CHAPTER 2

OVERVIEW OF HARDWARE DESCRIPTION LANGUAGES

A digital system can be described at different levels of abstractions and from different points of view. As the design process progresses, the level and view are changed, either by human designers or by software tools. It is desirable to have a common framework to exchange information among the designers and various software tools. *Hardware description languages* (*HDLs*) serve this purpose. In this chapter we provide an overview of the design, use and capability of HDLs. The basic concept and essential modeling features are introduced by a series of codes to show the "big picture" of HDLs. The detailed syntax, language constructs and associated semantics are discussed in subsequent chapters.

2.1 HARDWARE DESCRIPTION LANGUAGES

A digital system can be described at different levels of abstraction and from different points of view. An HDL should faithfully and accurately model and describe a circuit, whether already built or under development, from either the structural or behavioral views, at the desired level of abstraction. Because HDLs are modeled after hardware, their semantics and use are very different from those of traditional programming languages. The following subsections discuss the need, use and design of an HDL.

2.1.1 Limitations of traditional programming languages

There are wide varieties of computer programming languages, from Fortran to C to Java. Unfortunately, they are not adequate to model digital hardware. To understand their limita-

tions, it is beneficial to examine the development of a language. A programming language is characterized by its syntax and semantics. The *syntax* comprises the grammatical rules used to write a program, and the *semantics* is the "meaning" associated with language constructs. When a new computer language is developed, the designers first study the characteristics of the underlying processes and then develop syntactic constructs and their associated semantics to model and express these characteristics.

Most traditional general-purpose programming languages, such as C, are modeled after a sequential process. In this process, operations are performed in sequential order, one operation at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be altered at will. The sequential process model has two major benefits. At the abstract level, it helps the human thinking process to develop an algorithm step by step. At the implementation level, the sequential process resembles the operation of a basic computer model and thus allows efficient translation from an algorithm to machine instructions.

The characteristics of digital hardware, on the other hand, are very different from those of the sequential model. A typical digital system is normally built by smaller parts, with customized wiring that connects the input and output ports of these parts. When a signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly. These operations are performed concurrently, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete. After completion, each part updates the value of the corresponding output port. If the value is changed, the output signal will in turn activate all the connected parts and initiate another round of operations. This description shows several unique characteristics of digital systems, including the connections of parts, concurrent operations, and the concept of propagation delay and timing. The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a need for special languages (i.e., HDLs) that are designed to model digital hardware.

2.1.2 Use of an HDL program

To better understand HDL, it is helpful to examine the use of an HDL program. In a traditional programming language, a program is normally coded to solve a specific problem. It takes certain input values and generates the output accordingly. The program is first compiled to machine instructions and then run on a host computer. On the other hand, the application of an HDL program is very different. The program plays three major roles:

- *Formal documentation.* A digital system normally starts with a word description. Unfortunately, since human language is not precise, the description is frequently incomplete and ambiguous, and the same description may be subject to different interpretations. Because the semantics and syntax of an HDL are defined rigorously, a system specified in an HDL program is explicit and precise. Thus, an HDL program can be used as a formal system specification and documentation among various designers and users.
- *Input to a simulator.* As we discussed in Chapter 1, simulation is used to study and verify the operation of a circuit without constructing the system physically. An HDL simulator provides a framework to model the concurrent operations in a sequential host computer, and has specific knowledge of the language's syntactic constructs and the associated semantics. An HDL program, combined with test vector generation and a data collection code, forms a testbench, which becomes the input to the

HDL simulator. During execution, the simulator interprets HDL code and generates responses accordingly.

- *Input to a synthesizer.* The modern development flow is based on the refinement process, which gradually converts a high-level behavioral description to a low-level structural description. Some refinement steps can be performed by synthesis software. The synthesis software takes an HDL program as its input and realizes the circuit from the components of a given library. The output of the synthesizer is a new HDL program that represents the structural description of the synthesized circuit.

2.1.3 Design of a modern HDL

The fundamental characteristics of a digital circuit are defined by the concepts of entity, connectivity, concurrency and timing. *Entity* is the basic building block, modeling after a part of a real circuit. It is self-contained and independent, and has no implicit information about other entities. *Connectivity* models the connecting wires among the parts. It is the way that entities interact with one another. Since the connections of a system are seldom formed as a single thread, many entities may be active at the same time and many operations are performed in parallel. *Concurrency* describes this type of behavior. *Timing* is related to concurrency. It specifies the initiation and completion of each operation and implicitly provides a schedule and order of multiple operations.

The goal of an HDL is to describe and model digital systems faithfully and accurately. To achieve this, the cornerstone of the language should be based on the model of hardware operation, and its semantics should be able to capture the fundamental characteristics of the circuits.

As we discussed in Chapter 1, a digital system can be described at four different levels of abstraction and from three different points of view. Although these descriptions have similar fundamental characteristics, their detailed representations and models vary significantly. Ideally, we wish to develop a single HDL to cover all the levels and all the views. However, this is hardly feasible because the vast differences between abstraction levels and views will make the language excessively complex. Modern HDLs normally cover descriptions in structural and behavior views, but not in physical view. They provide constructs to support modeling at the gate and RT levels, and to a limited degree, at processor and transistor levels. The highlights of modern HDLs are as follows:

- The language semantics encapsulate the concepts of entity, connectivity, concurrency, and timing.
- The language can effectively incorporate propagation delay and timing information.
- The language consists of constructs that can explicitly express the structural implementation (i.e., a block diagram) of a circuit.
- The language incorporates constructs that can describe the behavior of a circuit, including constructs that resemble the sequential process of traditional languages, to facilitate abstract behavioral description.
- The language can efficiently describe the operations and structures at the gate and RT levels.
- The language consists of constructs to support a hierarchical design process.

