PART I

BASIC DIGITAL CIRCUITS

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

1.1 INTRODUCTION

Verilog is a hardware description language. It was developed in the mid-1980s and later transferred to the IEEE (Institute of Electrical and Electronics Engineers). The language is formally defined by IEEE Standard 1364. The standard was ratified in 1995 (referred to as Verilog-1995) and revised in 2001 (referred to as Verilog-2001). Many useful enhancements are added in the revised version. We use Verilog-2001 in this book.

Verilog 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 Verilog, we introduce the key Verilog synthesis constructs by examining a collection of examples. Several advanced topics are examined further in Chapter 7 and detailed Verilog coverage may be explored through the sources listed in the bibliographic section at the end of the chapter.

Although the syntax of Verilog is somewhat like that of the C language, its semantics (i.e., "meaning") is based on concurrent hardware operation and is totally different from the sequential execution of C. The subtlety of some language constructs and certain inherent non-deterministic behavior of Verilog can lead to difficult-to-detect errors and introduce a discrepancy between simulation and synthesis. The coding of this book follows a "better-safe-than-buggy" philosophy. Instead of writing quick and short codes, the focus is on style and constructs that are clear and synthesizable and can accurately describe the desired hardware.

Table 1.1 Truth table of 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 use a simple comparator to illustrate the skeleton of a Verilog program. The description uses only logic operators and represents a gate-level combinational circuit, which is composed of simple logic gates. In Chapter 3, we cover the remaining Verilog operators and constructs and examine the register-transfer-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 Verilog 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

```
module eq1
   // I/O ports
   (
    input wire i0, i1,
    output wire eq
   );

   // signal declaration
   wire p0, p1;

   // body
   // sum of two product terms
   assign eq = p0 | p1;
   // product terms
   assign p0 = ~i0 & ~i1;
   assign p1 = i0 & i1;

endmodule
```

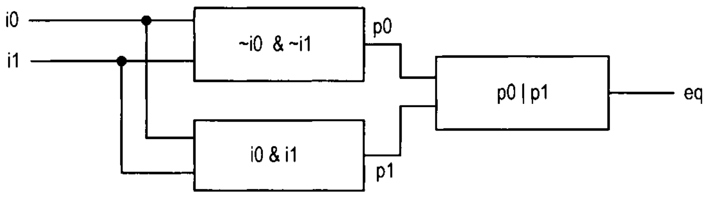

Figure 1.1 Graphical representation of a comparator program.

The best way to understand an HDL (hardware description language) program is to think in terms of hardware circuits. This program consists of three portions. The I/O port portion describes the input and output ports of this circuit, which are i0 and i1, and eq, respectively. The signal declaration portion specifies the internal connecting signals, which are p0 and p1. The body portion describes the internal organization of the circuit. There are three continuous assignments in this code. Each can be thought of as a circuit part that performs certain simple logical operations. We examine the language constructs and statements of this code in the next section.

The graphical representation of this program is shown in Figure 1.1. The three continuous assignments constitute the three circuit parts. The connections among these parts are specified implicitly by the signal and port names.

1.3 BASIC LEXICAL ELEMENTS AND DATA TYPES

1.3.1 Lexical elements

Identifier An *identifier* gives a unique name to an object, such as eq1, i0, or p0. It is composed of letters, digits, the underscore character (_), and the dollar sign ($). $ is usually used with a system task or function.

The first character of an identifier must be a letter or underscore. It is a good practice to give an object a descriptive name. For example, mem_addr_en is more meaningful than mae for a memory address enable signal.

Verilog is a *case-sensitive language*. Thus, data_bus, Data_bus, and DATA_BUS refer to three different objects. To avoid confusion, we should refrain from using the case to create different identifiers.

Keywords *Keywords* are predefined identifiers that are used to describe language constructs. In this book, we use boldface type for Verilog keywords, such as **module** and **wire** in Listing 1.1.

White space White space, which includes the space, tab, and newline characters, is used to separate identifiers and can be used freely in the Verilog code. We can use proper white spaces to format the code and make it more readable.

Comments A *comment* is just for documentation purposes and will be ignored by software. Verilog has two forms of comments. A one-line comment starts with //, as in

```
// This is a comment.
```

A multiple-line comment is encapsulated between /* and */, as in

```
/* This is comment line 1.
   This is comment line 2.
   This is comment line 3.   */
```

In this book, we use italic type for comments, as in the examples above.

