CHAPTER 13

HIERARCHICAL DESIGN IN VHDL

As the size of a digital system increases, its complexity grows accordingly. One method of managing the complexity is to describe the system in a hierarchical structure, in which the system is gradually divided into smaller parts. With a hierarchy, we only need to focus on a small, manageable part at a time. One of the goals of VHDL is to facilitate the development and modeling of large digital systems. It consists of versatile mechanisms and language constructs to specify and configure a design hierarchy and to organize design information and files. This chapter provides an overview of constructs relevant to the RT-level design and synthesis.

13.1 INTRODUCTION

Hierarchical design is a methodology that divides a system recursively into small modules and then constructs each module independently. The term *recursively* means that the division process can be applied repeatedly and the modules can be further decomposed. For example, consider the sequential multiplier of Section 11.3.3. One possible design hierarchy is shown in Figure 13.1. The system is first divided into a control path and a data path. The control path is then divided into the next-state logic and the state register, and the data path is divided into a routing circuit, functional units and a data register. The functional units are then further decomposed into an adder and a decrementor. If needed, we can continue the process and further refine the leaf modules. The sequential multiplier can also be part of a larger system. For example, it can be a module of an arithmetic unit, which in turn, can be a module of a processor.

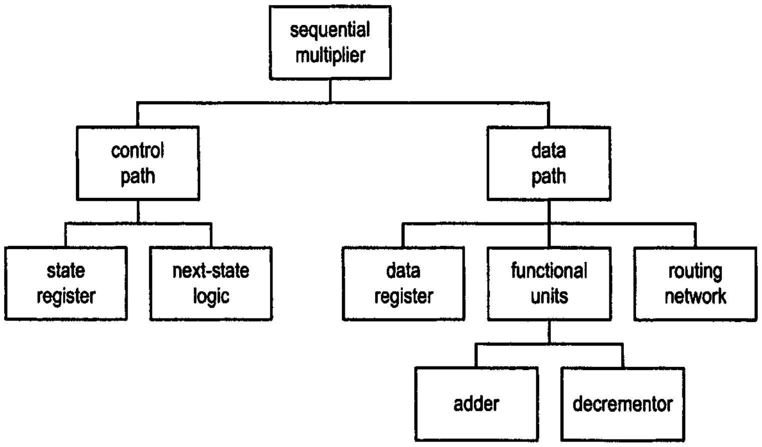

Figure 13.1 Hierarchical description of a sequential multiplier.

13.1.1 Benefits of hierarchical design

There are two major benefits of using the hierarchy: *complexity management* and *design reuse*. As the size of transistor continues to decrease, more functionality can be included in a device and the digital system grows larger and more complex. Managing the complexity becomes a key challenge in today's design. Hierarchical design methodology allows us to apply the divide-and-conquer strategy and break a system into smaller modules. This approach helps us to manage a large design in several ways:

- Instead of looking at the entire system, we can focus on a manageable portion of the system, and analyze, design and verify each module in isolation.
- Once the hierarchy and modules are specified, a large system can be constructed in stages by a designer or concurrently by a team of designers.
- The synthesis software may require a significant amount of memory space and computation time to synthesize a large system. Breaking the system into smaller modules and synthesizing them independently can make the process more effective.

Hierarchical design methodology also helps to facilitate design reuse:

- Some predesigned modules or third-party cores (i.e., IPs) may exist and can be used in the system. Therefore, we don't need to construct every system from scratch.
- Many systems contain some common or similar functionalities. After we design and verify a module in a system, the same module can be used in future design.
- Some design may contain certain device-dependent components, such as an SRAM module. To achieve portability, we can isolate these components in the top level of the hierarchy and substitute them according to the target technology.

13.1.2 VHDL constructs for hierarchical design

One objective of VHDL is to facilitate the modeling and developments of complex digital systems. Many language constructs are designed for this purpose. These include the following:

- Component
- Generic

- Configuration
- Library
- Package
- Subprogram

The *component*, *generic* and *configuration* constructs provide flexible and versatile mechanism to describe a hierarchical design. These constructs are discussed in Sections 13.2, 13.3 and 13.4. The *library*, *package*, and *subprogram* help the management of complicated code and are briefly reviewed in Section 13.5. To take full advantage of the hierarchical design methodology, we have to develop general and flexible modules. This issue is discussed in Chapters 14 and 15.

13.2 COMPONENTS

Hierarchical design methodology basically divides a system into smaller modules and then constructs the modules accordingly. Although not stated explicitly, our previous derivations generally followed this approach. We usually started with a top-level diagram with several major parts and then derived the VHDL code according to the diagram, with a VHDL segment for each part. The VHDL component construct provides a formal and explicit way to describe a hierarchical design.

We examined the VHDL component construct briefly in Section 2.2.2. It is the mechanism used to describe a digital system in a structural view. Recall that a structural view is essentially a block diagram, in which we specify the types of parts used and the interconnections among these parts. While the component construct is supported in both VHDL 87 and VHDL 93, the syntax of VHDL 93 is much simpler. However, since the IEEE RTL synthesis standard is based on VHDL 87, it follows the old syntax. To obtain maximal portability, our discussion mainly follows the IEEE RTL synthesis standard (i.e., VHDL 87). In Section 13.4.4, we briefly examine the newer version.

In VHDL 87, using a component involves two steps. The first step is *component declaration*, in which a component is "make known" to an architecture. The second step is *component instantiation*, in which an instance of the component is created and its external I/O interface is specified.

