PART I

BASIC DIGITAL CIRCUITS DEVELOPMENT

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

HDL (hardware description language) is used to describe and model digital systems. SystemVerilog is one of the major HDLs. In this chapter, we use a simple comparator to illustrate the skeleton of a SystemVerilog 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 remaining operators and constructs and examine the register-transfer-level combinational circuits, which are composed of intermediate-sized components, such as adders, comparators, and multiplexers.

1.1 INTRODUCTION

1.1.1 Brief history of Verilog and SystemVerilog

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 and the document is known as the *LRM* (*Language Reference Manual*). The standard was ratified in 1995 (known as Verilog-1995) and significantly revised in 2001 (known as Verilog-2001). A further revision, which contains a few minor changes, was published in 2005. Unless otherwise specified, the term "Verilog" used in the book is referred to Verilog-2001.

Verilog was developed for gate-level and register-transfer-level design and modeling and it did not include advanced high-level verification features, such as assertions, functional coverage, and constrained random testing. SystemVerilog first served as an *extension of Verilog* that supports the verification features. The extension was ratified by IEEE in 2005 and formally defined by IEEE Standard 1800. It is referred to as SystemVerilog-2005.

In 2009, Verilog and SystemVerilog were combined into a single standard and defined by IEEE Standard 1800. The merged languages are called SystemVerilog and referred to as SystemVerilog-2009. The merge and name selection implies that Verilog is now part of SystemVerilog and the Verilog language has ceased to exist.

The merge and naming scheme may cause some confusion. SystemVerilog-2005 is a pure hardware verification language but the newer SystemVerilog (SystemVerilog-2009 and beyond) is a *hardware description and verification language* that incorporates both design and verification features into a single framework.

Unless otherwise specified, the term "SystemVerilog" used in the book is referred to SystemVerilog-2009, which includes hardware description portion and is a "superset" of the original Verilog.

1.1.2 Book coverage

SystemVerilog is an extremely complex language. Only a small subset of the language constructs is intended to describe gate-level and register-transfer-level systems and even a smaller subset can be recognized by the synthesis software tool and transformed into physical hardware.

The focus of this book is on hardware design rather than on the language. We introduce the key SystemVerilog synthesis constructs by examining a collection of examples. Although the syntax of SystemVerilog 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 nondeterministic behavior of SystemVerilog can lead to difficult-to-detect errors and can 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. The illustration of the covered language subset is shown in Figure 1.1. Several advanced synthesis related topics are examined further in Chapter 8 and more detailed SystemVerilog coverage may be explored through the sources listed in the bibliographic section at the end of the chapter.

Besides merging the two standards, SystemVerilog-2009 made many enhancements in the "hardware description portion" of the original Verilog-2001 standard.

Verilog FYI

We use some of these new features in the book. The book occasionally includes paragraphs to explain the difference between a new SystemVerilog-2009 feature and the original Verilog-2001 construct. They are highlighted by a **Verilog FYI** side bar, as shown at left. The main purpose of these paragraphs is to help the reader understand the older Verilog codes. Note that SystemVerilog is backward-compatible with Verilog-2001 and thus these codes can be accepted by the SystemVerilog synthesis tool as well. The paragraphs with side bars can be skipped without affecting the subsequent reading.

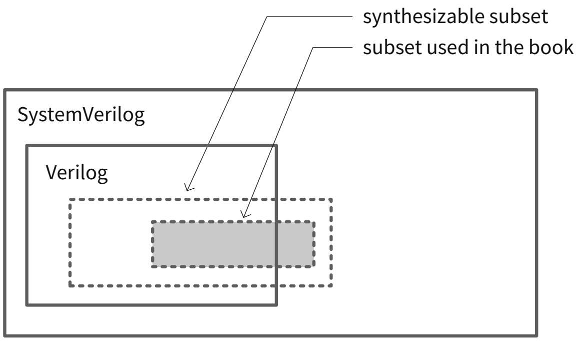

Figure 1.1 Subset covered in the book.

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

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.