2.1.4 VHDL

VHDL and Verilog are the two most widely used HDLs. Although the syntax and "appearance" of the two languages are very different, their capabilities and scopes are quite similar.

Both are industrial standards and are supported by most software tools. VHDL is used in this book since it has better support for parameterized design.

VHDL stands for VHSIC (very high speed integrated circuit) HDL. The development of VHDL was sponsored initially by the US Department of Defense as a hardware documentation standard in the early 1980s and then was transferred to the IEEE (Institute of Electrical and Electronics Engineers). IEEE ratified it as IEEE standard 1076 in 1987, which is referred to as VHDL-87. Each IEEE standard is reviewed every few years and is revised as needed. IEEE revised the VHDL standard in 1993, which is referred to as VHDL-93, and made minor modifications and bug fixes in 2001, which is referred to as VHDL-2001. Since no new language construct is added in the new version, there is no significant difference between VHDL-93 and VHDL-2001. A suffix is sometimes added to the IEEE standard to indicate the year the standard was released. For example, VHDL-87 and VHDL-2001 are known as IEEE standards 1076-1987 and IEEE 1076-2001 respectively.

After the initial release, various extensions were developed to facilitate various design and modeling requirements. These extensions are documented in several IEEE standards:

- IEEE standard 1076.1-1999, *VHDL Analog and Mixed Signal Extensions (VHDL-AMS)*: defines the extension for analog and mixed-signal modeling.
- IEEE standard 1076.2-1996, *VHDL Mathematical Packages*: defines extra mathematical functions for real and complex numbers.
- IEEE standard 1076.3-1997, *Synthesis Packages*: defines arithmetic operations over a collection of bits.
- IEEE standard 1076.4-1995, *VHDL Initiative Towards ASIC Libraries (VITAL)*: defines a mechanism to add detailed timing information to ASIC cells.
- IEEE standard 1076.6-1999, *VHDL Register Transfer Level (RTL) Synthesis*: defines a subset that is suitable for synthesis.
- IEEE standard 1164-1993 *Multivalue Logic System for VHDL Model Interoperability (std_logic_1164)*: defines new data types to model multivalue logic.
- IEEE standard 1029.1-1998, *VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES)*: defines how to use VHDL to exchange information in a simulation environment.

Standards 1076.3, 1076.6 and 1164 are related to synthesis and are discussed in Chapter 3.

2.2 BASIC VHDL CONCEPT VIA AN EXAMPLE

As its name indicates, HDL describes hardware. Thus, it is essential to read or write HDL code from hardware's perspective. A simple example in this section shows the basic modeling concepts used in HDL and demonstrates the semantic differences between HDLs and traditional programming languages. The example is coded in VHDL and the language constructs are mostly self-explanatory. The purpose of the example is to provide a big picture of HDL and VHDL. The detailed syntax and language constructs are studied in subsequent chapters.

The example is a circuit that detects even parity. There are one output, `even`, and three inputs, `a(2)`, `a(1)` and `a(0)`, which are grouped as a bus. The output is asserted when there are even numbers (i.e., 0 or 2) of 1's from the inputs. The truth table of this circuit is shown in Table 2.1.

The VHDL codes for a general description, pure structural description, pure behavioral description and testbench are discussed in the following subsections.

Table 2.1 Truth table of an even-parity detector circuit

a(2)	a(1)	a(0)	even
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

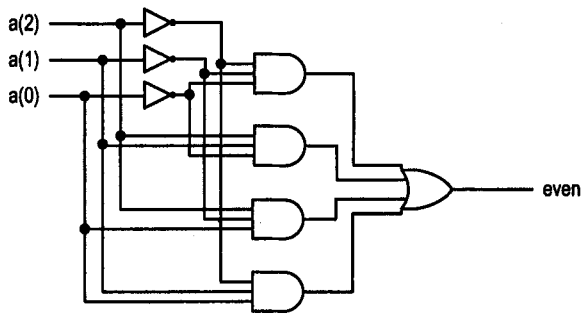


Figure 2.1 Two-level and–or implementation of an even-parity detector circuit.

2.2.1 General description

From Boolean algebra, we know that each row of a truth table represents a product term and the output can be written as the sum-of-products expression

$$even = a(2)' \cdot a(1)' \cdot a(0)' + a(2)' \cdot a(1) \cdot a(0) + a(2) \cdot a(1)' \cdot a(0) + a(2) \cdot a(1) \cdot a(0)'$$

The expression can be realized by a two-level and–or circuit, as shown in Figure 2.1.