13.2.1 Component declaration

Component declaration provides information about the external interface of a component, which includes the input and output ports and relevant parameters. The information is similar to that provided in an entity declaration. The simplified syntax of component declaration is as follows:

```
component component_name
   generic(
      generic_declaration;
      generic_declaration;
      . . .
   );
   port(
      port_declaration;
      port_declaration;
      . . .
   );
```

```
end component;
```

The generic portion is optional and consists of relevant parameters to be passed into the component. It is discussed in the next section. The port portion consists of port declarations, which are similar to those in an entity declaration. Note that the **is** keyword is in the entity declaration but not in the component declaration (the **is** keyword is allowed in VHDL 93). As in entity declaration, no information about internal implementation is specified in component declaration.

Assume that we have already designed a decade (i.e., mod-10) counter and that its entity declaration is

```
entity dec_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
   );
end dec_counter;
```

If we want to use it as a component in other designs, the easiest way is to declare a component that has the same name and ports:

```
component dec_counter
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
   );
end component;
```

Note that the same information is used in the entity declaration and the corresponding component declaration. Graphically, a component can be thought of as a circuit part with properly named input and output ports. The conceptual diagram of the `dec_counter` component is shown in Figure 13.2(a).

During the elaboration process, the component eventually has to be bound with an architecture body. The complete VHDL code of the decade counter is shown in Listing 13.1. The en input functions as an enable signal. The `pulse` output is a status signal and is asserted when the counter reaches 9 and is ready to warp around.

Listing 13.1 Decade counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity dec_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
   );
end dec_counter;
```

```
architecture up_arch of dec_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   constant TEN: integer:= 10;
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
          if r_reg=(TEN-1) then
             r_next <= (others=>'0');
          else
             r_next <= r_reg + 1;
          end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
   pulse <= '1' when r_reg=(TEN-1) else
            '0';
end up_arch;
```

In a VHDL code, a component declaration is included in the declaration part of an architecture body, as shown in Listing 13.2. If it is used in multiple architecture bodies, the declaration may be placed in a package. Use of packages is discussed in Section 13.5.3.

13.2.2 Component instantiation

Once a component is declared, its instance can be created inside the architecture body. The simplified syntax of component instantiation is

```
instance_label: component_name
   generic map(
      generic_association;
      generic_association;
      . . .
   )
   port map(
      port_association;
      port_association;
      . . .
   );
```

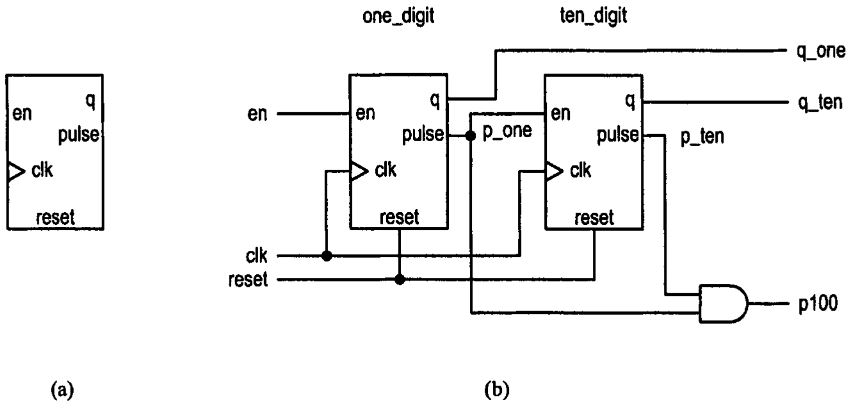

Figure 13.2 Block diagram of a two-digit decimal counter.

In the first line, `component_name` specifies the component to be used, and `instance_label` assigns the instance with a unique label for identification. The **generic map** portion assigns the actual values to the generics. This portion, which is optional, is discussed in Section 13.3. Note that there is no semicolon after the generic map portion. The **port map** portion specifies the connections (i.e., "association") between the component's ports (known as *formal signals*) and the external signals (known as *actual signals*). The `port_association` term has the general format

```
port_name => signal_name
```

This is known as the *named association*.

The use of component instantiation can best be explained by an example. Assume that we want to implement a two-digit decimal counter, which counts up in BCD format (i.e., from 00 to 99) and wraps around. One possible implementation is to cascade two decade counters. The diagram is shown in Figure 13.2(b). The left decade counter represents the digit in the one's place. Its `pulse` port is connected to an external wire, which is labeled as the `p_one` signal, which in turn is connected to the `en` port of the right decade counter, which represents the digit in the ten's place. If the two-digit decimal counter is enabled, the left decade counter asserts `p_one` signal every 10 clock cycles and wraps around. The right decade counter is controlled by the `p_one` signal and thus counts only once for every 10 clock cycles. The `p100` output is a pulse to indicate that the two-digit decimal counter reaches 99 and is ready to wrap around.

To describe this diagram in VHDL, we need to create two instances of the `dec_counter` component and specify the relevant I/O connections. The main task in component instantiation is to specify the mapping between the formal signals and the actual signals. This is a tedious and error-prone task. The best way to do it is to draw a properly labeled block diagram and then derive the VHDL code following the connections of the diagram. The diagram should contain necessary information, which includes the component names, instance labels, and properly labeled ports and connection signals. The block diagram in Figure 13.2(b) is created for this purpose. In our convention, the information from the component declaration, which includes the component name and port names (i.e., the formal signals), is placed inside the block. The external signal names (i.e., actual signals) and instance names, on the other hand, are placed outside the block. Note that a formal signal name and an actual signal name can be the same.

Following the diagram, we can derive the code segments for the two instances:

```
one_digit: dec_counter
   port map (clk=>clk, reset=>reset, en=>en,
             pulse=>p_one, q=>q_one);
ten_digit: dec_counter
   port map (clk=>clk, reset=>reset, en=>p_one,
             pulse=>p_ten, q=>q_ten);
```

The port mapping used here is known as the *named association* because both the name of the formal signal and the name of the actual signal are listed in each port association. The order of the port associations does not matter. For example, the first instance can also be written as

