PART I

BASIC DIGITAL CIRCUITS DEVELOPMENT

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

HDL (hardware description language) is used to describe and model digital systems. VHDL is one of the two major HDLs. 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.

1.1 OVERVIEW OF VHDL

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 first ratified in 1987 and revised several times. The main versions are referred to as VHDL-87, VHDL-93, VHDL-2000, and VHDL-2008. The synthesis subset does not change significantly after VHDL-93 and this book mainly follows 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 Bibliographic Notes section.

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

In this chapter, we introduce the HDL concepts, basic VHDL language constructs, logical operators, and program structure. A simple gate-level combinational circuit is used for demonstration. 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.

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

```vhdl
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" 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, such as a data bus. 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 a term like `a(7 downto 4)` to specify a desired range and a 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. Bitwise 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 a conventional language, such as C,
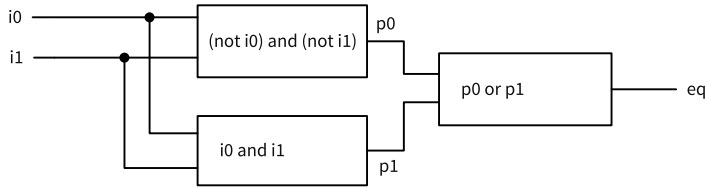
Figure 1.1   Graphical representation of a comparator program.

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.

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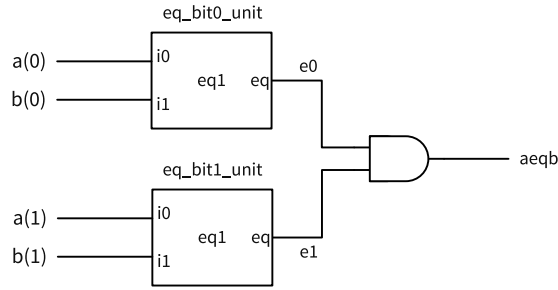
Figure 1.2   Construction of a 2-bit comparator from 1-bit comparators.

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.

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 a *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., in 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 in a single HDL framework.

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
```

1.4   TOP-LEVEL SIGNAL MAPPING

When an HDL program is targeted to a physical device of a prototyping board,
the design is subject to a variety of *constraints*. One constraint is the locations of
the I/O pins. For example, the switches and LEDs of the board are "pre-wired"
to specific I/O pins of the FPGA device and they cannot be altered. The pin
assignment is defined in a *constraint file*, which is processed in conjunction with
HDL files.

The designs of this book use a constraint file that specifies the pin assignment
for all the I/O signals on the Nexys 4 DDR prototyping board. To use this file,
the top-level HDL module must have the same predefined I/O signal names. This
can be achieved by creating an HDL file to "wrap" the original design and map
its original I/O signals to the prototyping board's I/O signals. For example, we
name the I/O pins connected to the slide switches and LEDs as `sw` and `led` and
specify their pin assignment in the constraint file. For a physical implementation,
the `a` and `b` signals of the previous comparator circuit can be connected to the four
switches and the output, `aeqb`, can be connected to an LED. The corresponding
wrapping code is shown in Listing 1.5.

Listing 1.5   Top-level wrapping circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity eq_top is
   port(
      sw  : in  std_logic_vector(3 downto 0); -- 4 switches
      led : out std_logic_vector(0 downto 0)  -- 1 red LED
   );
end eq_top;

architecture struc_arch of eq_top is
begin
   -- instantiate 2-bit comparator
   eq2_unit : entity work.eq2(struc_arch)
      port map(
```
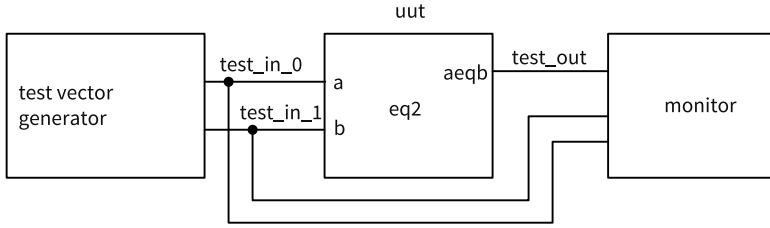
Figure 1.3 Testbench for a 2-bit comparator.

```
        a    => sw(3 downto 2),
        b    => sw(1 downto 0),
        aeqb => led(0)
    );
end struc_arch;
```

The code essentially maps the "logical" port names of the comparator to the physical signals on the prototyping board. Note that the output `led` signal is defined as a one-element vector to accommodate future expansion. The procedure to include the constraint file is demonstrated in Appendix A.2.

1.5   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 1.6.

Listing 1.6 Testbench for a 2-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq2_tb is
end eq2_tb;

architecture tb_arch of eq2_tb 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;
      -- terminate simulation
      assert false
         report "Simulation Completed"
         severity failure;
   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 1.4. 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.
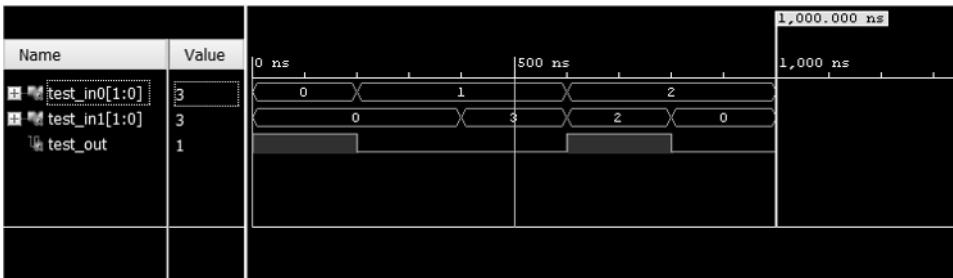
Figure 1.4   Simulated waveforms.

1.6   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.

1.7   SUGGESTED EXPERIMENTS

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

1.7.1   Code for gate-level greater-than circuit

Develop the HDL codes in Experiment 2.6.1. The code can be simulated and synthesized after we complete Chapter 2.

1.7.2   Code for gate-level binary decoder

Develop the HDL codes in Experiment 2.6.2. The code can be simulated and synthesized after we complete Chapter 2.