The first VHDL description is based on this expression and the code is shown in Listing 2.1. In this book, the reserved words are in boldface font, as in **library**, and comments are in italic font, as in *-- this is a comment.*

Listing 2.1 Even-parity detector based on a sum-of-products expression

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

-- entity declaration
entity even_detector is
    port(
       a: in std_logic_vector(2 downto 0);
       even: out std_logic
    );
end even_detector;
```

```
-- architecture body
architecture sop_arch of even_detector is
   signal p1, p2, p3, p4 : std_logic;
begin
   even <= (p1 or p2) or (p3 or p4) after 20 ns;
   p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
   p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
   p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
   p4 <= a(2) and a(1) and (not a(0)) after 12 ns;
end sop_arch ;
```

The code consists of two major units: entity declaration and architecture body. The *entity declaration* is:

```
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;
```

It specifies the input and output ports of this circuit. There are one output port, even, and one input port, a, which is a three-element array, representing a(2), a(1) and a(0).

The *architecture body* specifies the internal operation or organization of a circuit. The first line of the architecture body shows the name of the body, sop_arch (for sum-of-products architecture), and the corresponding entity, even_detector:

```
architecture sop_arch of even_detector is
```

The next line is the signal declaration:

```
signal p1, p2, p3, p4: std_logic;
```

The p1, p2, p3 and p4 signals here can be interpreted as wires that connect the internal parts. The declaration is visible inside this architecture.

The actual architectural description is encompassed within **begin** and **end** sop_arch:

```
even <= (p1 or p2) or (p3 or p4) after 20 ns;
p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
p4 <= a(2) and a(1) and (not a(0)) after 12 ns;
```

The fundamental building block inside the architecture body is a concurrent statement. For example, the first line is a *concurrent statement*:

```
even <= (p1 or p2) or (p3 or p4) after 20 ns;
```

A concurrent statement can be thought of as a circuit part. The left-hand-side signal or port is the output, and all the signals and ports appearing in the right-hand-side expression are the input signals. The right-hand-side expression can be considered as the operation performed by this circuit. The result is available after a specific amount of propagation delay, which is specified by the **after** clause. This particular concurrent statement can be interpreted as a circuit with inputs, p1, p2, p3 and p4, and with an output, even. It performs the or operation among the four inputs, and the operation takes 20 ns. The other four statements can be interpreted in a similar fashion.

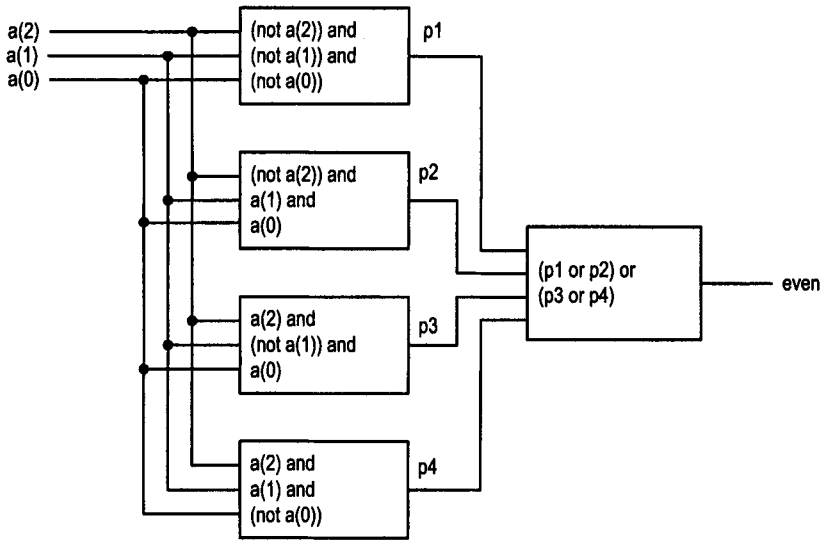


Figure 2.2 Conceptual diagram of `sop_arch` architecture.

This architecture body consists of five concurrent statements, which can be interpreted as a collection of five circuit parts. These concurrent statements are linked through common signals (or *nets*). When a signal appears on both the right- and left-hand sides, it implies that there is a wire connecting the two parts. Thus, a larger circuit is constructed implicitly through these connections. The conceptual diagram described by this code is shown in Figure 2.2.

Note that since each concurrent statement represents a circuit part and its interconnection, the order of these concurrent statements does not matter. For example, we can rearrange the code as

```
p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
even <= (p1 or p2) or (p3 or p4) after 20 ns;
p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
p4 <= a(2) and a(1) and (not a(0)) after 12 ns;
```

Unlike sequential execution of statements in traditional programming language, concurrent statements are independent and can be activated in parallel. When a concurrent statement's input changes, it is "awakened" and evaluates the expression accordingly. The result will be available after the specific propagation delay, and the new value will be assigned to the output signal. The change of output signals, in turn, may activate other statements and invoke a new round of evaluations.

The incorporation of propagation delay with each concurrent statement is the key ingredient to model the operation of hardware and to ensure the proper interpretation of VHDL code. Sometimes the after clause is omitted because the delay information is not available, as in

```
even <= (p1 or p2) or (p3 or p4);
```

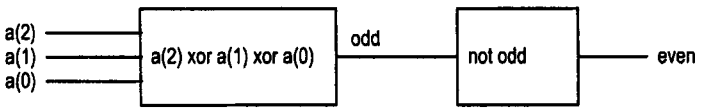


Figure 2.3 Conceptual diagram of the `xor_arch` architecture body.

In this case, VHDL semantics specifies that there is an implicit δ-delay (delta delay) associated with the operation. A *δ-delay* is an infinitesimal delay that is greater than zero but smaller than any physical number. The previous line can be interpreted as

```
even <= (p1 or p2) or (p3 or p4) after δ;
```

Thus, regardless whether there is an after clause, there is always a propagation delay associated with a concurrent statement.

The truth table is just one method to realize the even-parity detector circuit. An alternative is to use an xor ($\oplus$) operation. Recall that the xor operation can be used to detect odd parity since the $a \oplus b$ expression becomes '1' only when there is a single '1' from the inputs. Thus, the even-parity detector circuit can be implemented by an xor network followed by an inverter and the expression can be written as

$$even = (a(2) \oplus a(1) \oplus a(0))'$$

The architecture body based on this description is shown in Listing 2.2.

Listing 2.2 Even-parity detector based on an xor network