```
one_digit: dec_counter
   port map (pulse=>p_one, reset=>reset, en=>en,
             q=>q_one, clk=>clk);
```

The complete VHDL code is shown in Listing 13.2.

Listing 13.2 Two-digit decimal counter using decade counters

```
library ieee;
use ieee.std_logic_1164.all;
entity hundred_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q_ten, q_one: out std_logic_vector(3 downto 0);
      p100: out std_logic
   );
end hundred_counter;

architecture vhdl_87_arch of hundred_counter is
   component dec_counter
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(3 downto 0);
         pulse: out std_logic
      );
   end component;
   signal p_one, p_ten: std_logic;
begin
   one_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>en,
                pulse=>p_one, q=>q_one);
   ten_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>p_ten, q=>q_ten);
   p100 <= p_one and p_ten;
end vhdl_87_arch;
```

In VHDL, component instantiation is just another concurrent statement and thus can be mixed with other statements. For example, a simple signal assignment statement is used to derive the p100 signal.

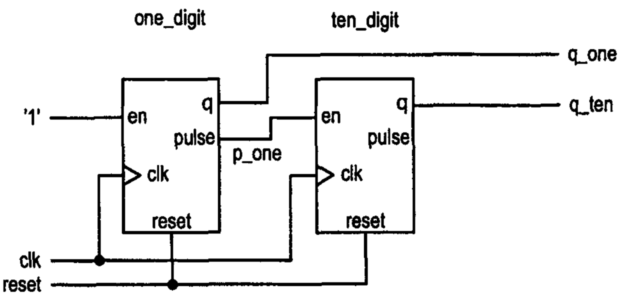

Figure 13.3 Block diagram of a free-running two-digit decimal counter.

13.2.3 Caveats in component instantiation

As long as we derive a proper block diagram, the use of components is straightforward. There are two caveats about component instantiation. One is the use of position association in port mapping, and the other is the handling of unused ports.

So far, we have used the *named association* method for port mapping. Alternatively, we can omit the formal signal names and place the actual names according to the positions of the formal signals. This is known as *positional association*. For example, the component declaration of `dec_counter` shows that the order of the ports is

```
    clk, reset, en, q, pulse
```

In the previous `vhdl_87_arch` architecture, we can put the actual signals in this order in component instantiation. The VHDL code becomes

```
one_digit: dec_counter
   port map (clk, reset, en, q_one, p_one);
ten_digit: dec_counter
   port map (clk, reset, p_one, q_ten, p_ten);
```

At first glance this method may seem to be more compact, but it can cause problems in the long run, especially for a component with many I/O ports. For example, we may revise the port declaration of `dec_counter` later and switch the order of the `clk` and `reset` signals:

```
   . . .
port(
   reset, clk: in std_logic;
   . . .
```

The modification has no effect for the `dec_counter` code in Listing 13.1 but introduces a serious problem for code that instantiates `dec_counter` with positional association. To make the code more reliable, it is good practice to use named association in port and generic mapping.

When we instantiate a component, some ports may not be needed to connect to actual signals. For example, assume that we wish to design a free-running two-digit decimal counter, in which the `en` and `p100` signals are removed. The modified block diagram is shown in Figure 13.3, in which the `en` signal of the `one_digit` instance is tied to logic '1', and the `pulse` signal of the `ten_digit` instance is left unconnected. To describe the mapping of `en`, we can simply use a constant expression to replace the actual signal and the association becomes `en=>'1'`. Some synthesis software may not accept the constant expression. To overcome this, we can create a signal, assign it with the desired constant and then use it as the actual signal in port mapping.

To specify the unused port, we can associate the port with the **open** keyword and the association becomes pulse=>**open**. Good synthesis software should know that the port is not used, backtrack the corresponding circuit and remove the unneeded circuit from implementation. The **open** keyword can also be associated with an input port, and the association means that the initial value in port declaration will be used for the port. Since it is not good practice to assign an initial value to a signal or port in synthesis, this should be avoided.

The VHDL code for the free-running two-digit decimal counter is shown in Listing 13.3.

Listing 13.3 Free-running two-digit decimal counter

```
library ieee;
use ieee.std_logic_1164.all;
entity free_run_hundred_counter is
   port(
      clk, reset: in std_logic;
      q_ten, q_one: out std_logic_vector(3 downto 0)
   );
end free_run_hundred_counter;

architecture vhdl_87_arch of free_run_hundred_counter is
   component dec_counter
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(3 downto 0);
         pulse: out std_logic
      );
   end component;
   signal p_one: std_logic;
begin
   one_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>'1',
                pulse=>p_one, q=>q_one);
   ten_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>open, q=>q_ten);
end vhdl_87_arch;
```

In named association, a formal port can be omitted in the list and VHDL assumes that it is mapped to **open** by default. To make the code reliable, it is good practice to list all ports in port map and explicitly associate the unused output ports with **open**.

13.3 GENERICS

The *generic* construct of VHDL is a mechanism to pass information into an entity and a component. Generics are like parameters. They are first declared in entity and component declaration and later assigned a value during component instantiation.

The use of generics starts with the entity declaration by adding a generic declaration section. The simplified syntax is

```
entity entity_name is
   generic(
      generic_names: data_type;
      generic_names: data_type;
      . . .
   );
   port(
      port_names: mode data_type;
      ...
   );
end entity_name;
```

Once a generic is declared, it can be used in subsequent port declarations and associated architecture bodies. For example, consider the free-running binary counter of Section 8.5.4, which has a fixed width of 4 bits. We can modify it to a more versatile parameterized free-running binary counter by defining a WIDTH generic to specify the desired width (i.e., number of bits). The modified entity declaration becomes

