For both built-in and user-defined primitives, `instance_name` is optional. Primitives can be instantiated only within modules. The first port of an instantiated primitive is always an output and the other ports are all inputs. The details of UDPs will be dealt with in Chapter 5.

1.2.7 Attributes

An attribute is a mechanism provided by Verilog HDL for annotating information about objects, statements and groups of statements in the source description to be used by various related tools. Almost all statements can be attributed in Verilog HDL. There are no standardized attributes. An attribute can be attached as a prefix to a declaration, a module item, a statement or a port connection and as a suffix to an operator or a function name in an expression. Attributes have the following syntax:

```
(*attr_spec {, attr_spec}*)
```

where `attr_spec` can be either `attr_name` or `attr_name = const_expr`. If an attribute is not assigned to a value, its value defaults to 1. The following are some examples:

```
// example 1: attach full_case attribute only
(* full_case=1, parallel_case = 0 *)
case (selection)
  <rest_of_case_statement>
endcase

// example 2: attach an attribute to a module
(* dont_touch *) module array_multiplier
     (x, y, product);
  <the body of array_multiplier>
endmodule

// example 3: attach an attribute to a reg variable
(* fsm_state *)   reg [1:0] state1;
(* fsm_state=1 *) reg [1:0] state2, state3;

// example 4: attach an attribute to a function call
x = add(* mode = "cla" *)(y, z);

// example 5: attach an attribute to an operator
x = y + (* mode = "cla" *) z;
```

Review Questions

Q1.6 How would you describe a bundle of signals in Verilog HDL when it is used to describe hardware modules?

Q1.7 How many possible values may a net or `reg` variable have?

Q1.8 How many data types are included in net data types? What are these?

Q1.9 How many data types are included in variable data types? What are these?

Q1.10 Define the following two terms: *scalar* and *vector*, in the sense of Verilog HDL.

Q1.11 What kinds of primitives are provided in Verilog HDL?

1.3 MODULE MODELING STYLES

In order to understand the design methodology of modern digital systems, it is instructive to distinguish among *design*, *model*, *synthesis*, *implementation* or *realization*. Design is a series of transformations from one representation of a system to another until a representation that can be fabricated exists; that is, it can be created, fashioned, executed or constructed according to a plan. Model is a process that converts a specification document into a HDL module. Model is a system of postulates, data and inferences presented as a mathematical description of an entity or the state of affair. Synthesis is a process that converts HDL modules into a structural representation. Synthesis can be divided into logic synthesis and high-level synthesis. Logic synthesis is a process that converts an RTL description into a gate-level netlist, while high-level synthesis is a process that converts a high-level description (i.e. specification) into RTL results. Implementation (or realization) is the process of transforming design abstraction into physical hardware components such as FPGAs or cell-based ICs (integrated circuits).

1.3.1 Modules

As mentioned previously, a Verilog HDL module consists of two major parts: the interface and the internal.

1.3.1.1 Modeling the Internal of a Module For each module in Verilog HDL, the internal (or body) can be modeled as one of the following styles:

1. *Structural style*. A design is described as a set of interconnected components. The components can be modules, UDPs, primitive gates and/or primitive switches.
   (a) Gate level. A design is said to be modeled at gate-level when it only comprises a set of interconnected gate primitives.
   (b) Switch level. A design is said to be modeled at switch-level when it only consists of a set of interconnected switch primitives.

   The structural style is a mechanism that can be used to construct a hierarchical design for a large digital system.
2. *Dataflow style*. The module is described by specifying the dataflow (i.e. data dependence) between registers and how the data is processed.
   (a) A module is specified as a set of continuous assignment statements.

3. *Behavioral* or *algorithmic style*. The design is described in terms of the desired design algorithm without concerning the hardware implementation details.
   (a) Designs can be described in any high-level programming languages.
4. *Mixed style*. The design is described in terms of the mixing use of above three modeling styles.
   (a) Mixed styles are most commonly used in modeling large designs.

In industry, the term *register-transfer level* (RTL) is often used to mean a structure that combines both behavioral and dataflow constructs and can be acceptable by logic synthesis tools.

1.3.1.2 Port Declaration The interface signals (including supply and ground) of any Verilog HDL module can be cast into one of the following three types:

(1) `input`: Declare a group of signals as input ports.
(2) `output`: Declare a group of signals as output ports.
(3) `inout`: Declare a group of signals as bidirectional ports; that is, they can be used as input or output ports but not at the same time.

