FPGA PROTOTYPING BY VHDL EXAMPLES

Xilinx Spartan™-3 Version

Pong P. Chu
Cleveland State University

A JOHN WILEY & SONS, INC., PUBLICATION

This Page Intentionally Left Blank

FPGA PROTOTYPING BY VHDL EXAMPLES

This Page Intentionally Left Blank

FPGA PROTOTYPING BY VHDL EXAMPLES

Xilinx Spartan™-3 Version

Pong P. Chu
Cleveland State University

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Chu, Pong P., 1959–
FPGA prototyping by VHDL examples / Pong P. Chu.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-18531-5 (cloth : alk. paper)
1. Field programmable gate arrays—Design and construction. 2. Prototypes, Engineering. 3.VHDL (Computer hardware description language) I. Title.
TK7895.G36C485 2008
621.39'5—dc22 2007029063

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my parents, Chia-Chi and Chi-Te, my wife, Lee, and my daughter, Patricia

This Page Intentionally Left Blank

CONTENTS

PREFACE

HDL (hardware description language) and *FPGA* (field-programmable gate array) devices allow designers to quickly develop and simulate a sophisticated digital circuit, realize it on a prototyping device, and verify operation of the physical implementation. As these technologies mature, they have become mainstream practice. We can now use a PC and an inexpensive FPGA prototyping board to construct a complex and sophisticated digital system. This book uses a "learning by doing" approach and illustrates the FPGA and HDL development and design process by a series of examples. A wide range of examples is included, from a simple gate-level circuit to an embedded system with an 8-bit soft-core microcontroller and customized I/O peripherals. All examples can be synthesized and physically tested on a prototyping board.

Focus and audience

Focus The main focus of this book is on the effective derivation of hardware, not the syntax of HDL. Instead of explaining every language construct, the book is limited to a small synthesizable subset and uses about a dozen code templates to provide the skeletons of various types of circuits. These templates are general and can easily be integrated to construct a large, complex system. Although this approach limits the "freedom" of syntactic expression, it will not prevent us from developing innovative hardware architecture. Because of the generality and flexibility of HDL, the same circuit can usually be described by a wide variety of language constructs and coding styles. Many of these codes are intended for modeling. They may lead to unnecessarily complex hardware implementation and sometimes cannot be synthesized at all. The template approach actually forces us to think more about hardware and develop a good coding practice for synthesis. Since we are

more interested in hardware, it is more beneficial to spend time on developing 10 different hardware architectures with the same code template rather than describing the same circuit with 10 different versions of codes.

There are two popular HDLs, *VHDL* and *Verilog*. Both languages are used widely and are IEEE standards. This book uses VHDL, and a separate book with a similar title uses Verilog. Despite the drastic syntactic differences in the two languages, their capabilities are very similar, particularly for our purposes. After we comprehend the design practice and coding methodology in one language, learning the other language is rather straightforward.

Although the book is intended for beginning designers, the examples follow strict design guidelines and prepare readers for future endeavors. The coding and design practice is "forward compatible," which means that:

- The same practice can be applied to large design in the future.
- The same practice can aid other system development tasks, including simulation, timing analysis, verification, and testing.
- The same practice can be applied to ASIC technology and different types of FPGA devices.
- The code can be accepted by synthesis software from different vendors.

In summary, the book is a hands-on, hardware-centric text that involves *minimal HDL overhead* and follows good design and coding practice to achieve *maximal forward comparability*.

Audience and perquisites The book contains three major parts: basic digital circuits, peripheral modules, and embedded microcontroller. The intended audience is students in an introductory or advanced digital system design course as well as practicing engineers who wish to learn FPGA- and HDL-based development. For the materials in the first two parts, readers need to have a basic knowledge of digital systems, usually a required course in electrical engineering and computer engineering curricula. For the materials in the third part, prior exposure to assembly language programming will be helpful.

Logistics

Although a major goal of this book is to teach readers to develop software-independent and device-neutral HDL codes, we have to choose a software package and a prototyping board to synthesize and implement the design examples. The synthesis software and FPGA devices from Xilinx, a leading manufacture in this area, are used in the book.