```
entity para_binary_counter  is
   generic(WIDTH: natural);
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(WIDTH-1 downto 0)
   );
end para_binary_counter;
```

Note that the range of the q output is not fixed, but is expressed in terms of the WIDTH generic, as in std_lgic_vector(WIDTH-1 **downto** 0).

After the declaration, the generic can be used in the associated architecture bodies. A generic cannot be modified inside the architecture body and thus functions like a constant. It is sometimes referred to as a *generic constant*. As a constant, we use uppercase letters for the generics in the book.

The corresponding architecture body of the binary counter is

```
architecture arch of para_binary_counter is
   signal r_reg, r_next: unsigned(WIDTH-1 downto 0);
begin
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   r_next <= r_reg + 1;
   q <= std_logic_vector(r_reg);
end arch;
```

Again, note that the WIDTH generic is used to specify the range of internal signals.

To use the parameterized free-running binary counter in a hierarchical design, a similar component declaration should be included in the architecture declaration. The generic can then be assigned a value in the generic mapping section when a component instance is instantiated. An example code is shown in Listing 13.4. The code creates a 4-bit counter and a 12-bit counter.

Listing 13.4 Example of the use of generics

```
library ieee;
use ieee.std_logic_1164.all;
entity generic_demo is
   port(
      clk, reset: in std_logic;
      q_4: out std_logic_vector(3 downto 0);
      q_12: out std_logic_vector(11 downto 0)
   );
end generic_demo;

architecture vhdl_87_arch of generic_demo is
   component para_binary_counter
      generic(WIDTH: natural);
      port(
         clk, reset: in std_logic;
         q: out std_logic_vector(WIDTH-1 downto 0)
      );
   end component;
begin
   four_bit: para_binary_counter
      generic map (WIDTH=>4)
      port map (clk=>clk, reset=>reset, q=>q_4);
   twe_bit: para_binary_counter
      generic map (WIDTH=>12)
      port map (clk=>clk, reset=>reset, q=>q_12);
end vhdl_87_arch;
```

In the second example, we consider the design of a parameterized mod-n counter, in which n can be specified as a parameter. The counter counts from 0 to $n-1$ and then wraps around. To count to n patterns, the counter needs at least $\lceil \log_2 n \rceil$ bits. Our first design uses two generics, the N generic for n and the WIDTH generic for the number of bits in the counter. The VHDL code is shown in Listing 13.5. It is patterned after the decade counter of Listing 13.1 and includes an en control signal and a pulse output signal, which is asserted when the counter reaches $n-1$.

Listing 13.5 Parameterized mod-n counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod_n_counter is
   generic(
      N: natural;
      WIDTH: natural
   );
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(WIDTH-1 downto 0);
      pulse: out std_logic
   );
end mod_n_counter;
```

```
architecture arch of mod_n_counter is
   signal r_reg: unsigned(WIDTH-1 downto 0);
   signal r_next: unsigned(WIDTH-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
         if r_reg=(N-1) then
            r_next <= (others=>'0');
         else
            r_next <= r_reg + 1;
         end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
   pulse <= '1' when r_reg=(N-1) else
            '0';
end arch;
```

Note that WIDTH is not an independent parameter. It can be derived from N and thus is not actually required. Section 13.5 shows how to achieve this.

We can redesign the two-digit decimal counter of Section 13.2.2 by replacing the two decade counters with two parameterized mod-n counters. To do this, we simply assign 10 to N and 4 to WIDTH in component instantiation and thus customize the mod-n counter as a mod-10 counter. The corresponding VHDL code is shown in Listing 13.6.

Listing 13.6 Two-digit decimal counter using parameterized mod-n counters

```
architecture generic_arch of hundred_counter is
   component mod_n_counter
      generic(
         N: natural;
         WIDTH: natural
      );
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(WIDTH-1 downto 0);
         pulse: out std_logic
      );
   end component;
   signal p_one, p_ten: std_logic;
begin
   one_digit: mod_n_counter
      generic map (N=>10, WIDTH=>4)
      port map (clk=>clk, reset=>reset, en=>en,
                pulse=>p_one, q=>q_one);
   ten_digit: mod_n_counter
      generic map (N=>10, WIDTH=>4)
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>p_ten, q=>q_ten);
   p100 <= p_one and p_ten;
end generic_arch;
```

The two examples show the potential of combining the generics and component. Instead of creating an array of counters with different widths, we can use a single parameterized module and customize it to the desired width. This makes the module more flexible and more versatile, and greatly enhances its chance to be reused. The next two chapters provide comprehensive coverage of the design of parameterized modules.

Another major application of generics is to pass delay information in modeling and simulation. For example, we can define the Tpd generic for the propagation delay. It can then be used in a statement like

```
y <= a + b after Tpd ns;
```

This allows us to pass delay information into the model when it becomes available.

13.4 CONFIGURATION

13.4.1 Introduction

When a component is declared and instantiated, as the example in Listing 13.2, only basic generic and port information is provided. *Configuration* of VHDL is the process of *binding* a component with a design entity and architecture. The process includes two parts:

1. Bind a component with a design entity.
2. Bind the design entity with an architecture body.

The configuration of VHDL is very flexible, and thus its detailed syntax and use are quite complex. However, most features are not needed in RT-level design and synthesis. The IEEE RTL synthesis standard supports only the second part of the process, the binding of entity and architecture. We discuss only default binding, and top-level entity and architectural body binding in the book.

Explicit configuration is not always required for a component. For example, the codes in previous sections use no configuration constructs. In this case, the component is bound by the *default binding*, which is processed as follows:

- The component is bound to an entity with the identical name.
- Component ports are bound to entity ports of the same names.
- The most recently analyzed architecture body is bound to the entity declaration.