The simplest way to describe the complete interface of a module is to divide it into three parts: port list, port declaration and data type declaration of each port. Usually, an output port, except that it is a net data type, must be declared a data type associated with it. However, input ports are often left with their data types undeclared. For example:

```
// port list style
module adder(x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output [3:0] sum;
output c_out;
reg [3:0] sum;
reg c_out;
```

This style of port declaration is known as *port list style*. The declaration of a port and its associated data type can be combined into a single line. Based on this idea, the above interface portion of the module can be rewritten as follows:

```
// port list style
module adder(x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output reg [3:0] sum;
output reg c_out;
```

Of course, port list, port declarations and their associated data types can also be put together into a single list. This style is often called *port list declaration style*. Hence, the above module interface can be rewritten as follows:

```
// port list declaration style
module adder(input [3:0] x, y,
             input c_in,
             output reg [3:0] sum,
             output reg c_out
); // sometimes called ANSI C style
```

This style is the same as that of ANSI C programming language so that we often name it as the ANSI C style. Note that all of the above interface styles are valid in Verilog HDL.

1.3.1.3 Port Connection Rules The port connection (also called port association) rules of Verilog HDL modules are consistent with those of actual hardware modules. That is, Verilog HDL allows ports to remain unconnected and with different sizes. In addition, unconnected inputs are driven to the "z" state; unconnected outputs are not used. Connecting ports to external signals can be done by one of the following two methods:

- *Named association*. The ports to be connected to external signals are specified by listing their names. The port order is not important.
- *Positional association*. The ports are connected to external signals by an ordered list. The signals to be connected must have the same order as the ports in the port list, leaving the unconnected port blank.

However, these two methods cannot be mixed in the same module. Moreover, Verilog HDL primitives (including both built-in and user-defined) can only be connected by positional association.

The operation to "call" (much like the *macro expansion* in assembly language) a built-in primitive, a user-defined primitive or the other module is called *instantiation* and each copy of the called primitive or module is called an *instance*. Figure 1.4 shows how to instantiate gate primitives and user-defined modules, as well as how to connect their ports through nets and input/output ports. The built-in primitives can only be connected by using positional association. They cannot be connected through named association. In addition, as mentioned before, the instance names of these primitives are optional.

The module `full_adder` depicted in Figure 1.4 shows how to instantiate an already defined module and how to connect their ports through nets. The user-defined modules can be connected by using either named association or positional association. In addition, the instance names of these instantiations are necessary.

Coding Styles

1. *A module cannot be declared within another module.*
2. *A module can instantiate other modules.*
3. *A module instantiation must have a module identifier (instance name) except for built-in primitives, gate and switch primitives and user-defined primitives (UDPs).*

```
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// -- half adder body-- //
// instantiate primitive gates
   xor xor1 (s, x, y);
   and and1 (c, x, y);
endmodule

module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1,c1,c2;  // outputs of both half adders
// -- full adder body-- //
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));
   or (cout, c1, c2);
endmodule
```

Can only be connected by using positional association

Instance name is optional

Connecting by using positional association

Connecting by using named association

Instance name is necessary

FIGURE 1.4 Port connection rules

4. *It should use named association at the top-level modules to avoid confusion that may arise from synthesis tools.*

1.3.2 Structural Modeling

As mentioned previously, the structural modeling of a design is by connecting required instantiations of built-in primitives, user-defined primitives or other (user-defined) modules through nets. The following example shows several examples of structural modeling:

EXAMPLE 1.4 ***An Example of Structural Modeling at Gate Level***

The `half_adder` instantiates two gate primitives and the `full_adder` instantiates two `half_adder` modules and one gate primitive. Finally, the `four_bit_adder` is constructed by four `full_adder` instances in turn.

```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// half adder body
// instantiate primitive gates
   xor (s,x,y);
   and (c,x,y);
endmodule
```



```
// gate-level description of full adder
module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1, c1, c2;  // outputs of both half adders
// full adder body
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (cin, s1, s, c2);
   or (cout, c1, c2);
endmodule

// gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input  [3:0] x, y;
input  c_in;
output [3:0] sum;
output c_out;
wire   c1, c2, c3; // intermediate carries
// four_bit adder body
// instantiate the full adder
   full_adder fa_1 (x[0], y[0], c_in, sum[0], c1);
   full_adder fa_2 (x[1], y[1], c1, sum[1], c2);
   full_adder fa_3 (x[2], y[2], c2, sum[2], c3);
   full_adder fa_4 (x[3], y[3], c3, sum[3], c_out);
endmodule
```

In fact, the structural style is one way to model a complex digital system in a hierarchical manner. An example of a 4-bit adder constructed in a hierarchical manner is depicted in Figure 1.5. Here, the 4-bit adder is composed of four full-adders and then each full-adder is built by basic logic gates in turn. Although this example is quite simple, it manifests several important features when designing a large digital system.