Software The synthesis software used in the book is the Web version of the Xilinx *ISE* package. The functionality is of this version is similar to that of the full version but supports only a limited number of devices. Most introductory development boards use FPGA devices from the inexpensive Spartan-3 family. Since the Web version supports the Spartan-3 device, it fits our need. The simulation software used in the book is the starter version of Mentor Graphics' *ModelSim XE III* package. It is a customized edition of *ModelSim*. Both software packages are free and can be downloaded from Xilinx's Web site.

FPGA prototyping board This book is prepared to be used with several entry-level FPGA prototyping boards manufactured by Digilent Inc., including the *Spartan-3 Starter*, *Nexys-2*, and *Basys* boards, all of which contain a Spartan-3/3E FPGA device and have

similar I/O peripherals. The design examples in the book are based on the Spartan-3 Starter board (or simply the *S3 board*), but most of them can be used directly in other boards as well. The applicability of the HDL codes is summarized below.

- **Spartan-3 Starter 3 (S3) board**. The S3 board contains all the peripherals and no additional accessory module is needed. All HDL codes and discussions can be applied to this board directly.
- **Nexys-2 board**. The Nexys-2 board is a newer board, which contains a larger FPGA device and a larger memory chip. Its peripherals are similar to those in the S3 board. There are two differences. First, the "color depth" of its VGA interface is expanded from 3 bits to 8 bits. The the output of the VGA interface circuits discussed in Chapters 12 and 13 needs to be modified accordingly. Second, it contains a more sophisticated external memory device. Although the device can be configured as an asynchronous SRAM, the timing characteristics is different from that of the S3 board's memory device, and thus the HDL codes for the memory controller in Chapter 10 cannot be used directly. However, the same design principle can be applied to construct a new controller.
- **Basys board**. The Basys board is a simpler board. It lacks the RS-232 connector. To implement the UART module and the serial interface discussed in Chapter 7, we need Digilent's *RS-232 converter peripheral module*. The Basys board has no external memory devices, and thus the discussion of the memory controller in Chapter 10 is not applicable.
- **Other FPGA boards**. Most peripherals discussed in this book are de facto industrial standards, and the corresponding HDL codes can be used as long as a board provides proper analog interface circuits and connectors. Except for the Xilinx-specific portions, the codes can be applied to the boards based on the FPGA devices from other manufacturers as well.

PC Accessories The design examples include interfaces to several PC peripheral devices. A keyboard, a mouse, and a VGA monitor are required for the respective modules, and a "straight-through" serial cable (the most commonly used type) is required for the UART module. These accessories are widely available and can probably be obtained from an old PC.

Book organization

The book is divided into three major parts. Part I introduces the elementary HDL constructs and their hardware counterparts, and demonstrates the construction of a basic digital circuit with these constructs. It consists of six chapters:

- Chapter 1 describes the skeleton of an HDL program, basic language syntax, and logical operators. Gate-level combinational circuits are derived with these language constructs.
- Chapter 2 provides an overview of an FPGA device, prototyping board, and development flow. The development process is demonstrated by a tutorial on Xilinx ISE synthesis software and a tutorial on Mentor Graphics ModelSim simulation software.
- Chapter 3 introduces HDL's relational and arithmetic operators and routing constructs. These correspond to medium-sized components, such as comparators, adders, and multiplexers. Module-level combinational circuits are derived with these language constructs.

- Chapter 4 covers the codes for memory elements and the construction of "regular" sequential circuits, such as counters and shift registers, in which the state transitions exhibit a regular pattern.
- Chapter 5 discusses the construction of a finite state machine (FSM), which is a sequential circuit whose state transitions do not exhibit a simple, regular pattern.
- Chapter 6 presents the construction of an FSM with data path (FSMD). The FSMD is used to implement register transfer (RT) methodology, in which the system operation is described by data transfers and manipulations among registers.

Part II applies the techniques from Part I to design an array of peripheral modules for the prototyping board. Each chapter covers the development, implementation, and verification of an individual peripheral. These modules can be incorporated to a larger project. Part II consists of seven chapters:

- Chapter 7 discusses the design of a universal asynchronous receiver and transmitter (UART), which provides a serial link to receive and transmit data via the prototyping board's RS-232 port.
- Chapter 8 covers the design of a keyboard interface, which reads scan code from a keyboard. The keyboard is connected via the prototyping board's PS2 port.
- Chapter 9 covers the design of a mouse interface, which obtains the button and movement information from a mouse. The mouse is also connected via the prototyping board's PS2 port.
- Chapter 10 discusses the implementation and timing issues of a memory controller. The controller is used to read data from and write data to the two static random access memory (SRAM) devices on the S3 board.
- Chapter 11 discusses the inference and application of Spartan-3 device-specific components. The focus is on the FPGA's internal memory blocks and the digital clock management (DCM) circuit.
- Chapter 12 presents the design and implementation of a video controller. The discussion covers the generation of video synchronization signals and shows the construction of simple bit- and object-mapped graphical interface. The monitor is connected to the prototyping board's VGA port.
- Chapter 13 continues development of the video controller. The discussion illustrates the construction of text interface and general tile-mapped scheme.