In RT-level design, the hierarchy is normally simple, and only one architecture body exists. The default binding should be satisfactory most of the time, and no explicit configuration statement is needed.

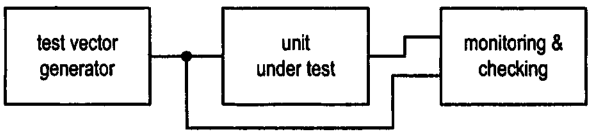

Figure 13.4 Block diagram of a testbench.

In synthesis, multiple architectures may be needed for several reasons. First, there is frequently a trade-off between area and performance in a digital circuit. A complex circuit, such as a multiplier, may have several implementations, each with a unique area–delay characteristics. Each implementation represents an architecture body. Second, we sometimes need to adjust certain circuit characteristics to fit a specific application. For example, we may need to force a counter to circulate different patterns. One way to accomplish this is to use a separate architecture body for each pattern.

In modeling and simulation, having multiple architectures is more common. For example, we discussed the concept of testbench in Section 2.2.4. The diagram of a basic testbench is shown in Figure 13.4. A complex design is normally first specified in an abstract behavioral description, converted to an RT-level description, and then synthesized to a cell-level structural description. Each description represents an architecture body. As the design progresses, a more detailed architecture body becomes available. We can use configuration to bind the new description for verification.

There are two ways to specify the configuration. It can be described in an independent design unit, which is known as *configuration declaration*, or included in the declaration section of the architecture body, which is known as *configuration specification*. The two methods are discussed in the following subsections. The IEEE RTL standard supports only the configuration declaration method.

13.4.2 Configuration declaration

In the configuration declaration method, we create a new kind of design unit, known as *configuration*, to specify the binding of a component. A configuration unit is an independent design unit in VHDL, just like an entity declaration and an architecture body. It is analyzed and stored independently when the VHDL code is processed. The simplified syntax of a configuration unit is

```
configuration conf_name of entity_name is
  for archiecture_name
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
    . . .
  end for;
end;
```

The conf_name term is the unique identifier for this configuration unit. The entity_name and architecture_name terms identify the entity and the architecture for which the configuration is intended. The instance_label term specifies a specific component instance, and the following "**use** . . ." clause indicates the entity and architecture to be bound to the instance. The lib_name term is the name of the library in which the entity and architecture reside. The library is discussed in Section 13.5. In the place of instance_label, we can use the **all** keyword to represent all instances of this particular component, or use **others** in the end to represent all the unbound instances of the component.

To demonstrate the use of configuration, we create a second architecture, down_arch, for the dec_counter entity of Section 13.2.1. The VHDL code is shown in Listing 13.7. It counts down from 9 to 0 and then wraps around. The pulse output is asserted when the counter reaches 0 and is ready to wrap around. If we use this architecture in the vhdl_87_arch architecture of the two-digit decimal counter of Section 13.2.2, the two-digit counter counts down from 99 to 00 and then wraps around.

Listing 13.7 Decade counter with a count-down sequence

```
architecture down_arch of dec_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   constant TEN: integer := 10;
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
         if r_reg=0 then
            r_next <= to_unsigned(TEN-1,4);
         else
            r_next <= r_reg - 1;
         end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
   pulse <= '1' when r_reg=0 else
            '0';
end down_arch;
```

Depending on the requirement of the direction of a two-digit decimal counter, we can create a configuration unit to specify the desired binding. The VHDL in Listing 13.8 binds the two instances with the down_arch architecture.

Listing 13.8 Configuration for a two-digit decimal counter

```
configuration count_down_config of hundred_counter is
   for vhdl_87_arch
      for one_digit: dec_counter
         use entity work.dec_counter(down_arch);
      end for;
      for ten_digit: dec_counter
         use entity work.dec_counter(down_arch);
      end for;
   end for;
end;
```

Note that the work library is the default library used in VHDL. It represents the current working library.

13.4.3 Configuration specification

The configuration declaration is general and flexible. However, for a simple design, creating a new design unit for this purpose is somewhat cumbersome. An alternative is to specify the relevant configuration in the declaration section of the architecture body. This is known as a *configuration specification*. The simplified syntax is

```
for instance_label: component_name
   use entity lib_name.bound_entity_name(bound_arch_name);
for instance_label: component_name
   use entity lib_name.bound_entity_name(bound_arch_name);
. . .
```

For example, the configuration declaration in Listing 13.8 can also be specified by a configuration specification. We simply revise the vhdl_87_arch by adding the relevant configuration information to the declaration section:

```
architecture vhdl_87_config_arch of hundred_counter is
   component dec_counter
      port(
        clk, reset: in std_logic;
        en: in std_logic;
        q: out std_logic_vector(3 downto 0);
        pulse: out std_logic
      );
   end component;
   for one_digit: dec_counter
      use entity work.dec_counter(down_arch);
   for ten_digit: dec_counter
      use entity work.dec_counter(down_arch);
   signal p_one, p_ten: std_logic;
begin
   . . .
```

13.4.4 Component instantiation and configuration in VHDL 93

Components and configuration are flexible in VHDL, but its syntax is involved and tedious. Since RT-level design uses relatively simple component instantiation and binding, the syntactic constructs becomes cumbersome. For example, consider the previous vhdl_87_config_arch architecture. We need a relatively lengthy declaration to use and bind the two component instances in design. VHDL 93 provides a much simpler mechanism. It allows a component to be bound directly to an entity and an architecture in component instantiation. The simplified syntax is

```
instance_label:
   entity lib_name.bound_entity_name(bound_arch_name)
      generic map (. . .)
      port map (. . .);
```

The **entity** lib_name.bound_entity_name(bound_arch_name) clause specifies the associated entity and architecture, and no component declaration or any additional configuration construct is needed. The (bound_arch_name) term is optional. If it is omitted, the most recently analyzed architecture will be bound to the entity.