```
architecture xor_arch of even_detector is
   signal odd: std_logic;
begin
   even <= not odd;
   odd <= a(2) xor a(1) xor a(0);
end xor_arch;
```

Again, the two concurrent statements represent two circuit parts, and the conceptual diagram is shown in Figure 2.3. Since no explicit after clause is used, both statements take a δ-delay to operate.

2.2.2 Structural description

In a structural view, a circuit is constructed of smaller parts. The description specifies what types of parts are used and how these parts are connected. The description is essentially a schematic, representing a block diagram or circuit diagram. Although we treat a concurrent statement of the preceding section as a circuit part, it is our interpretation and the code is not considered as a real structural description. Formal VHDL structural description is done by using the concept of *component*. A component can be either an existing or a hypothetical part. It first has to be *declared* (make known) and then can be *instantiated* (actually used) in the architecture body as needed.

Let us consider the even-parity detector circuit again. Assume that there is a library with predesigned parts, `xor2` and `not1`, which perform the xor and inverting functions respectively. The even-parity detector circuit can be realized by the two parts, as shown in the circuit diagram of Figure 2.4. Based on the schematic, a structural description can be derived accordingly. The code of the architecture body is shown in Listing 2.3.

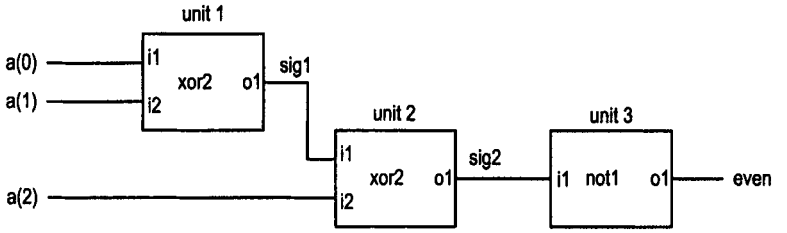


Figure 2.4 Structural diagram of the `str_arch` architecture.

Listing 2.3 Even-parity detector based on a structural description

```
architecture str_arch of even_detector is
   -- declaration for xor gate
   component xor2
      port(
         i1, i2: in std_logic;
         o1: out std_logic
      );
   end component;
   -- declaration for invertor
   component not1
      port(
         i1: in std_logic;
         o1: out std_logic
      );
   end component;
   signal sig1,sig2: std_logic;

begin
   -- instantiation of the 1st xor instance
   unit1: xor2
      port map (i1 => a(0), i2 => a(1), o1 => sig1);
   -- instantiation of the 2nd xor instance
   unit2: xor2
      port map (i1 => a(2), i2 => sig1, o1 => sig2);
   -- instantiation of invertor
   unit3: not1
      port map (i1 => sig2, o1 => even);
end str_arch;
```

Inside the architecture, the components are declared first. For example, the declaration for xor2 is

```
component xor2
   port(
      i1, i2: in std_logic;
      o1: out std_logic
   );
end component;
```

The information contained inside the declaration is similar to that of entity declaration, which specifies the input and output ports of a circuit. In addition to component declaration, there is also a declaration for two internal signals, `sig1` and `sig2`.

The architecture body consists of three statements, each representing a *component instantiation*. The first one is

```
unit1: xor2
   port map (i1=>a(0), i2=>a(1), o1=>sig1);
```

There are three elements in this statement. The first is the label, `unit1`, which serves as a unique id for this part. The second is the initiated component, `xor2`. The last is **port map** $\cdots$, which specifies the mapping between the formal signals (the I/O ports used in component declaration) and actual signals (the signals used in the architecture body). The mapping indicates that `i1`, `i2` and `o1` are connected to `a(0)`, `a(1)` and `sig1` respectively. The code is essentially the textual description of the circuit diagram in Figure 2.4. The three component instantiations together describe the complete circuit. The connections between the components are done implicitly by using the same signal names.

Component instantiation is one type of concurrent statement and can be mixed with other types of concurrent statements. When an architecture body consists only of component instantiations, as in this example, it is just a textual description of a schematic. This is a clumsy way for humans to conceptualize and comprehend this kind of representation. However, textual description put everything into a single VHDL framework so that the description can be handled by the same software tools. There is special design entry software that can convert a schematic to structural VHDL code and vice versa.

Component declaration contains only I/O port information, as in entity declaration. The components can be treated as empty sockets, which provide no clues about their internal functions. A component can be an existing, predesigned circuit or a hypothetical system that is still under construction. An architecture body will be bound with the component at the time of simulation or synthesis. In this example, the components may already be coded, compiled and stored in a library earlier. Their VHDL descriptions are shown in Listing 2.4.

Listing 2.4 Predesigned component

```
-- 2-input xor gate
library ieee;
use ieee.std_logic_1164.all;
entity xor2 is
   port(
      i1, i2: in std_logic;
      o1: out std_logic
  );
end xor2;

architecture beh_arch of xor2 is
begin
   o1 <= i1 xor i2;
end beh_arch;

-- invertor
library ieee;
use ieee.std_logic_1164.all;
entity not1 is
   port(
```