1.4 DATA TYPES

1.4.1 Four-value system

Four basic values are used in most data types:

- 0: for "logic 0", or a false condition
- 1: for "logic 1", or a true condition
- z: for the high-impedance state
- x: for an unknown value

The z value corresponds to the output of a tri-state buffer. The x value is usually used in modeling and simulation, representing a value that is not 0, 1, or z, such as an uninitialized input or output conflict.

1.4.2 Data type groups

Verilog has two main groups of data types: *net* and *variable*.

Net group The data types in the net group represent the physical connections between hardware components. They are used as the outputs of *continuous assignments* and as the connection signals between different modules. The most commonly used data type in this group is **wire**. As the name indicates, it represents a connecting wire.

The **wire** data type represents a 1-bit signal, as in

```
wire p0, p1;    // two 1-bit signals
```

When a collection of signals is grouped into a bus, we can represent it using a one-dimensional array (vector), as in

```
wire [7:0] data1, data2; // 8-bit data
wire [31:0] addr;        // 32-bit address
wire [0:7] revers_data;  // ascending index should be avoided
```

While the index range can be either descending (as in [7:0]) or ascending (as in [0:7]), the former is preferred since the leftmost position (i.e., 7) corresponds to the MSB of a binary number.

A two-dimensional array is sometimes needed to represent a memory. For example, a 32-by-4 memory (i.e., a memory has 32 words and each word is 4 bits wide) can be represented as

```
wire [3:0] mem1 [31:0]; // 32-by-4 memory
```

The other data types in the net group imply certain logical behavior or functionality, such as **wand** (for wired-and connection) and **supply0** (for circuit ground connection). We don't use these data types in this book. Verilog-2001 also allows the **signed** data type and this issue is discussed in Section 7.3.

Variable group The data types in the variable group represent abstract storage in behavioral modeling and are used in the outputs of *procedural assignments*. There are five data types in this group: **reg**, **integer**, **real**, **time**, and **realtime**. The most commonly used data type in this group is **reg** and it can be synthesized. The inferred circuit may or *may not* contain physical storage components. The last three data types can only be used in modeling and simulation, and the use of the **integer** data type is discussed in Section 7.3.

In Verilog-1995, the variable group is known as the *register group*. Since this term is the same for a physical hardware register (i.e., a collection of flip-flops), it is changed in the Verilog-2001 documentation to avoid confusion. In this book, we use the term *variable* for the data type, and use the term *register* for the physical register circuit. **FYI**

1.4.3 Number representation

An integer constant in Verilog can be represented in various formats. Its general form is

```
[sign][size]'[base][value]
```

The `[base]` term specifies the base of the number, which can be the following:

- `b` or `B`: binary
- `o` or `O`: octal
- `h` or `H`: hexadecimal
- `d` or `D`: decimal

The `[value]` term specifies the value of the number in the corresponding base. The underline character (_) can be included for clarity.

The `[size]` term specifies the number of bits in a number. It is optional. The number is known as a *sized number* when a `[size]` term exists and is known as an *unsized number* otherwise.

Sized number A sized number specifies the number of bits explicitly. If the size of the value is smaller than the `[size]` term specified, zeros are padded in front to extend the number, except in several special cases. The `z` or `x` value is padded if the MSB of the value is `z` or `x`, and the MSB is padded if the **signed** data type is used. Several sized number examples are shown in the top portion of Table 1.2.

Unsized number An unsized number omits the `[size]` term. Its actual size depends on the host computer but must be at least 32 bits. The `'[base]` term can also be omitted if the number is in decimal format. Assume that 32 bits are used in the host machine. Several unsized number examples are shown in the bottom portion of Table 1.2.

1.4.4 Operators

Verilog has about two dozen operators. For the gate-level description, we need only the following bitwise operators: `~` (not), `&` (and), `|` (or), and `^` (xor). These operators infer basic gate-level cells. Other operators are discussed in Section 3.2.