Consider the two-digit decimal counter of Section 13.2.2. With this mechanism, a more compact code can be derived, as shown in Listing 13.9.

Listing 13.9 Two-digit decimal counter with direct entity binding

```
architecture vhdl_93_arch of hundred_counter is
   signal p_one, p_ten: std_logic;
begin
   one_digit: entity work.dec_counter(up_arch)
      port map (clk=>clk, reset=>reset, en=>en,
                pulse=>p_one, q=>q_one);
   ten_digit: entity work.dec_counter(up_arch)
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>p_ten, q=>q_ten);
   p100 <= p_one and p_ten;
end vhdl_93_arch;
```

Since this kind of instantiation is valid only in VHDL 93, it is not supported by the IEEE RTL synthesis standard. However, some software does accept this type of component instantiation.

13.5 OTHER SUPPORTING CONSTRUCTS FOR A LARGE SYSTEM

13.5.1 Library

As we discussed in Section 3.2.5, a VHDL program is analyzed and stored as individual design units, which include entity declaration, architecture body, configuration declaration, and so on. A VHDL *library* is the virtual repository that stores the analyzed design units. VHDL does not define the physical location of a library. Most software maps a library to a physical directory in a hard disk. By default, the current design units are stored in a working library named work. For example, the work library is used in the previous component instantiation:

```
. . .
one_digit: entity work.dec_counter(up_arch)
. . .
```

For a complex design, there may exist a large number of design units. It is desirable to organize these units and store them in separate places. Also, we may have a collection of commonly used design units that are shared by many different designs. It is more effective to save these units in a common library rather than duplicating them in every design directory.

To access the content of a library, we must first make it known by using a library statement. The syntax is

```
library lib_name, lib_name, ... , lib_name;
```

For example, assume that we create a library named c_lib and save the previous dec_counter entity and relevant architectures in the library. The count_down_config configuration discussed in Section 13.4.2 must be revised accordingly:

```
library c_lib;  -- make c_lib visible
configuration clib_config of hundred_counter is
   for vhdl_87_arch
      for one_digit: dec_counter
         use entity c_lib.dec_counter(down_arch); -- c_lib
      end for;
      for ten_digit: dec_counter
         use entity c_lib.dec_counter(down_arch); -- c_lib
      end for;
   end for;
end;
```

Note that the work library of the original code is replaced with c_lib.

If a design unit is accessed frequently, we can make it visible by adding a **use** clause. The syntax is

```
use lib_name.unit_name;
```

The unit can then be accessed directly without referring to the library. The **all** keyword can be used in place of unit_name to make all units of the library visible. For example, the previous code can be revised as:

```
library c_lab;
use c_lib.dec_counter; -- make dec_counter visible
configuration clib_config of hundred_counter is
   for vhdl_87_arch
      for one_digit: dec_counter
         use entity dec_counter(down_arch); -- lib dropped
      end for;
      for ten_digit: dec_counter
         use entity dec_counter(down_arch);
      end for;
   end for;
end;
```

Note that the library name is dropped from the "**use entity** dec_counter(down_arch);" statement.

The work library is declared implicitly by VHDL definition, and thus there is no need for the "**library** work;" statement.

13.5.2 Subprogram

Subprograms in VHDL include *functions* and *procedures*. Their bodies are made of sequential statements, and their behaviors are similar to those in traditional programming languages. Unlike entity and architecture, procedures and functions are not design units and thus cannot be processed independently. For example, we cannot isolate a function from the code and synthesize it separately. Therefore, while the functions and procedures are basic building blocks of software hierarchy, they are not adequate to describe the hardware hierarchy.

VHDL functions are more versatile and useful than procedures, and thus our discussion focuses mainly on functions. In synthesis, functions should not be used to specify the design hierarchy, but should be treated as a shorthand for simple, repeatedly used operations. Functions are also needed to perform certain house-keeping tasks, such as data type conversion or operator overloading in IEEE packages.

A VHDL function takes several parameters and returns a single value. It must first be declared in the declaration section and then can be called later. A function can be thought of as an extension of the expression and can be "called" wherever an expression is used. The simplified syntax of a function is

```
function func_name(parameter_list) return data_type is
   declarations;
begin
   sequential statement;
   sequential statement;
   . . .
   return(expression);
end;
```

The following examples illustrate the construction of a function. The first example is a function that performs a majority function. It returns '1' if two or more input parameters, a, b and c, are '1'. The function can be treated as a shorthand for the $a \cdot b + a \cdot c + b \cdot c$ expression.

```
function maj(a, b, c: std_logic) return std_logic is
   variable result: std_logic;
begin
   result := (a and b) or (a and c) or (b and c);
   return result;
end maj;
```

The maj function must be declared and then can be invoked. The following code segment illustrates its use:

```
architecture arch of . . .
   -- declaration
   function maj(a, b, c: std_logic) return std_logic is
      variable result: std_logic;
   begin
      result := (a and b) or (a and c) or (b and c);
      return result;
   end maj;
   signal i1, i2, i3, i4, x, y: std_logic;
begin
   . . .
```

```
    x <= maj(i1, i2, i3) or i4;
    y <= i1 when maj(i2, i3, i4)='1' else
    . . .
```

Note that the entire function definition is included in the declaration section of the architecture body. This may become cumbersome. An alternative is to declare the function in a package, which is discussed in the next subsection.

The second example is a function that performs data type conversion. It converts the std_logic data type to the boolean data type.

```
function to_boolean(a: std_logic) return boolean is
   variable result: boolean;
begin
   if a='1' then
      result := true;
   else
      result := false;
   end if;
   return result;
end to_boolean;
```

If this function is declared, we can use to_boolean(a) to replace the a='1' expression.