1.3.3 Dataflow Modeling

The essential structure used to model a design in the dataflow style is the continuous assignment. In a continuous assignment, a value is assigned onto a net. It must be a net because continuous assignments are used to model the behavior of combinational logic circuits. A continuous assignment starts with the keyword `assign` and has the syntax:

```
assign [delay] l_value = expression;
```

Anytime the value of an operand used in the `expression` changes, the `expression` is evaluated and the result is assigned to `l_value` after the specified `delay`. The

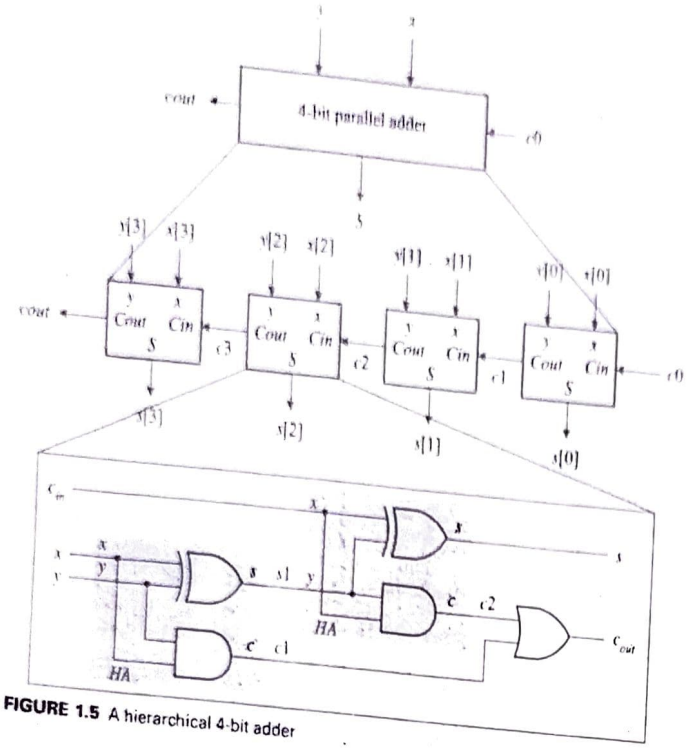

FIGURE 1.5 A hierarchical 4-bit adder

`delay` specifies the amount of time between a change of operand used in the expression and the assignment to `l_value`. If no `delay` is specified, the default is zero delay. Continuous assignments in a module execute concurrently regardless of the order they appear.

The following example illustrates how a continuous assignment is used to describe the 1-bit adder (i.e. full adder) depicted in Figure 1.6.

EXAMPLE 1.5 *A Full Adder Modeling in Dataflow Style*

In this example, we assume that the adder requires 5 time units to complete its operations. The delay will be ignored by the synthesis tools when the module is

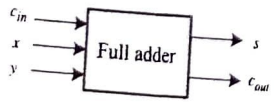

FIGURE 1.6 The block diagram of a full adder

synthesized because the delay will be replaced by the actual delays of the gates used to realize the adder.

```
module full_adder_dataflow(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;

// specify the function of a full adder
  assign #5 {c_out, sum} = x + y + c_in;
endmodule
```

■

1.3.4 Behavioral Modeling

The behavioral style uses the following two procedural constructs: `initial` and `always`. The `initial` statement can only be executed once and therefore is usually used to set up initial values of variable data types whereas the `always` statement, as the name implies, is executed repeatedly. The `always` statements are used to model combinational or sequential logic. Each `always` corresponds to a piece of logic.

The `l_value` used in an expression within an `initial` or `always` statement must be a variable data type, which retains its value until a new value is assigned. All `initial` statements and `always` statements begin their execution at simulation time 0 concurrently.

The following example illustrates how a procedural construct is used to describe the 1-bit adder (i.e. full adder) depicted in Figure 1.6.

EXAMPLE 1.6 *A Full Adder Modeling in Behavioral Style*

Basically, the expression used to describe the operations of a 1-bit full adder in behavioral style is the same as that of the dataflow style except that it needs to be put inside an `always` statement. In addition, the `@(x, y, c_in)` is used to sensitize the changes of input signals.