Part III introduces an FPGA-based soft-core microcontroller, known as *PicoBlaze*, and demonstrates the integration of a general-purpose processor and customized circuit. It includes four chapters:

- Chapter 14 provides an overview of the organization and instruction set of PicoBlaze.
- Chapter 15 introduces the basic assembly programming and provides an overview of the development process.
- Chapter 16 discusses PicoBlaze's I/O feature and illustrates the procedure to derive customized circuits to interface other I/O peripherals.
- Chapter 17 discusses PicoBlaze's interrupt capability and demonstrates the construction of a customized interrupt-handling circuit.

In addition to regular chapters, the appendix summarizes and lists all code templates.

Special marks$^{Xilinx\ specific}$ While the examples of this book are implemented on a Xilinx-based prototyping board and the codes are synthesized by Xilinx ISE software, we try to make the HDL codes device-independent and software-neutral as much as possible. Most discussions and codes can be applied to different target devices and different synthesis

software as well. However, certain codes or device features are unique to Xilinx ISE software or Spartan-3 FPGA devices. We use the *Xilinx specific* superscript, as in the heading of this section, to indicate that the discussion in the corresponding section or chapter is unique to Xilinx.

Similarly, we use marginal notes, such as the one shown on the outer edge, to indicate that the discussion in the paragraph is unique to Xilinx. This note indicates that the code or design is no longer portable and needs to be revised when a different software package or target device is used.

Xilinx specific

Instructional use

The book can be a good companion text for an introductory digital systems course or an advanced project-oriented course. In an introductory digital systems course, the book supplies the lab portion of the curriculum. The chapters in Part I basically follow the sequence of a typical curriculum and can be presented along with regular lectures. One or two peripheral modules can be selected as case studies, and corresponding experiments can be used as term projects.

In an advanced project-oriented course, the book provides a base for independent projects. The materials in Part I should be treated as an overview or refresher, which provides a general background on HDL, synthesis, and FPGA boards. Some modules in Part II can be used to demonstrate the design of more complex circuits. These modules can also be considered as building blocks (i.e., IPs) or subsystems to be integrated into final projects. The PicoBlaze microcontroller in Part III can be used as general-purpose processor if an embedded-system type of project is desired.

Companion Web site

An accompanying Web site (`http://academic.csuohio.edu/chu_p/rtl`) provides additional information, including the following materials:

- Errata
- Code templates
- HDL code listing and relevant files
- Links to synthesis and simulation software
- Links to referenced materials
- Additional project ideas

Errata The book is self-prepared, which means that the author has produced all aspects of the text, including illustrations, tables, code listings, indexing, and formatting. As errors are always bound to happen, the accompanying Web site provides an updated errata sheet and a place to report errors.

P. P. CHU

Cleveland, Ohio
October 2007

This Page Intentionally Left Blank

ACKNOWLEDGMENTS

The author would like to express his gratitude to Professor George L. Kramerich for his encouragement and help.

The author also thanks John Wiley & Sons, Inc. for giving permission to use Figures 3.1, 3.2, 4.2, 4.10, 4.11, and 6.5 from my text *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, and Xilinx, Inc. for giving permission to use Figures 2.3 and 8.3 from the *Spartan-3 Starter Kit Board User Guide*.

All trademarks used or referred to in this book are the property of their respective owners.

P. P. Chu

This Page Intentionally Left Blank

PART I

BASIC DIGITAL CIRCUITS

This Page Intentionally Left Blank

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

1.1 INTRODUCTION

VHDL stands for "VHSIC (very high-speed integrated circuit) hardware description language." It was originally sponsored by the U.S. Department of Defense and later transferred to the IEEE (Institute of Electrical and Electronics Engineers). The language is formally defined by IEEE Standard 1076. The standard was ratified in 1987 (referred to as VHDL 87), and revised several times. This book mainly follows the revision in 1993 (referred to as VHDL 93).

