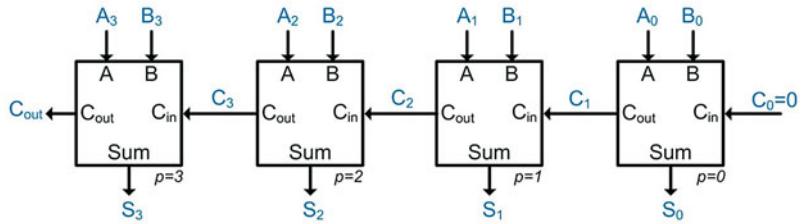
Example 4.5
Design of a full adder out of half adders

4.2.3 Ripple Carry Adder (RCA)

The full adder can now be used in the creation of multi-bit adders. The simplest architecture exploiting the full adder is called a *ripple carry adder* (RCA). In this approach, full adders are used to create the sum and carry out of each bit position. The carry out of each full adder is used as the carry in for the next higher position. Since each subsequent full adder needs to wait for the carry to be produced by the preceding stage, the carry is said to *ripple* through the circuit, thus giving this approach its name. Example 4.6 shows how to design a 4-bit ripple carry adder using a chain of full adders. Notice that the carry in for the full adder in position 0 is tied to a logic 0. The 0 input has no impact on the result of the sum but enables a full adder to be used in the 0th position.

Example: Design of a 4-Bit Ripple Carry Adder (RCA)

Full adders can be cascaded together to form a multi-bit adder. The symbols are typically drawn in the following fashion to mirror a positional number system.



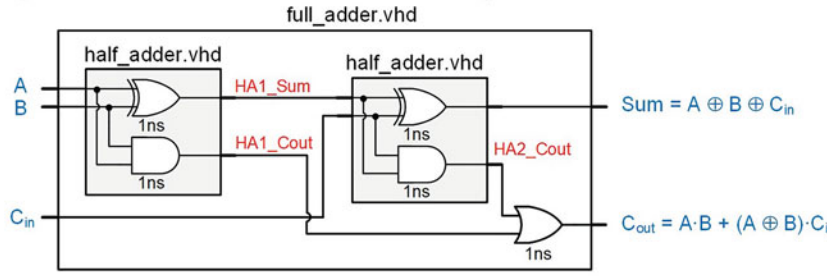
The sum of position 1 cannot complete until it receives the carry in (C_1) from the sum in position 0. The position 2 sum cannot complete until it receives the carry in (C_2) from the sum in position 1, etc. In this way, the carry "ripples" through the circuit from right to left. This configuration is known as a Ripple Carry Adder (RCA).

Example 4.6
Design of a 4-bit ripple carry adder (RCA)

4.2.4 Structural Model of a Ripple Carry Adder in VHDL

Now that the hierarchical design of the RCA is complete, we can now model it in VHDL as a system of lower-level components. Example 4.7 shows the structural model for a full adder in VHDL consisting of two half adders. The full adder is created by instantiating two versions of the half adder as components. In this example, all gates are modeled with a delay of 1 ns.

Example: Structural Model of a Full Adder in VHDL Using Two Half Adders



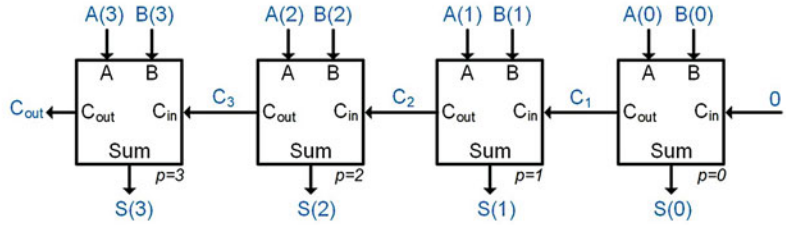

```
entity half_adder is
  port (A, B       : in  bit;
        Sum, Cout  : out bit);
end entity;

architecture half_adder_arch of half_adder is

 begin

   Sum  <= A xor B after 1 ns;
   Cout <= A and B after 1 ns;

end architecture;
```

```
entity full_adder is
  port (A, B, Cin  : in  bit;
        Sum, Cout  : out bit);
end entity;

architecture full_adder_arch of full_adder is

   component half_adder
     port (A, B       : in  bit;
           Sum, Cout  : out bit);
   end component;

   signal HA1_Sum, HA1_Cout, HA2_Cout : std_logic;

 begin

   HA1 : half_adder port map (A, B, HA1_Sum, HA1_Cout);
   HA2 : half_adder port map (HA1_Sum, Cin, Sum, HA2_Cout);

   Cout <= HA1_Cout or HA2_Cout after 1 ns;

end architecture;
```

Example 4.7
Structural model of a full adder in VHDL using two half adders

Example 4.8 shows the structural model of a 4-bit ripple carry adder in VHDL. The RCA is created by instantiating four full adders. Notice that a logic 0 can be directly inserted into the port map of the first full adder to model the behavior of $C_0 = 0$.