```
        i1: in std_logic;
        o1: out std_logic
    );
end not1;
architecture beh_arch of not1 is
begin
    o1 <= not i1;
end beh_arch;
```

Structural description and the use of components help the design in several ways. First, they facilitate hierarchical design. A complex system can be divided into several smaller subsystems, each represented by a component and designed individually. The subsystem, if needed, can be further divided into even smaller modules. Second, they provide a method to use predesigned circuits. These circuits, including complex IP cores and certain specialized library cells, can be instantiated in the description and treated as black boxes. Finally, structural description can be used to represent the result of synthesis: a gate- or cell-level netlist.

2.2.3 Abstract behavioral description

In a large design, the implementation can be very complex and the construction can be a time-consuming process. In the beginning, we frequently just want to study system operation rather than focusing on construction of the actual circuit, and prefer an abstract description. Since human reasoning and algorithms resemble a sequential process, the sequential semantics of traditional language is more adequate. VHDL provides language constructs that resemble the sequential semantics, including the use of variable and sequential execution. These features are considered as exceptions to the regular VHDL semantics, and they are encapsulated in a special construct, known as a *process*. This kind of code is sometimes referred to as *behavioral description*. However, there is no precise definition for the term behavioral description. According to VHDL, all codes, except for pure component instantiation, are considered as behavioral.

The basic skeleton of a process is

```
process(sensitivity_list)
    variable declaration;
begin
    sequential statements;
end process;
```

A process has a *sensitivity list*, which is composed of a set of signals. When a signal in the sensitivity list changes, the process is activated. Inside the process, the semantic is similar to that of a traditional programming language. Variables can be used and execution of the statements is sequential. The use of process is shown in two examples, both describing the even-parity detector circuit. The first example is based on the xor network, as in the `xor_arch` architecture. The architecture body is shown in Listing 2.5.

Listing 2.5 Even-parity detector based on a behavioral description

```
architecture beh1_arch of even_detector is
signal odd: std_logic;
begin
    -- invertor
    even <= not odd;
```

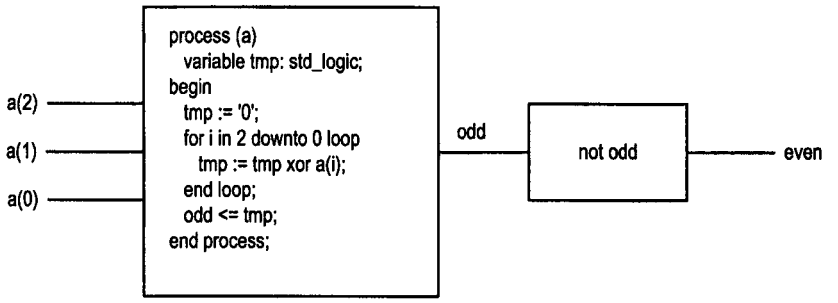


Figure 2.5 Conceptual diagram of the `beh1_arch` architecture.

```
   -- xor network for odd parity
   process(a)
      variable tmp: std_logic;
   begin
      tmp := '0';
      for i in 2 downto 0 loop
         tmp := tmp xor a(i);
      end loop;
      odd <= tmp;
   end process;
end beh1_arch;
```

The xor network is described by a process that utilizes a variable and a for loop statement. Unlike signal and signal assignment in a concurrent statement, the variable and loop do not have direct hardware counterparts. We treat a process as one indivisible part whose behavior is specified by the sequential statements. The graphic interpretation of the `beh1` architecture is shown in Figure 2.5.

The second example uses a single process to describe the desired operation in an algorithm. The algorithm first sums up the number of 1's from input, performs a modulo-2 operation to find the remainder, and then uses an if statement to check the value of the remainder to generate the final result. The VHDL code is shown in Listing 2.6.

Listing 2.6 Even-parity detector based on another behavioral description

```
architecture beh2_arch of even_detector is
begin
   process(a)
      variable sum, r: integer;
   begin
      sum := 0;
      for i in 2 downto 0 loop
         if a(i)='1' then
            sum := sum +1;
         end if;
      end loop ;
      r := sum mod 2;
      if (r=0) then
         even <= '1';
      else
```

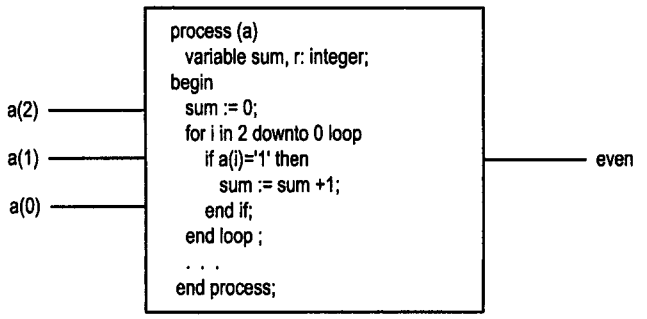


Figure 2.6 Conceptual diagram of the `beh2_arch` architecture.