VHDL is intended for describing and modeling a digital system at various levels and is an extremely complex language. The focus of this book is on hardware design rather than the language. Instead of covering every aspect of VHDL, we introduce the key VHDL synthesis constructs by examining a collection of examples. Detailed VHDL coverage may be explored through the sources listed in the Bibliography.

In this chapter, we use a simple comparator to illustrate the skeleton of a VHDL program. The description uses only logical operators and represents a gate-level combinational circuit, which is composed of simple logic gates. In Chapter 3, we cover the more sophisticated VHDL operators and constructs and examine module-level combinational circuits, which are composed of intermediate-sized components, such as adders, comparators, and multiplexers.

FPGA Prototyping by VHDL Examples. By Pong P. Chu

Table 1.1 Truth table of a 1-bit equality comparator

input $i0\ i1$	output eq
0 0	1
0 1	0
1 0	0
1 1	1

1.2 GENERAL DESCRIPTION

Consider a 1-bit equality comparator with two inputs, `i0` and `i1`, and an output, `eq`. The `eq` signal is asserted when `i0` and `i1` are equal. The truth table of this circuit is shown in Table 1.1.

Assume that we want to use basic logic gates, which include *not*, *and*, *or*, and *xor* cells, to implement the circuit. One way to describe the circuit is to use a sum-of-products format. The logic expression is

$$eq = i0 \cdot i1 + i0' \cdot i1'$$

One possible corresponding VHDL code is shown in Listing 1.1. We examine the language constructs and statements of this code in the following subsections.

Listing 1.1 Gate-level implementation of a 1-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq1 is
   port(
      i0, i1: in std_logic;
      eq: out std_logic
   );
end eq1;

architecture sop_arch of eq1 is
   signal p0, p1: std_logic;
begin
   -- sum of two product terms
   eq <= p0 or p1;
   -- product terms
   p0 <= (not i0) and (not i1);
   p1 <= i0 and i1;
end sop_arch;
```

1.2.1 Basic lexical rules

VHDL is case insensitive, which means that upper- and lowercase letters can be used interchangeably, and free formatting, which means that spaces and blank lines can be inserted freely. It is good practice to add proper spaces to make the code clear and to associate special meaning with cases. In this book, we reserve uppercase letters for constants.

An *identifier* is the name of an object and is composed of 26 letters, digits, and the underscore (_), as in `i0`, `i1`, and `data_bus1_enable`. The identifier must start with a letter.

The comments start with `--` and the text after it is ignored. In this book, the VHDL keywords are shown in boldface type, as in **entity**, and the comments are shown in italics type, as in

```
-- this is a comment
```

1.2.2 Library and package

The first two lines,

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

invoke the `std_logic_1164` package from the `ieee` library. The package and library allow us to add additional types, operators, functions, etc. to VHDL. The two statements are needed because a special data type is used in the code.

1.2.3 Entity declaration

The entity declaration

```
entity eq1 is
   port(
      i0, i1: in std_logic;
      eq: out std_logic
   );
end eq1;
```

essentially outlines the I/O signals of the circuit. The first line indicates that the name of the circuit is `eq1`, and the port section specifies the I/O signals. The basic format for an I/O port declaration is

```
signal_name1, signal_name2, ... : mode data_type;
```

The `mode` term can be **in** or **out**, which indicates that the corresponding signals flow "into" or "out of" the circuit. It can also be **inout**, for bidirectional signals.

1.2.4 Data type and operators

VHDL is a *strongly typed language*, which means that an object must have a data type and only the defined values and operations can be applied to the object. Although VHDL is rich in data types, our discussion is limited to a small set of predefined types that are suitable for synthesis, mainly the `std_logic` type and its variants.

std_logic type The `std_logic` type is defined in the `std_logic_1164` package and consists of nine values. Three of the values, `'0'`, `'1'`, and `'Z'`, which stand for logical 0, logical 1, and high impedance, can be synthesized. Two values, `'U'` and `'X'`, which stand for "uninitialized" and "unknown" (e.g., when signals with `'0'` and `'1'` values are tied together), may be encountered in simulation. The other four values, `'-'`, `'H'`, `'L'`, and `'W'`, are not used in this book.

A signal in a digital circuit frequently contains multiple bits. The `std_logic_vector` data type, which is defined as an array with elements of `std_logic`, can be used for this purpose. For example, let `a` be an 8-bit input port. It can be declared as