The last example is a function that performs $\lceil \log_2 n \rceil$, which is frequently needed in calculating the width of data signals.

```
function log2c(n: integer) return integer is
   variable m, p: integer;
begin
   m := 0;
   p := 1;
   while p < n loop
      m := m + 1;
      p := p * 2;
   end loop;
   return m;
end log2c;
```

13.5.3 Package

As a system becomes complex, more information is included in the declaration section. The declaration section of an architecture body may consist of the declarations of constants, data types, components, functions and so on. When a system is divided into several smaller subsystems, some declarations must be duplicated in many different design units. The VHDL *package* construct is a method of organizing declarations. We can gather the commonly used declarations in a design, group them together and store them in a package. A design unit just needs to include a use clause to access these declarations.

A VHDL package is divided into *package declaration* and *package body*. The declaration items are placed in a package declaration. If an item is a subprogram, only the declaration of the subprogram is included. The body (i.e., the implementation) of the subprogram is placed in the associated package body. The package body is optional and is needed only when subprograms exist.

Package declaration and package body are design units of VHDL. They are analyzed independently and stored in a library. To make a declaration item visible, a use clause is needed. Its syntax is

```
use lib_name.package_name.item_name;
```

Most of the time, we use the **all** keyword in place of `item_name` to make all items of the named package visible.

Many extensions to VHDL are done by defining additional packages, such as the IEEE `std_logic_1164` and `numeric_std` packages. Almost all of our VHDL codes include the statement

```
use ieee.std_logic_1164.all;
```

It makes all of the declaration items of the predefined `std_logic_1164` package visible, and thus we can use the `std_logic` and `std_logic_vector` data types in VHDL code.

We can also define our own package. The syntax of a package declaration is very simple:

```
package package_name is
   declaration item;
   declaration item;
   . . .
end package_name;
```

If the declaration items include subprograms, an associated package body is needed. Its syntax is

```
package body package_name is
   subprogram;
   subprogram;
   . . .
end package_name;
```

An example of package declaration is shown in the first part of Listing 13.10. It consists of the definition of the `std_logic_2d` data type, which is a two-dimensional array with element of `std_logic` data type, and the declaration of the `log2c` function. Note that the package also invokes the IEEE `std_logic_1164` package so that the `std_logic` data type can be used. The corresponding package body is shown in the second part of Listing 13.10, which is the implementation of the `log2c` function.

Listing 13.10 Example of a package

```
-- package declaration
library ieee;
use ieee.std_logic_1164.all;
package util_pkg is
   type std_logic_2d is
      array(integer range <>, integer range <>) of std_logic;
   function log2c (n: integer) return integer;
end util_pkg ;

--package body
package body util_pkg is
   function log2c(n: integer) return integer is
      variable m, p: integer;
   begin
```

```
      m := 0;
      p := 1;
      while p < n loop
         m := m + 1;
         p := p * 2;
      end loop;
      return m;
   end log2c;
end util_pkg;
```

For the parameterized mod-n counter of Section 13.3, one drawback is that we must use a redundant WIDTH generic to specify the width of the output signal. To overcome the problem, we need a previously defined function to calculate WIDTH from N. The log2c function of the util_pkg package can be used for this purpose. We can invoke this package before the entity declaration. Assume that the package is saved in the same working directory. The improved code is shown in Listing 13.11.

Listing 13.11 Improved parameterized mod-n counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity better_mod_n_counter is
   generic(N: natural);
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(log2c(N)-1 downto 0);
      pulse: out std_logic
   );
end better_mod_n_counter;

architecture arch of better_mod_n_counter is
   constant WIDTH: natural := log2c(N);
   signal r_reg: unsigned(WIDTH-1 downto 0);
   signal r_next: unsigned(WIDTH-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
          if r_reg=(N-1) then
             r_next <= (others=>'0');
```

```
            else
               r_next <= r_reg + 1;
            end if;
         end if;
      end process;
      -- output logic
      q <= std_logic_vector(r_reg);
      pulse <= '1' when r_reg=(N-1) else
               '0';
   end arch;
```

We add the statement