Suppose 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 SystemVerilog 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 logic i0, i1,
    output logic eq
   );

   // signal declaration
   logic p0, p1;

   // body
   // sum of two product terms
   assign eq = p0 | p1;
```

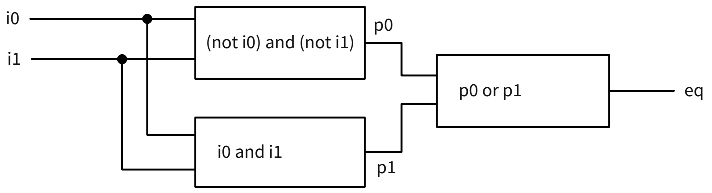

Figure 1.2 Graphical representation of a comparator program.

```
   // product terms
   assign p0 = ~i0 & ~i1;
   assign p1 = i0 & i1;
endmodule
```

The best way to understand an HDL 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 two sections.

The graphical representation of this program is shown in Figure 1.2. The three continuous assignments constitute the three circuit parts. The connections among these parts are specified implicitly by the signal and port names. The order of the continuous statements is clearly irrelevant and the three statements can be rearranged arbitrarily.

1.3 BASIC LEXICAL ELEMENTS AND DATA TYPES

1.3.1 Lexical elements

The basic SystemVerilog lexical elements include identifiers, keyword, white space, and comment.

Identifier An *identifier* gives a unique name to an object, such as `eq`, `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.

SystemVerilog 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.

Keyword A *Keyword* is a predefined identifier that is used to describe language constructs. In this book, we use boldface type for SystemVerilog keywords, such as **module** and **logic** 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 SystemVerilog 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. SystemVerilog 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.

1.3.2 Data types used in the book

SystemVerilog supports a rich collection of data types. However, we only use a very small restricted set in the book to describe the circuit. The set consists of the following:

1. the **logic** type
2. the **integer** type
3. the **tri** type
4. the user-defined enumerate type

The **logic** type is the most commonly used data type in design. It represents the value of a one-bit signal or the content of a one-bit memory element. The **logic** type can assume a value from a *four-state set*:

- `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 tristate 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.

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

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

The one-dimensional array can be interpreted as a collection of independent bits or an unsigned binary number. 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 (most significant bit) of a binary number.

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

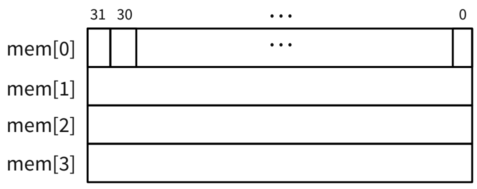

Figure 1.3 Illustration of a two-dimensional array.

```
logic [31:0] mem [0:3]; // 4-by-32 memory
```

Note that the outer dimension (i.e., `[0:3]`) is in ascending order, representing the memory module depicted in Figure 1.3.

The **integer** type is a special case of one-dimensional **logic** array. Its size is fixed at 32 bits and it is interpreted as a signed binary number. We use the **integer** type mainly for constants and parameters to represent threshold values, array boundaries, etc.

In our book, the **tri** type is only used to infer the tristate buffer of a bidirectional port and the user-defined enumerate type is used to represent the symbolic states of an FSM (finite state machine). These types are discussed in more detail in Sections 3.1.9 and 5.2.1.

1.3.3 Number representation

The value of a one-dimensional **logic** array is represented as a constant number. Its general format 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.

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.

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.

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

1.3.4 Operators

SystemVerilog has several dozens operators and only a subset of them can be synthesized. 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 3.1.

1.4 PROGRAM SKELETON

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

1.4.1 Port declaration

The module declaration and port declaration of Listing 1.1 are

```
module eq1
   (
    input logic i0, i1,
    output logic 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. Since the book focuses on design description, we only use the **logic** type for the input and output ports and use the **tri** type for bidirectional port

Verilog 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
    logic i0, i1;
    logic eq;
```

We do not use this format in this book.

1.4.2 Signal declaration

The declaration portion specifies the internal variables and local parameters used in the module. Since the variables frequently resemble the interconnecting wires between the circuit parts, as shown in Figure 1.2, we call them "signals" when appropriate.

The simplified syntax of signal declaration is

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

Two internal signals are declared in Listing 1.1:

```
    logic p0, p1;
```

Note that an identifier does not need to be declared explicitly. The previous declaration statement is actually optional. If a declaration is omitted, the signal is assumed to be an *implicit net*. 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.

1.4.3 Program body

The program body of a synthesizable SystemVerilog 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. There are three continuous assignments in Listing 1.1 and they correspond to the three circuit parts shown in Figure 1.2.

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 a more complex circuit operation. The always block is discussed in Section 3.2.

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 1.5.

1.4.4 Concurrent semantics

Although the "appearance" of an HDL program is somewhat like a traditional programming language, such as C, its semantics is very different. The statements in a C programs are run on a centralized processor and executed sequentially. The statements of an HDL program are "autonomous" and executed concurrently. For example, consider the statement

```
assign eq = p0 | p1;
```

It is executed as follows:

1. When a signal on the left-hand-side expression (i.e., `p0` or `p1`) changes, the statement is activated.
2. The left-hand-side expression (i.e., `p0 | p1`) is evaluated.
3. The evaluated result is passed to the right-hand signal after a delay (an implicit delta delay or an explicitly specified delay).
4. Repeat the process continuously.

Note that the execution resembles the operation of a circuit.

The continuous assignments can be activated at the same time and run concurrently. Its behavior is totally different from a C program statement. We intentionally put the assignment

```
assign eq = p0 | p1;
```

as the first line of the program body in Listing 1.1. The arrangement will lead to erroneous result in a traditional programming C language but has no effect on an HDL program since the order of the continuous assignments.

The execution of always block and component instantiation are more complex but can be reasoned in a similar way. In summary, the continuous assignment, always block, and module instantiation can be treated as "concurrent building constructs." Each construct *runs autonomously and continuously* and the overall operation of the code is executed in parallel.

1.4.5 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 1.2.

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

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

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

1.5 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. SystemVerilog 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 1.4.5 is to utilize previously constructed 1-bit comparators as the building blocks. The diagram is shown in Figure 1.4, 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 1.3.

Listing 1.3 Structural description of a 2-bit comparator

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

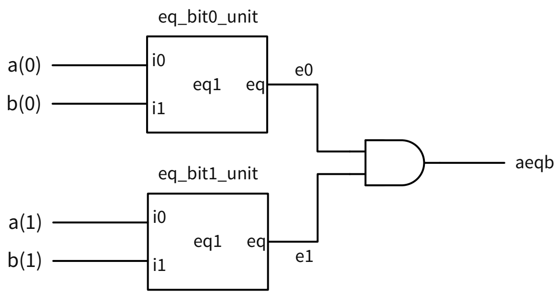

Figure 1.4 Construction of a 2-bit comparator from 1-bit comparators.

```
   // internal signal declaration
   logic 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 line 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 remaining 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 1.3, 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 1.4. 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.

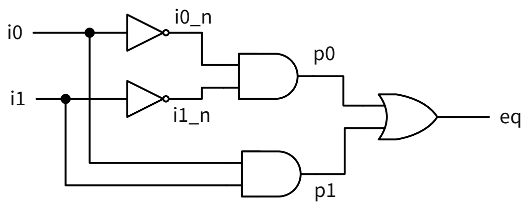

Figure 1.5 Low-level diagram of a 1-bit comparator.

The port names and signal names are sometimes identical and these mappings can be represented as "`.*`" in SystemVerilog. For example, the instantiation statement

```
eq1 eq_unit (.i0(i0), .i1(i1), .eq(eq));
```

can be abbreviated as

```
eq1 eq_unit (.*);
```

and the instantiation statement

```
eq1 eq_unit (.i0(i0), .i1(i1), .eq(result));
```

can be abbreviated as

```
eq1 eq_unit (.*, .eq(result));
```

Verilog 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 1.3 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 lead to difficult-to-find bugs. We always use the connection-by-name scheme in this book.