1.5 PROGRAM SKELETON

As its name indicates, HDL is used to describe hardware. When we develop or examine a Verilog code, it is much easier to comprehend if we think in terms of "hardware organization"

Table 1.2 Examples of sized and unsized numbers

number	stored value	comment
5'b11010	11010	
5'b11_010	11010	_ ignored
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	00000000000000000000000000011010	extended to 32 bits
'hee	00000000000000000000000011101110	extended to 32 bits
1	00000000000000000000000000000001	extended to 32 bits
-1	11111111111111111111111111111111	extended to 32 bits

rather than "sequential algorithm." Most Verilog codes in this book follow the basic skeleton shown in Listing 1.1. It consists of three portions: I/O port declaration, signal declaration, and module body.

1.5.1 Port declaration

The module declaration and port declaration of Listing 1.1 are

```
module eq1
   (
    input wire i0, i1,
    output wire eq
   );
```

The I/O declaration specifies the modes, data types, and names of the module's I/O ports. The simplified syntax is

```
module [module_name]
   (
    [mode] [data_type] [port_names],
    [mode] [data_type] [port_names],
    . . .
    [mode] [data_type] [port_names]
   );
```

The `[mode]` term can be **input**, **output**, or **inout**, which represent the input, output, or bidirectional port, respectively. Note that there is no comma in the last declaration. The `[data_type]` term can be omitted if it is **wire**.

FYI ***Verilog-1995 port declaration*** In Verilog-1995, port names, modes, and data types are declared separately. For example, the preceding port declaration becomes

```
module eq1 (i0, i1, eq);   // only port names in brackets
   // declare mode
   input i0, i1;
   output eq;
   // declare data type
   wire i0, i1;
   wire eq;
```

We do not use this format in this book.

1.5.2 Program body

Unlike a program in the C language, in which the statements are executed sequentially, the program body of a synthesizable Verilog module can be thought of as a collection of circuit parts. These parts are operated in parallel and executed concurrently. There are several ways to describe a part:

- Continuous assignment
- "Always block"
- Module instantiation

The first way to describe a circuit part is by using a *continuous assignment*. It is useful for simple combinational circuits. Its simplified syntax is

```
assign [signal_name] = [expression];
```

Each continuous assignment can be thought as a circuit part. The signal on the left-hand side is the output and the signals used in the right-hand-side expression are the inputs. The expression describes the function of this circuit. For example, consider the statement

```
assign eq = p0 | 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 propagation delay. There are three continuous assignments in Listing 1.1 and they correspond to the three circuit parts shown in Figure 1.1. Since the assignments correspond to the circuit parts, the order of these statements does not matter.

The second way to describe a circuit part is by using an *always block*. More abstract *procedural assignments* are used inside the always block and thus it can be used to describe more complex circuit operation. The always block is discussed in Section 3.3.

The third way to describe a circuit part is by using *module instantiation*. Instantiation creates an instance of another module and allows us to incorporate predesigned modules as subsystems of the current module. Instantiation is discussed in Section 1.6.

1.5.3 Signal declaration

The declaration portion specifies the internal signals and parameters used in the module. The internal signals can be thought of as the interconnecting wires between the circuit parts, as shown in Figure 1.1.

The simplified syntax of signal declaration is

```
[data_type] [port_names];
```

Two internal signals are declared in Listing 1.1:

```
wire p0, p1;
```

FYI

Implicit net In Verilog, an identifier does not need to be declared explicitly. If a declaration is omitted, it is assumed to be an *implicit net*. The default data type is **wire**. We can remove the explicit declarations in Listing 1.1 and the simplified code is shown in Listing 1.2.

Listing 1.2 Code with implicit net

```
module eq1_implicit
   (
    input i0, i1,   // no data type declaration
    output eq
   );

// no internal signal declaration

// product terms must be placed in front
assign p0 = ~i0 & ~i1;    //implicit declaration
assign p1 = i0 & i1;      //implicit declaration
// sum of two product terms
assign eq = p0 | p1;

endmodule
```

Although the code is more compact, it may introduce subtle errors of misspelled identifiers. For clarity and documentation, we always use explicit declarations in this book.

1.5.4 Another example

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 1.3.

Listing 1.3 Gate-level implementation of a 2-bit comparator

```
module eq2_sop
   (
   input wire[1:0] a, b,
   output wire aeqb
   );

   // internal signal declaration
   wire p0, p1, p2, p3;