```
        even <='0';
      end if;
   end process;
end beh2_arch;
```

Since there is only one process, the graphic interpretation has only one part, as in Figure 2.6. While the code is very straightforward and easy to understand, it provides no clues about the underlying structure or how to realize the code in hardware.

2.2.4 Testbench

One major use of a VHDL program is simulation, which is used to study the operation of a circuit or to verify the correctness of a design. Performing simulation is similar to doing an experiment with a physical circuit, in which we connect the circuit's input to a stimulus (e.g., a function generator) and observe the output (e.g., by logic analyzer). Simulating a VHDL description is like doing a virtual experiment, in which the physical circuit is replaced by the corresponding VHDL description. Furthermore, we can develop VHDL utility routines to imitate the stimulus generator (which is known as a *test vector generator*) and to collect and compare the output responses. The framework is known as a *testbench*.

A simple VHDL testbench for the previous even detection circuit is shown in Listing 2.7. The testbench includes a test vector generator that generates a stimulus and a verifier that verifies the correctness of the output response. The testbench consists of an entity declaration and architecture body. Since the testbench is self-contained, no port is specified in the entity declaration. There are three concurrent statements in the architecture body, including one component instantiation and two processes. The component instantiation specifies that the `even_detector` is used and its I/O pins are connected to the internal test generator and verifier. The first process is the stimulus generator. It produces all possible test vector combinations, from "000" to "111". These vectors are generated in sequential order, each lasting for 200 ns. The second process is the verifier. It takes the input test vector, waits for 100 ns to let the output settle down, checks the output value with the known value and reports the results. The two processes are for demonstration purposes only, and we don't need to worry about the syntax detail.

Listing 2.7 Simple testbench

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

```
entity even_detector_testbench is
end even_detector_testbench;

architecture tb_arch of even_detector_testbench is
   component even_detector
      port(
         a: in std_logic_vector(2 downto 0);
         even: out std_logic
      );
   end component;
   signal test_in: std_logic_vector(2 downto 0);
   signal test_out: std_logic;

begin
   -- instantiate the circuit under test
   uut: even_detector
      port map( a=>test_in, even=>test_out);
   -- test vector generator
   process
   begin
      test_in <= "000";
      wait for 200 ns;
      test_in <= "001";
      wait for 200 ns;
      test_in <= "010";
      wait for 200 ns;
      test_in <= "011";
      wait for 200 ns;
      test_in <= "100";
      wait for 200 ns;
      test_in <= "101";
      wait for 200 ns;
      test_in <= "110";
      wait for 200 ns;
      test_in <= "111";
      wait for 200 ns;
   end process;
   --verifier
   process
      variable error_status: boolean;
   begin
      wait on test_in;
      wait for 100 ns;
      if ((test_in="000" and test_out = '1') or
          (test_in="001" and test_out = '0') or
          (test_in="010" and test_out = '0') or
          (test_in="011" and test_out = '1') or
          (test_in="100" and test_out = '0') or
          (test_in="101" and test_out = '1') or
          (test_in="110" and test_out = '1') or
          (test_in="111" and test_out = '0'))
      then
```

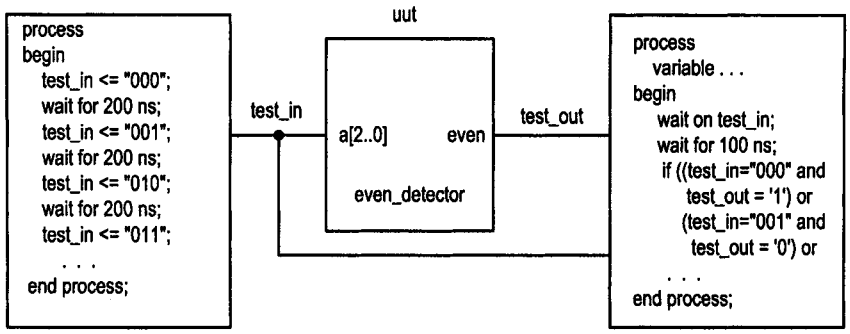


Figure 2.7 Conceptual diagram of an `even_detector` testbench.

```
            error_status := false;
        else
            error_status := true;
        end if;
        -- error reporting
        assert not error_status
            report "test failed."
            severity note;
    end process;
end tb_arch;
```

The graphic interpretation of this VHDL code is shown in Figure 2.7. Most of today's simulation software can keep track of the execution of a VHDL program and display the relevant information in a tabular or graphic format.

2.2.5 Configuration

The VHDL intentionally separates the entity declaration and architecture body into two independent design units. We can associate multiple architecture bodies with a single entity declaration. For example, the `even_detector` entity of this section has about a half dozen architecture bodies. At the time of simulation or synthesis, we can choose a specific architecture body to *bind* with the entity.

An analogy of the entity and architecture is the socket and IC chip. An entity declaration can be thought of as a socket of a printed circuit board, which is empty but has fixed input and output pins. Architecture bodies can be thought of as IC chips with the same outline. While the input and output pins of these chips are identical, their internal circuitry and performances may be very different. We can select a chip and insert it into the socket according to our particular need.

VHDL provides a mechanism, known as *configuration*, to specify the binding information. In the previous example, the `even_detector` entity has five different architecture bodies. The component declaration and component instantiation of `test_bench` does not specify which body is to be used. The `test_bench` is like a printed circuit board with an empty socket, and one of the five possible chips can be inserted into the circuit. A simple configuration declaration unit is shown in Listing 2.8, in which the `sop_arch` architecture is bound with the `even_detector` entity.

Listing 2.8 Simple configuration