Example: Structural Model of a 4-Bit Ripple Carry Adder in VHDL




```
entity rca_4bit is
  port (A, B  : in  bit_vector(3 downto 0);
        Sum   : out bit_vector(3 downto 0);
        Cout  : out bit);
end entity;

architecture rca_4bit_arch of rca_4bit is

   component full_adder
     port (A, B, Cin  : in  bit;
           Sum, Cout  : out bit);
   end component;

   signal C1, C2, C3 : std_logic;

 begin

   A0 : full_adder port map (A(0), B(0), '0', Sum(0), C1);
   A1 : full_adder port map (A(1), B(1), C1,  Sum(1), C2);
   A2 : full_adder port map (A(2), B(2), C2,  Sum(2), C3);
   A3 : full_adder port map (A(3), B(3), C3,  Sum(3), Cout);

end architecture;
```

Example 4.8
Structural model of a 4-bit ripple carry adder in VHDL

Concept Check

CC4.2 Why is the use of hierarchy considered a good design practice?

(A) Hierarchy allows the design to be broken into smaller pieces, each with simpler functionality that can be verified independently prior to being used in a higher-level system.

(B) Hierarchy allows a large system to be broken into smaller sub-systems that can be designed by multiple engineers, thus decreasing the overall development time.

(C) Hierarchy allows a large system to be broken down into smaller sub-systems that can be more easily understood so that debugging is more manageable.

(D) All of the above.

Summary

- A *component* is how a VHDL system uses another VHDL file as a sub-system.
- VHDL components are treated as concurrent sub-systems.
- To use a component, it must first be *declared*, which defines the name and entity of the sub-system to be used. This occurs before the *begin* statement in the architecture.

- A component can be *instantiated* one or more times, which includes one or more copies of the sub-system in the higher-level system. This occurs after the *begin* statement in the architecture.
- The ports of the component can be connected using either *explicit* or *positional port mapping*.
- Explicit port mapping involves listing both the names of the lower-level component's ports along with the higher-level signals that form the connection. The connections in explicit port mapping can be listed in any order. Explicit port mapping is less prone to mistaken connections.
- Positional port mapping involves listing only the names of the higher-level signals during instantiation. The order in which the signals are listed will be connected to the ports of the lower-level sub-system in the order that the ports were declared. Positional port mapping provides a more compact approach to port mapping. Positional port mapping is more prone to mistaken connections due to potentially listing the signals in the wrong order during mapping.

Exercise Problems

Section 4.1: Components

4.1.1 How many times does a component need to be declared within an architecture?

4.1.2 How many times can a component be instantiated?

4.1.3 Does declaring a component occur before or after the *begin* statement in the architecture?

4.1.4 Does instantiating a component occur before or after the *begin* statement in the architecture?

4.1.5 Which port mapping technique is more compact, explicit or positional?

4.1.6 Which port mapping technique is less prone to connection errors because the names of the lower-level ports are listed within the mapping?

Section 4.2: Structural Design Examples

4.2.1 Design a VHDL model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4, etc.) and then instantiate them in your upper-level architecture to create the desired functionality. The lower-level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

4.2.2 Design a VHDL model to implement the behavior described by the 3-input maxterm list shown in Fig. 3.2. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4, etc.) and then instantiate them in your upper-level architecture to create the desired functionality. The lower-level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

4.2.3 Design a VHDL model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4, etc.) and then instantiate them in your upper-level architecture to create the desired functionality. The lower-level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

4.2.4 Design a VHDL model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4, etc.) and then instantiate them in your upper-level architecture to create the desired functionality. The lower-level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

4.2.5 Design a VHDL model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4, etc.) and then instantiate them in your upper-level architecture to create the desired functionality. The lower-level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

4.2.6 Design a VHDL model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4, etc.) and then instantiate them in your upper-level architecture to create the desired functionality. The lower-level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.