   // sum of product terms
   assign aeqb = p0 | p1 | p2 | p3;
   // product terms
   assign p0 = (~a[1] & ~b[1]) & (~a[0] & ~b[0]);
   assign p1 = (~a[1] & ~b[1]) & (a[0] & b[0]);
   assign p2 = (a[1] & b[1]) & (~a[0] & ~b[0]);
   assign p3 = (a[1] & b[1]) & (a[0] & b[0]);

endmodule
```

The a and b ports are now declared as a two-element array. Derivation of the architecture body is similar to that of the 1-bit comparator. The p0, p1, p2, and p3 signals represent

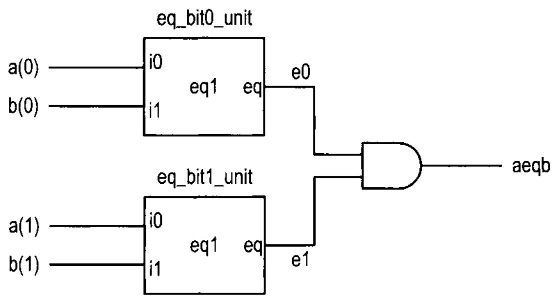

Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

the results of the four product terms, and the final result, aeqb, is the logic expression in sum-of-products format.

1.6 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. Verilog provides a mechanism, known as *module 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 1.5.4 is to utilize 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 both bits are equal. The corresponding code is shown in Listing 1.4.

Listing 1.4 Structural description of a 2-bit comparator

```
module eq2
   (
    input  wire[1:0] a, b,
    output wire aeqb
   );

   // internal signal declaration
   wire e0, e1;

   // body
   // instantiate two 1-bit comparators
   eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
   eq1 eq_bit1_unit (.eq(e1), .i0(a[1]), .i1(b[1]));

   // a and b are equal if individual bits are equal
   assign aeqb = e0 & e1;

endmodule
```

The code includes two module instantiation statements. The simplified syntax of module instantiation is

```
[module_name] [instance_name]
  (
   .[port_name]([signal_name]),
   .[port_name]([signal_name]),
   . . .
   );
```

The first portion of the statement specifies which component is used. The [module_name] term indicates the name of the module and the [instance_name] term gives a unique id for an instance. The second portion is port connection, which indicates the connections between the I/O ports of an instantiated module (the lower-level module) and the external signals used in the current module (the higher-level module). This form of mapping is known as *connection by name*. The order of the port-name and signal-name pairs does not matter.

In Listing 1.4, the first component instantiation statement is

```
eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
```

The eq1 is the module name defined in Listing 1.1. The port mapping reflects the connections shown in Figure 1.2. The component instantiation statement 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 the diagram, it puts all representations into a single HDL framework.

Xilinx specific

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.

FYI

Connection by ordered list An alternative scheme to associate the ports and external signals is *connection by ordered list* (sometimes also known as *connection by position*). In this scheme, the port names of the lower-level module are omitted and the signals of the higher-level module are listed in the same order as the lower-level module's port declaration. With this scheme, the two module instantiation statements in Listing 1.4 can be rewritten as

```
eq1 eq_bit0_unit (a[0], b[0], e0);
eq1 eq_bit1_unit (a[1], b[1], e1);
```

Although this scheme makes the code more compact, it is error prone, especially for a module with many I/O ports. For example, if we modify the code of the lower-level module and switch the order of two ports in the port declaration, all the instantiated modules need to be corrected as well. If this is done accidentally during code editing, the altered port order may be left undetected during synthesis and leads to difficult-to-find bugs. We always use the connection-by-name scheme in this book.

FYI

Verilog primitive Verilog includes a set of predefined *primitives* that can be instantiated as modules. These primitives correspond to simple gate-level function blocks, such as the *and*, *or*, and *not cells*. For example, the eq1 circuit can be implemented by using simple cells, as shown in Figure 1.3. The corresponding primitive-based code is shown in Listing 1.5.

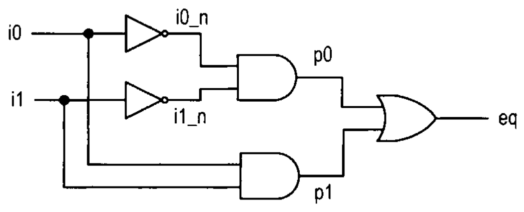

Figure 1.3 Low-level diagram of a 1-bit comparator.

Listing 1.5 Implementation with Verilog primitive

```
module eq1_primitive
  (
   input wire i0, i1,
   output wire eq
  );

  // internal signal declaration
  wire i0_n, i1_n, p0, p1;