Verilog 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 1.5. The corresponding primitive-based code is shown in Listing 1.4.

Listing 1.4 Implementation with Verilog primitive

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

  // internal signal declaration
  logic 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 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 1.5. **Verilog FYI**

Listing 1.5 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
```

A UDP 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.

1.6 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

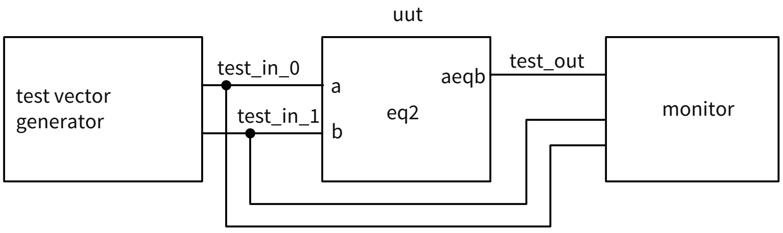

Figure 1.6 Testbench for a 2-bit comparator.

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 1.6.

Listing 1.6 Top-level wrapping circuit

```
module eq2_top
   (
    input logic [3:0] sw,
    output logic [0:0] led
   );

   // body
   // instantiate 2-bit comparator
   eq2 eq_unit (.a(sw[3:2]), .b(sw[1:0]), .aeqb(led[0]));
endmodule
```

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.

1.7 TESTBENCH

After code is developed, it can be *simulated* in a host computer to verify the correctness of the circuit operation and then *synthesized* to a physical device. Simulation is usually performed within the same language framework. We create a special program, known as a *testbench*, to mimic a physical lab bench.

The development of testbench and verification are beyond the scope of this book. We just provide several examples to illustrate the basic concepts. The templates can be used to simulate and observe inputs and outputs of a simple circuit. The sketch of a 2-bit comparator testbench is shown in Figure 1.6. The `uut` segment is the unit under test, the `test vector generator` segment generates testing input patterns, and the `monitor` segment 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
   logic [1:0] test_in0, test_in1;
   logic 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 language 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 the test vector 2:

```
test_in0 = 2'b01;
test_in1 = 2'b00;
# 200;
```

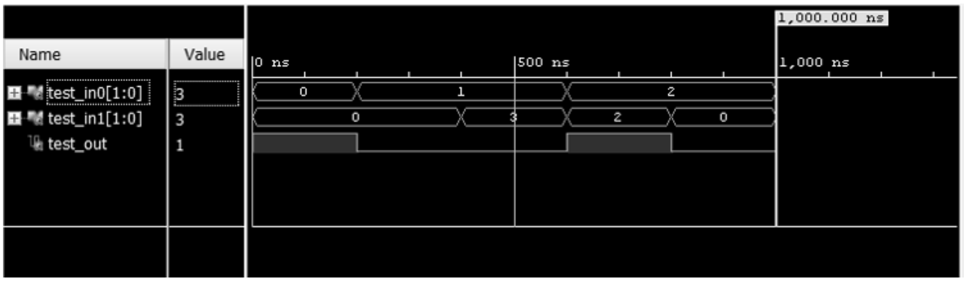

Figure 1.7 Simulated waveforms.

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 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 1.7.

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

1.8 BIBLIOGRAPHIC NOTES

In this book, a short bibliographic section is included in the end of each chapter to provide the most relevant references for further exploration. A more comprehensive bibliography can be found in the end of the book.

SystemVerilog is a very complex language. The standard is specified in *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, IEEE Std 1364-2001. Logic Design and Verification Using SystemVerilog,* by D. Thomas highlights the usage and capability of the language. *SystemVerilog for Design, second ed.* by S. Sutherland et al. and *SystemVerilog for Verification* by T. Fitzpatrick et al. provide detailed coverage on the design and modeling portion and the verification portion of the language, respectively. Derivation of the testbench for a large digital system is a difficult task. *Writing Testbenches Using SystemVerilog* 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 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 2.6.1. 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 2.6.2. The code can be simulated and synthesized after we complete Chapter 2.