```
use work.util_pkg.all;
```

to make the log2c function visible. The function can then be used in the range specification for the q output, as in `std_logic_vector(log2c(N)-1 downto 0)`, as well as the calculation of the internal WIDTH constant.

13.6 PARTITION

VHDL provides powerful mechanisms and versatile language constructs to support hierarchical design methodology and to manage the design of large systems. To apply these features, we must first determine the design hierarchy and divide the system into smaller parts. The process is sometimes known as *design partition*.

For synthesis, the design partition can be viewed from two perspectives: physical partition and logical partition. *Physical partition* is the division of the physical implementation. It specifies how the circuit is divided during synthesis. *Logical partition* is imposed by human designers. The goal is to make the design, development and verification of a system manageable. The two kinds of partitions are correlated but not necessarily identical.

13.6.1 Physical partition

A digital system can be described by a hierarchy of an arbitrary number of levels. The circuit parts becomes simpler as we traverse down the hierarchy. In VHDL code, the circuit parts are described as component instances. When the code is processed, the components are replaced by the actual architecture bodies level by level, and the hierarchy is gradually converted to a flattened description. One way to perform synthesis is to collapse the entire hierarchy into a one big, flattened circuit and then to synthesize the circuit accordingly. More sophisticated software provides mechanisms to selectively flatten the hierarchy and preserve some high-level components. The software will synthesize and optimize these components separately and then merge the resulting netlists (i.e., the cell level descriptions) to form the final circuit.

An important issue is the size of the preserved components. Since synthesis involves many sophisticated algorithms, the required computation time and memory space are normally much worse than the linear order, $O(n)$, where n is the size of the circuit. This implies that synthesizing a large circuit will take much more computation time and memory space than that required by several smaller circuits. For example, assume that an algorithm is on the order of $O(n^3)$. If it requires 1 second to synthesize a 1000-gate circuit, it will take 125 seconds (i.e., 5^3 seconds) to synthesize a 5000-gate circuit and 35 hours (i.e., 50^3 seconds) to synthesize a 50,000-gate circuit. However, if we break the 50,000-gate circuit into ten 5000-gate circuits, it requires only about 21 minutes (i.e., $10 * 5^3$ seconds) to synthesize the ten small parts.

On the other hand, when we preserve a component, it implicitly forms a "synthesis boundary," and optimization can only be performed within the boundary. This prevents synthesis software from exploring optimization opportunities that exist between components. Thus, small component size hinders the optimization process and leads to less efficient overall implementation. Many software tools suggest that the maximal gate count for synthesis is between 5000 and 50,000 gates.

13.6.2 Logical partition

The logical partition is determined by human designers. A good partition simplifies and streamlines the development and verification process, and makes the code reliable and portable. To facilitate the synthesis, a "logical circuit part" in the hierarchy should be within the range of the maximal gate count recommended by the synthesis software. On the other hand, since synthesis software can flatten and collapse a part of the hierarchy, smaller "logical parts" can be used in the design hierarchy.

In addition to synthesis concerns, partition should also be used to help develop reliable design, and portable and reusable code. We should pay particular attention to the circuits that may hinder portability or introduce problems in development flow. It is a good idea to separate these circuits from ordinary logic and instantiate them as components in a design hierarchy. Two types of circuits of concern are:

- Device-dependent circuits
- Non-Boolean circuits

Device-dependent circuits Device-dependent circuits are those not synthesized by generic logic gates. They are predesigned or even prefabricated for a specific device technology. For example, most device technology has various types of prefabricated memory modules. These circuits are inferred by component instantiation and require no synthesis. Once a device-dependent circuit is used, the VHDL code becomes device dependent. To maintain portability, one way is to isolate these circuits in the top-level hierarchy and instantiate them as individual components. If the VHDL code is used later for a different device technology, we need only substitute these components with equivalent circuits of the new technology and keep the remaining code intact.

Non-Boolean circuits Digital system design is primarily based on a mathematical model of Boolean algebra and its derivations. The algorithms in analysis, synthesis, verification and testing are developed within this framework. If a circuit does not follow the basic mathematical model, we call it a *non-Boolean* circuit. Some examples are listed below.

- *Tri-state buffer.* It has a third possible value, high impedance, in its output. The high impedance cannot be optimized or propagated as regular logic values.
- *Delay-sensitive circuit.* It uses logic gates to introduce a specific amount of propagation delay. The function of the circuit relies on the delay characteristics, not on Boolean algebra manipulation.
- *Clock distribution circuit.* It distributes the clock signal to the connected FFs. The circuit functions as a current amplifier and performs no logic operation.
- *Synchronization circuit.* It uses FFs to resolve the metastable condition, not for regular storage. This is discussed in Chapter 16.

These circuits should be isolated in the hierarchy so that later they can be processed independently.

13.7 SYNTHESIS GUIDELINES

- Use components, not subprograms, to specify the design hierarchy.
- Use the `std_logic` and `std_logic_vector` data types in the ports of components to maintain portability.
- Use named association, not positional association, in port mapping and generic mapping.
- List all ports of a component in port mapping and use **open** for unused output ports.
- For synthesis, partition the system into 5000- to 50,000-gate modules. Collapse and flatten low-level hierarchy if the components are too small.
- Separate device-dependent parts from ordinary logic and instantiate them as components in a hierarchy.
- Separate non-Boolean circuits from ordinary logic and instantiate them as components in a hierarchy.

13.8 BIBLIOGRAPHIC NOTES

This chapter provides a detailed discussion of VHDL components and gives a brief review of many other language constructs. The review is aimed primarily at synthesis of the RT-level system. Comprehensive coverage and the syntax of these constructs can be found in VHDL texts, such as *The Designer's Guide to VHDL, 2nd edition*, by P. J. Ashenden.

Problems

13.1 Consider a three-digit decimal counter that counts from 000 to 999 and wraps around. Use the `dec_counter` of Section 13.2.1 as a component to design this circuit.

(a) Derive the block diagram and properly label the formal and actual signals.
(b) Follow the block diagram to derive the VHDL code.
(c) Use a configuration specification for configuration.
(d) Same as part (c), but use a configuration declaration for configuration.

13.2 Redesign the three-digit decimal counter of Problem 13.1 using the mod-n counter of Section 13.3.

(a) Derive the block diagram and properly label the formal and actual signals.
(b) Follow the block diagram to derive the VHDL code.

13.3 We want to design a timer that counts from 00 to 59 seconds and then wraps around. Assume that the system clock is 1 MHz. Use the mod-n counter of Section 13.3 as a component to design this circuit.

(a) Derive the block diagram and properly label the formal and actual signals.
(b) Follow the block diagram to derive the VHDL code.

13.4 Consider a counter that counts from m to n and then wraps around. Derive VHDL code for the counter. Use generics, `M` and `N`, for m and n of the counter.

13.5 Divide the FIFO control circuit in Figure 9.14 into a hierarchy of two counters and two comparison circuits.

(a) Derive VHDL entities and architectures for the counter and comparison circuits.

(b) Follow the diagram in Figure 9.14 to derive the VHDL code.

13.6 Some synthesis software does not accept the ** operator. Derive a VHDL function, `power2`, that implements the function $f(n) = 2^n$.

13.7 Derive a function that converts the `boolean` data type into the `std_logic` data type. The `true` and `false` values of the `boolean` type are converted to `'1'` and `'0'` of the `std_logic` data type respectively.