```
configuration demo_config of even_detector_testbench is
   for tb_arch
      for uut: even_detector
         use entity work.even_detector(sop_arch);
      end for;
   end for;
end demo_config;
```

In a VHDL program, a configuration unit is not always needed. If there is no configuration unit, the entity is automatically bound with the last compiled architecture body. The configuration is particularly helpful for the development and verification of large systems.

2.3 VHDL IN DEVELOPMENT FLOW

The examples from the previous sections show the basic language constructs and capabilities of VHDL. The choice of these constructs is not accidental. They are carefully selected to facilitate system development. In the following subsections, we discuss the use of VHDL in the development flow and the difference between coding for modeling and coding for synthesis.

2.3.1 Scope of VHDL

The scope and coverage of VHDL in a simplified development flow is illustrated in Figure 2.8. The design of a complex system normally begins with an abstract high-level description, which describes the desired behavior of the system, and a testbench, which includes a set of test vectors to exercise various functions of the system. The description and testbench allow designers to study the system operation in detail, discover any misconception or inconsistency, clarify and finalize the specification, and eventually establish the desired I/O behavior for future verification. The `beh2_arch` architecture (in Listing 2.6) of `even_detector` and the corresponding `test_bench` (in Listing 2.7) resemble these kinds of codes. In a large system, the abstract description is normally not suitable for synthesis. It either leads to unnecessarily complex circuitry or cannot be synthesized at all.

Once the specification and behavior of a system are completely understood, a synthesis-oriented code can be developed. This code is normally an RT-level description and provides a "sketch" of the underlying hardware organization so that synthesis software can derive an efficient implementation. The `xor_arch` and `beh1_arch` architecture bodies in Listings 2.2 and 2.5 resemble this kind of description. A synthesis-oriented description needs to be verified first. By utilizing the VHDL configuration, we can bind the new architecture body to the entity and use the same testbench and the previously established test vectors. After comparing the simulation responses with the known results, we can easily determine whether the new description meets the specification.

Once verified, the synthesis-oriented description can be synthesized. The result is a gate-level netlist, represented by a structural VHDL description. The code will be similar to the `str_arch` architecture body Listing 2.3. In a large design, the description is normally too tedious for humans to comprehend. Instead, it is usually plugged into the testbench via a new configuration unit. The testbench will be simulated to verify the correctness of synthesis and to study the system timing. The netlist description can then be passed to placement and routing software for further processing. The placement and routing tool

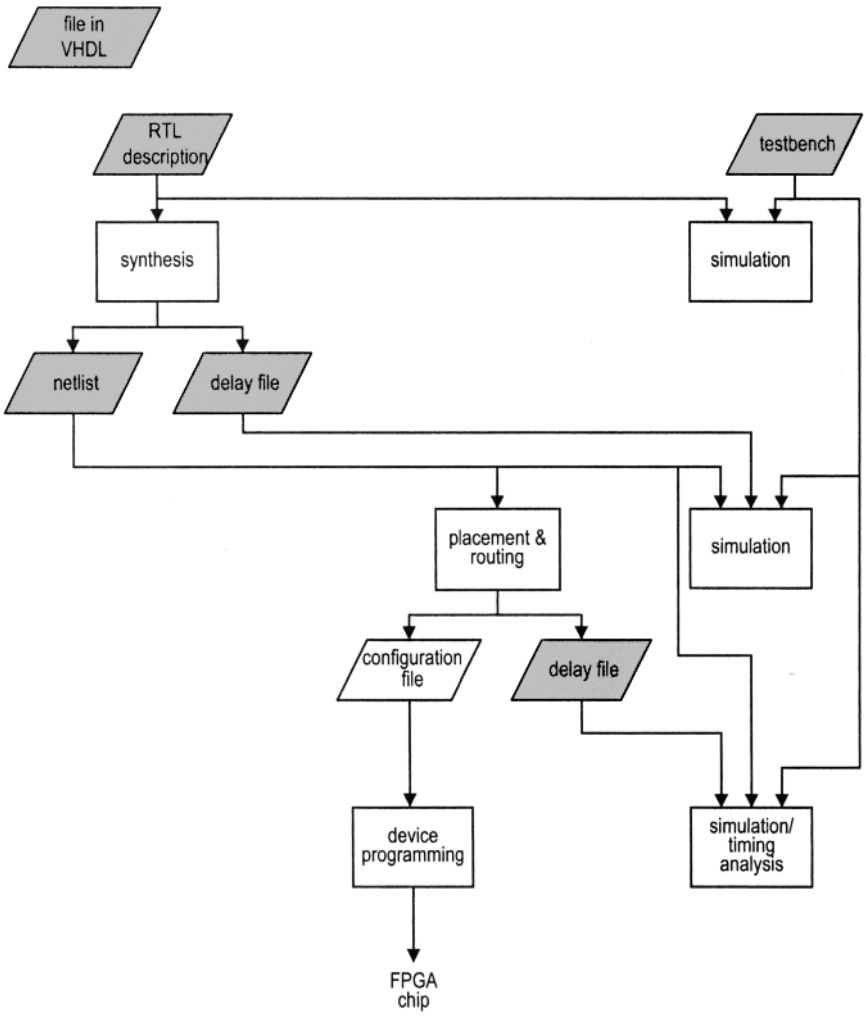


Figure 2.8 Coverage of VHDL in development flow.

will generate the layout or configuration files, which are not in VHDL. However, additional timing information will be augmented to the previous structural description. The new description will again be plugged into the testbench for final timing verification.

In summary, VHDL provides a unified environment for the entire development flow. It not only contains constructs to describe the design at various stages of the design, from the abstraction behavior to the post placement-and-routing cell-level netlist, but also provides a framework for simulation and verification.

2.3.2 Coding for synthesis

VHDL is used to model all aspects of digital hardware and to facilitate the entire design process. After a VHDL code is developed, it can be "executed" in a simulator or synthesizer. The natures of the two executions are quite different.

In simulation, the design is realized only in a virtual environment: the software simulator. The host computer utilizes its instruction set to mimic operation of the circuit. Since the host computer normally contains one processing unit, the circuit simulation is done sequentially, in which all constructs and operators of the VHDL code implicitly shared a single resource in a time-multiplexing fashion. In synthesis, on the other hand, all constructs and operators of the VHDL code are mapped to hardware. Let us consider a task that consists of 10 addition operations. In simulation, the number of addition operators, +, in VHDL code does not play a significant role since only one addition can be simulated at a time. In synthesis, each addition operator is mapped to a hardware adder, which is fairly complex, and thus it is desirable to share the hardware and to reduce the number of addition operators in VHDL description. Similarly, sophisticated control structures, such as loop or conditional branch, can be easily simulated in a sequential host but cannot be efficiently mapped to hardware.