```
a: in std_logic_vector(7 downto 0);
```

We can use term like `a(7 downto 4)` to specify a desired range and term like `a(1)` to access a single element of the array. The array can also be declared in ascending order:

```
a: in std_logic_vector(0 to 7);
```

We generally avoid this format since it is more natural to associate the MSB with the leftmost position.

Logical operators Several logical operators, including **not**, **and**, **or**, and **xor**, are defined over the `std_logic_vector` and `std_logic` data type. Bit-wise operation is used when an operator is applied to an object with the `std_logic_vector` data type. Note that the **and**, **or**, and **xor** operators have the same precedence and we need to use parentheses to specify the desired order of evaluation, as in

```
(a and b) or (c and d)
```

1.2.5 Architecture body

The architecture body,

```
architecture sop_arch of eq1 is
   signal p0, p1: std_logic;
begin
   -- sum of two product terms
   eq <= p0 or p1;
   -- product terms
   p0 <= (not i0) and (not i1);
   p1 <= i0 and i1;
end sop_arch;
```

describes operation of the circuit. VHDL allows multiple bodies associated with an entity, and thus the body is identified by the name `sop_arch` ("sum-of-products architecture").

The architecture body may include an optional declaration section, which specifies constants, internal signals, and so on. Two internal signals are declared in this program:

```
signal p0, p1: std_logic;
```

The main description, encompassed between **begin** and **end**, contains three *concurrent statements*. Unlike a program in C language, in which the statements are executed sequentially, concurrent statements are like circuit parts that operate in parallel. The signal on the left-hand side of a statement can be considered as the output of that part, and the expression specifies the circuit function and corresponding input signals. For example, consider the statement

```
eq <= p0 or p1;
```

It is a circuit that performs the or operation. When `p0` or `p1` changes its value, this statement is activated and the expression is evaluated. The new value is assigned to `eq` after the default propagation delay.

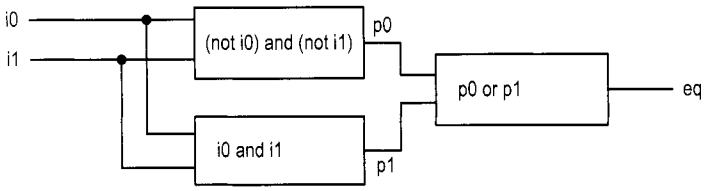

Figure 1.1 Graphical representation of a comparator program.

The graphical representation of this program is shown in Figure 1.1. The three circuit parts represent the three concurrent statements. The connections among these parts are implicitly specified by the signal and port names. The order of the concurrent statements is clearly irrelevant and the statements can be rearranged arbitrarily.

1.2.6 Code of a 2-bit comparator

We can expand the comparator to 2-bit inputs. Let the input be a and b and the output be aeqb. The aeqb signal is asserted when both bits of a and b are equal. The code is shown in Listing 1.2.

Listing 1.2 Gate-level implementation of a 2-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq2 is
   port(
      a, b: in std_logic_vector(1 downto 0);
      aeqb: out std_logic
   );
end eq2;

architecture sop_arch of eq2 is
   signal p0,p1,p2,p3: std_logic;
begin
   -- sum of product terms
   aeqb <= p0 or p1 or p2 or p3;
   -- product terms
   p0 <= ((not a(1)) and (not b(1))) and
         ((not a(0)) and (not b(0)));
   p1 <= ((not a(1)) and (not b(1))) and (a(0) and b(0));
   p2 <= (a(1) and b(1)) and ((not a(0)) and (not b(0)));
   p3 <= (a(1) and b(1)) and (a(0) and b(0));
end sop_arch;
```

The a and b ports are now declared as a two-element std_logic_vector. Derivation of the architecture body is similar to that of a 1-bit comparator. The p0, p1, p2, and p3 signals represent the results of the four product terms, and the final result, aeqb, is the logic expression in sum-of-products format.