  //primitive gate instantiations
  not unit1 (i0_n, i0);        // i0_n = ~i0;
  not unit2 (i1_n, i1);        // i1_n = ~i1;
  and unit3 (p0, i0_n, i1_n); // p0 = i0_n & i1_n;
  and unit4 (p1, i0, i1);      // p1 = i0 & i1;
  or unit5 (eq, p0, p1);       // eq = p0 | p1;

endmodule
```

This form of code is very tedious and can easily be replaced with simple bitwise logical operators. We do not use primitives in this book.

In addition to the predefined primitives, we can also define customized primitives, known as *user-defined primitives* (UDPs). For example, we can define a 1-bit comparator circuit in a UDP, as shown in Listing 1.6.

Listing 1.6 UDP of a 1-bit comparator

```
primitive eq1_udp(eq, i0, i1);
   output eq;
   input i0, i1;

  table
  // i0 i1 : eq
      0  0  : 1;
      0  1  : 0;
      1  0  : 0;
      1  1  : 1;
  endtable

endprimitive
```

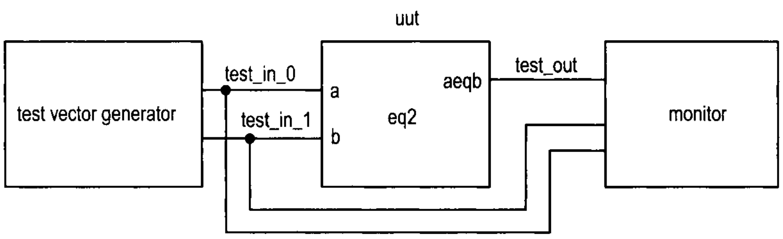

Figure 1.4 Testbench for a 2-bit comparator.

A UPD is essentially a table-based description of a circuit. The same table can also be described by a case statement (discussed in Section 3.5). We use the latter approach and do not use UDPs in this book.

1.7 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 is shown in Figure 1.4. 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 1.7.

Listing 1.7 Testbench for a 2-bit comparator

```
// The 'timescale directive specifies that
// the simulation time unit is 1 ns and
// the simulation timestep is 10 ps
`timescale 1 ns/10 ps

module eq2_testbench;
   // signal declaration
   reg  [1:0] test_in0, test_in1;
   wire  test_out;

   // instantiate the circuit under test
   eq2 uut
      (.a(test_in0), .b(test_in1), .aeqb(test_out));

   //  test vector generator
   initial
   begin
      // test vector 1
      test_in0 = 2'b00;
      test_in1 = 2'b00;
      # 200;
      // test vector 2
      test_in0 = 2'b01;
      test_in1 = 2'b00;
      # 200;
      // test vector 3
      test_in0 = 2'b01;
      test_in1 = 2'b11;
      # 200;
      // test vector 4
      test_in0 = 2'b10;
      test_in1 = 2'b10;
      # 200;
      // test vector 5
      test_in0 = 2'b10;
      test_in1 = 2'b00;
      # 200;
      // test vector 6
      test_in0 = 2'b11;
      test_in1 = 2'b11;
      # 200;
      // test vector 7
      test_in0 = 2'b11;
      test_in1 = 2'b01;
      # 200;
      // stop simulation
      $stop;
   end

endmodule
```

The code consists of a module instantiation statement, which creates an instance of the 2-bit comparator, and an *initial block*, which generates a sequence of test patterns. The initial block is a special Verilog construct, which is executed once when simulation starts. The statements inside an initial block are executed sequentially. Each test pattern is generated by three statements, as in

```
      // test vector 2
      test_in0 = 2'b01;
      test_in1 = 2'b00;
      # 200;
```

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 time units. The last statement, **$stop**, is a Verilog system function that stops the simulation and returns the control to simulation software.

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 Verilog. For now, 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. We provide a review of additional modeling and simulation-related language constructs and demonstrate the construction of a more sophisticated testbench in Section 7.5.

1.8 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.

Verilog is a complex language. The standard is specified in *IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001*. *Verilog HDL, 2nd edition,* by S. Palnitkar and *Starter's Guide to Verilog 2001* by M. D. Ciletti provide detailed coverage of the language's syntax and constructs. Verilog-2001 includes many improvements over the old standard. The article "The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It" by S. Sutherland summarizes the new features. 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.

1.9 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.

1.9.1 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.

1.9.2 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.