For synthesis, only a subset of VHDL can be used. Many modeling language constructs, such as file operations and assertion statements, are not meaningful for hardware implementation. The others, such as floating-point number or complicated operators, are too complex to be synthesized automatically. IEEE defines a subset of VHDL that is suitable for RT-level synthesis in IEEE standard 1076.6. Even though the scope of the synthesizable subset is restricted, it still contains a rich collection of language constructs and is very flexible. The same circuit can be coded in a wide variety of descriptions, ranging from abstract high-level behavioral-like specification to detailed gate-level structural description. Although all these descriptions can be synthesized, there is no guarantee that the synthesized circuit is an efficient implementation. The synthesis software can perform only local search and local optimization, and the resulting circuit depends heavily on the initial description. An inadequate description consumes a large amount of CPU time during synthesis, introduces excessively complex circuitry and even fails to be synthesized.

This book focuses on RT-level design and synthesis, not VHDL. We are using VHDL as a vehicle to describe our intended hardware implementation. Our emphasis is on coding for synthesis, which means to develop VHDL code that accurately describes the underlying hardware structure and to provide adequate information to guide the synthesis software to generate an efficient implementation.

2.4 BIBLIOGRAPHIC NOTES

HDL is very different from a traditional programming language. The book, *Hardware Description Languages: Concepts and Principles* by S. Ghosh, discusses general issues

in designing HDL. Both VHDL and Verilog are IEEE standards. They are documented by *IEEE Standard VHDL Language Reference Manual* and *IEEE Standard for Verilog Hardware Description Language* respectively. The other relevant VHDL standards are also documented in the IEEE publications. The standards themselves are difficult to read. The text, *The Designer's Guide to VHDL* by P. J. Ashenden, provides a detailed and comprehensive discussion of VHDL. The texts, *Starter's Guide to Verilog 2001* by M. D. Ciletti, and *Verilog HDL, 2nd edition*, by S. Palnitkar, provide good coverage on Verilog.

The verification of a design and the derivation of the testbench are two of the major tasks in the development flow. The text, *Writing Testbenches: Functional Verification of HDL Models, 2nd edition*, by J. Bergeron, discusses this topic in detail.

Problems

2.1 What are the syntax and semantics of a programming language?

2.2 List three major differences between an HDL and a traditional programming language, such as C.

2.3 In a traditional programming language, such as C, we can write the statement `a=!a`, and in VHDL, we can write a concurrent statement as `a <= not a after 10 ns;`.

(a) Draw the circuit diagram for the VHDL statement.

(b) Describe the operation of the circuit in part (a).

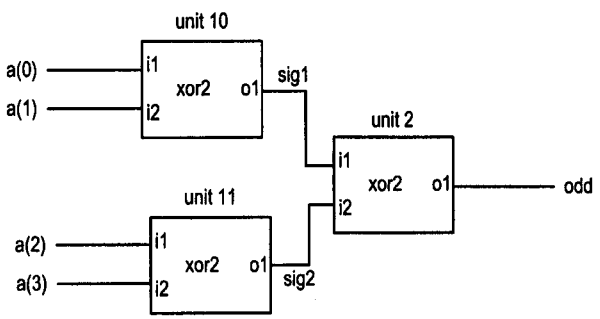
(c) Discuss the differences between the VHDL and C statements.

2.4 For the even-parity detector circuit, rewrite the expression in product-of-sums format. Revise the code of the `sop_arch` architecture body according to the new expression.

2.5 For the VHDL code shown below, treat each concurrent statement as a circuit part and draw the conceptual block diagram accordingly.

```
y <= e1 and e0;
e0 <= (a0 and b0) or ((not a0) and (not b0));
e1 <= (a1 and b1) or ((not a1) and (not b1));
```

2.6 A circuit diagram consisting of the `xor2` component is shown below. Follow the code of the `str_arch` architecture body to derive a structural VHDL description for this circuit.



2.7 The VHDL structural description of a circuit is shown below. Derive the block diagram according to the code.

```
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);
      p_ten: out std_logic
   );
end hundred_counter;

architecture str_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);
end str_arch;
```

2.8 From the description of the VHDL process in Section 2.2.3, discuss the differences between the VHDL process and the traditional programming languages' procedure and function.

2.9 We want to change the input of the even-parity detector circuit from 3 bits to 4 bits, i.e., from `a(2 downto 0)` to `a(3 downto 0)`. Revise the VHDL codes of the five architecture bodies to accommodate the change.

2.10 If we want to change the input of the even-parity detector circuit from 3 bits to 10 bits, discuss the amount of code modifications needed in each architecture body.

2.11 Explain why VHDL treats the entity declaration and architecture body as two separate design units.

2.12 Think of two applications that can use the configuration construct of the VHDL.