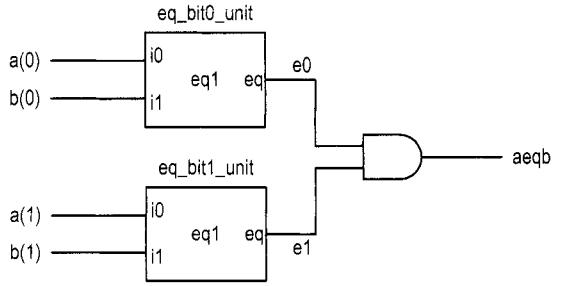

Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

1.3 STRUCTURAL DESCRIPTION

A digital system is frequently composed of several smaller subsystems. This allows us to build a large system from simpler or predesigned components. VHDL provides a mechanism, known as *component instantiation*, to perform this task. This type of code is called *structural description*.

An alternative to the design of the 2-bit comparator of Section 1.2.6 is to utilize the previously constructed 1-bit comparators as the building blocks. The diagram is shown in Figure 1.2, in which two 1-bit comparators are used to check the two individual bits and their results are fed to an and cell. The `aeqb` signal is asserted only when the two bits are equal.

The corresponding code is shown in Listing 1.3. Note that the entity declaration is the same and thus is not included.

Listing 1.3 Structural description of a 2-bit comparator

```
architecture struc_arch of eq2 is
   signal e0, e1: std_logic;
begin
   -- instantiate two 1-bit comparators
   eq_bit0_unit: entity work.eq1(sop_arch)
      port map(i0=>a(0), i1=>b(0), eq=>e0);
   eq_bit1_unit: entity work.eq1(sop_arch)
      port map(i0=>a(1), i1=>b(1), eq=>e1);
   -- a and b are equal if individual bits are equal
   aeqb <= e0 and e1;
end struc_arch;
```

The code includes two component instantiation statements, whose syntax is:

```
unit_label: entity lib_name.entity_name(arch_name)
   port map(
      formal_signal=>actual_signal,
      formal_signal=>actual_signal,
      . . .
   );
```

The first portion of the statement specifies which component is used. The `unit_label` term gives a unique id for an instance, the `lib_name` term indicates where (i.e., which library) the component resides, and the `entity_name` and `arch_name` terms indicate the names of the

entity and architecture. The `arch_name` term is optional. If it is omitted, the last compiled architecture body will be used. The second portion is port mapping, which indicates the connection between *formal signals*, which are I/O ports declared in a component's entity declaration, and *actual signals*, which are the signals used in the architecture body.

The first component instantiation statement is

```
eq_bit0_unit: entity work.eq1(sop_arch)
   port map(i0=>a(0), i1=>b(0), eq=>e0);
```

The `work` library is the default library in which the compiled entity and architecture units are stored, and `eq1` and `sop_arch` are the names of the entity and architecture defined in Listing 1.1. The port mapping reflects the connections shown in Figure 1.2. The component instantiation statement is also a concurrent statement and represents a circuit that is encompassed in a "black box" whose function is defined in another module.

This example demonstrates the close relationship between a block diagram and code. The code is essentially a textual description of a schematic. Although it is a clumsy way for humans to comprehend a diagram, it puts all representations into a single HDL framework. The Xilinx ISE package includes a simple schematic editor utility that can perform schematic capture in graphic format and then convert the diagram into an HDL structural description. **Xilinx specific**

The component instantiation statement is added in VHDL 93. Older codes may use the mechanism in VHDL 87, in which a component must first be declared (i.e., made known) and then used. The code in this format is shown in Listing 1.4.

Listing 1.4 Structural description with VHDL-87

```
architecture vhd_87_arch of eq2 is
   -- component declaration
   component eq1
      port(
         i0, i1: in std_logic;
         eq: out std_logic
      );
   end component;
   signal e0, e1: std_logic;
begin
   -- instantiate two 1-bit comparators
   eq_bit0_unit: eq1   -- use the declared name, eq1
      port map(i0=>a(0), i1=>b(0), eq=>e0);
   eq_bit1_unit: eq1   -- use the declared name, eq1
      port map(i0=>a(1), i1=>b(1), eq=>e1);
   -- a and b are equal if individual bits are equal
   aeqb <= e0 and e1;
end vhd_87_arch;
```

Note that the original clause,

```
eq_bit0_unit: entity work.eq1(sop_arch)
```

is replaced by a clause with the declared component name

```
eq_bit0_unit: eq1
```

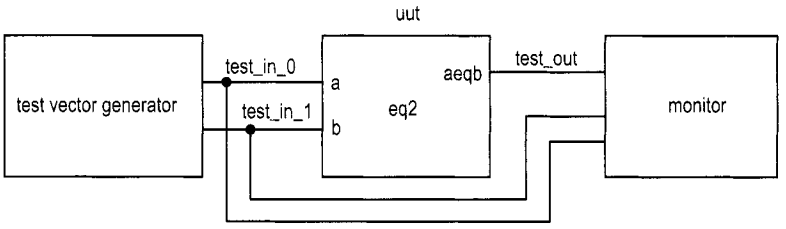

Figure 1.3 Testbench for a 2-bit comparator.