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;
reg    sum, c_out;  // sum and c_out need to be declared
                    //  as reg types.

// specify the function of a full adder
always @(x, y, c_in)// can also use always @(*) or
                    // always@(x or y or c_in)
  #5 {c_out, sum} = x + y + c_in;
endmodule
```

■

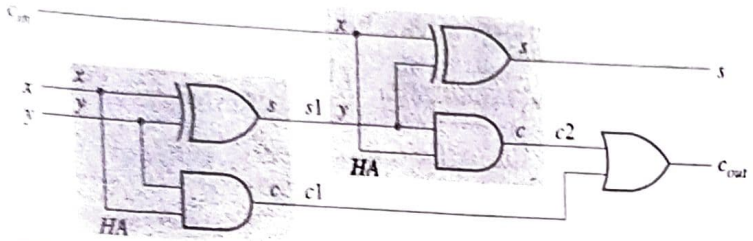

FIGURE 1.7 A full adder constructed with basic logic gates

1.3.5 Mixed-Style Modeling

As mentioned above, the mixed-style modeling is usually used to construct a hierarchical design in large systems. However, we are still able to model a simple design in mixed style. For example, as depicted in Figure 1.7 the full adder is constructed with two half adders and an OR gate. An example of the full adder being modeled in mixed style is shown in the following example.

EXAMPLE 1.7 *A Full Adder Module in Mixed Style*

The first half adder is modeled in structural style, the second half adder in dataflow style and the OR gate in behavioral style.

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input  x, y, c_in;
output s, c_out;
reg    c_out;
wire   s1, c1, c2;
// structural modeling of HA 1.
   xor xor_ha1 (s1, x, y);
   and and_ha1(c1, x, y);
// dataflow modeling of HA 2.
   assign s = c_in ^ s1;
   assign c2 = c_in & s1;
// behavioral modeling of output OR gate.
always @(c1, c2) // can also use always @(*)
   c_out = c1 | c2;
endmodule
```

■

Review Questions

Q1.12 What are the differences of design, model, synthesis and implementation (also called realization)?

Q1.13 Describe the features of structural style.

Q1.14 What is the basic statement used in dataflow style?

Q1.15 What are the basic statements used in behavioral style?

Q1.16 Can we write a module by mixing the use of various modeling styles?

1.4 SIMULATION

For a design to be useful, it must be verified so as to make sure that it can correctly operate according to the requirement. Verilog HDL not only provides capabilities to model a design, but also provides facilities to generate and control stimuli, monitor and store responses, and check the results. In this section, we use a 4-bit adder as an example to illustrate how to verify a design entirely through the mechanism provided by Verilog HDL.

1.4.1 Basic Simulation Constructs

Two basic simulation structures in Verilog HDL are shown in Figure 1.8. The first structure is to take the unit under test (UUT) as an instantiated module in the stimulus module. This is often used in simple or small projects since it is intuitively simple to write. The second construct considers both stimulus block and UUT as the separate instantiated module at the top-level module. This is suitable for large projects. In this book, we will use the first structure when writing a test bench.

In general, a test bench comprises several basic parts: an instantiation of the UUT, stimulus generation and control, response monitoring and storing, and result checking. Except for the instantiation of the UUT, the rest of the parts are usually modeled in behavioral style. In the rest of this section, we deal with these features of a typical test

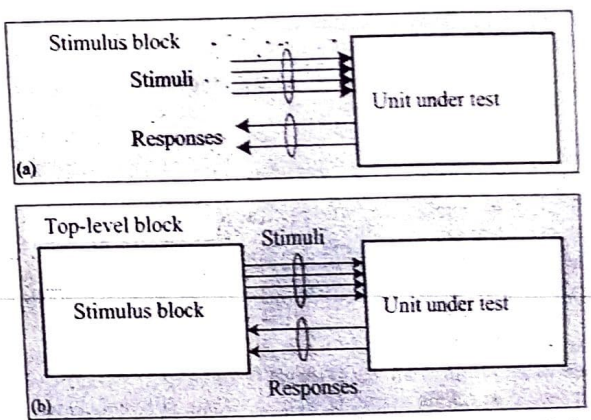

FIGURE 1.8 Two basic constructs for a simulation: (a) stimulus block at the top-level module; (b) stimulus block considered as a separate module