1.4 TESTBENCH

After code is developed, it can be *simulated* in a host computer to verify the correctness of the circuit operation and can be *synthesized* to a physical device. Simulation is usually performed within the same HDL framework. We create a special program, known as a *testbench*, to mimic a physical lab bench. The sketch of a 2-bit comparator testbench program is shown in Figure 1.3. The `uut` block is the unit under test, the `test vector generator` block generates testing input patterns, and the `monitor` block examines the output responses.

A simple testbench for the 2-bit comparator is shown in Listing 1.5.

Listing 1.5 Testbench for a 2-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq2_testbench is
end eq2_testbench;

architecture tb_arch of eq2_testbench is
   signal test_in0, test_in1: std_logic_vector(1 downto 0);
   signal test_out: std_logic;
begin
   -- instantiate the circuit under test
   uut: entity work.eq2(struc_arch)
      port map(a=>test_in0, b=>test_in1, aeqb=>test_out);
   -- test vector generator
   process
   begin
      -- test vector 1
      test_in0 <= "00";
      test_in1 <= "00";
      wait for 200 ns;
      -- test vector 2
      test_in0 <= "01";
      test_in1 <= "00";
      wait for 200 ns;
      -- test vector 3
      test_in0 <= "01";
      test_in1 <= "11";
      wait for 200 ns;
      -- test vector 4
```

```
        test_in0 <= "10";
        test_in1 <= "10";
        wait for 200 ns;
        -- test vector 5
        test_in0 <= "10";
        test_in1 <= "00";
        wait for 200 ns;
        -- test vector 6
        test_in0 <= "11";
        test_in1 <= "11";
        wait for 200 ns;
        -- test vector 7
        test_in0 <= "11";
        test_in1 <= "01";
        wait for 200 ns;
    end process;
end tb_arch;
```

The code consists of a component instantiation statement, which creates an instance of a 2-bit comparator, and a process statement, which generates a sequence of test patterns.

The process statement is a special VHDL construct in which the operations are performed sequentially. Each test pattern is generated by three statements. For example,

```
        -- test vector 2
        test_in0 <= "01";
        test_in1 <= "00";
        wait for 200 ns;
```

The first two statements specify the values for the `test_in0` and `test_in1` signals, and the third indicates that the two values will last for 200 ns.

The code has no monitor. We can observe the input and output waveforms on a simulator's display, which can be treated as a "virtual logic analyzer." The simulated timing diagram of this testbench is shown in Figure 2.16.

Writing code for a comprehensive test vector generator and a monitor requires detailed knowledge of VHDL and is beyond the scope of this book. This listing can serve as a testbench template for other combinational circuits. We can substitute the `uut` instance and modify the test patterns according to the new circuit.

1.5 BIBLIOGRAPHIC NOTES

A short bibliographic section appears at the end of each chapter to provide some of the most relevant references for further exploration. A comprehensive bibliography is included at the end of the book.

VHDL is a complex language. *The Designer's Guide to VHDL* by P. J. Ashenden provides detailed coverage of the language's syntax and constructs. The author's *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability* provides a comprehensive discussion on developing effective, synthesizable codes. The derivation of the testbench for a large digital system is a difficult task. *Writing Testbenches: Functional Verification of HDL Models, 2nd edition*, by J. Bergeron focuses on this topic.

1.6 SUGGESTED EXPERIMENTS

At the end of each chapter, some experiments are suggested as exercises. The experiments help us to better understand the concepts and provide a hands-on opportunity to design and debug actual circuits.

1.6.1 Code for gate-level greater-than circuit

Develop the HDL codes in Experiment 2.9.1. The code can be simulated and synthesized after we complete Chapter 2.

1.6.2 Code for gate-level binary decoder

Develop the HDL codes in Experiment 2.9.2. The code can be simulated and synthesized after we complete Chapter 2.