3

Structural Descriptions

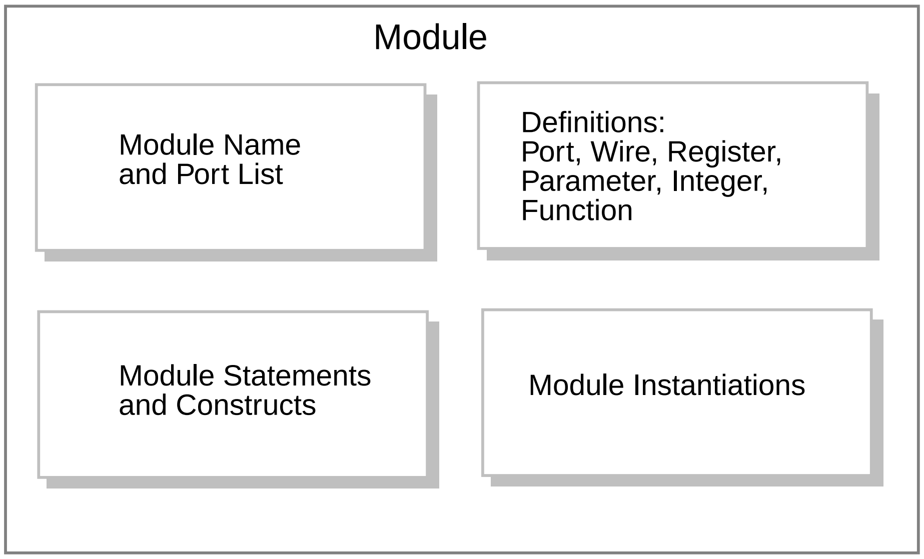
A Verilog structural description defines a connection of components that form a physical circuit. This chapter details the construction of structural descriptions, in the following major sections:

- Modules
- Macromodules
- Port Definitions
- Module Statements and Constructs
- Module Instantiations

Modules

The principal design entity in the Verilog language is the module. A module consists of the module name, its input and output description (port definition), a description of the functionality or implementation for the module (module statements and constructs), and named instantiations. Figure 3-1 illustrates the basic structural parts of a module.

Figure 3-1 Structural Parts of a Module


Module
Module Name and Port List
Definitions: Port, Wire, Register, Parameter, Integer, Function
Module Statements and Constructs
Module Instantiations

Example 3-1 shows a simple module that implements a 2-input NAND gate by instantiating an AND gate and an INV gate. The first line of the module definition gives the name of the module and a list of ports. The second and third lines give the direction for all ports. (Ports are either inputs, outputs, or bidirectionals.)

The fourth line of the description creates a `wire` variable. The next two lines instantiate the two components, creating copies named `instance1` and `instance2` of the components `AND` and `INV`. These components connect to the ports of the module and are finally connected by use of the variable `and_out`.

Example 3-1 Module Definition

```
module NAND(a,b,z);
    input a,b;     //Inputs to NAND gate
    output z;       //Outputs from NAND gate
    wire and_out; //Output from AND gate

    AND instance1(a,b,and_out);
    INV instance2(and_out, z);
endmodule
```

Macromodules

The macromodule construct makes simulation more efficient, by merging the macromodule definition with the definition of the calling (parent) module. However, HDL Compiler treats the macromodule construct as a module construct. Whether you use module or macromodule, the synthesis process, the hierarchy that synthesis creates, and its result are the same. Example 3-2 shows how to use the macromodule construct.

Example 3-2 Macromodule Construct

```
macromodule adder (in1,in2,out1);
    input [3:0] in1,in2;
    output [4:0] out1;

    assign out1 = in1 + in2;
endmodule
```

Note:

When Design Compiler instantiates a macromodule, a new level of hierarchy is created. To eliminate this new level of hierarchy, use the `ungroup` command. See the *Design Compiler User Guide* for information on the `ungroup` command.

Port Definitions

A port list consists of port expressions that describe the input and output interfaces for a module. Define the port list in parentheses after the module name, as shown here:

```
module name ( port_list );
```

A port expression in a port list can be any of the following:

- An identifier
- A single bit selected from a bit vector declared within the module
- A group of bits selected from a bit vector declared within the module
- A concatenation of any of the above

Concatenation is the process of combining several single-bit or multiple-bit operands into one large bit vector. For more information, see “Concatenation Operators” on page 4-13.

Declare each port in a port list as input, output, or bidirectional in the module by use of an `input`, `output`, or `inout` statement. (See “Concatenation Operators” on page 4-13.) For example, the module definition in Example 3-1 on page 3-3 shows that module NAND has three ports: a, b, and z, connected to 1-bit nets a, b, and z. Declare these connections in the `input` and `output` statements.

Port Names

Some port expressions are identifiers. If the port expression is an identifier, the port name is the same as the identifier. A port expression is not an identifier if the expression is a single bit, a group of bits selected from a vector of bits, or a concatenation of signals. In these cases, the port is unnamed unless you explicitly name it.

Example 3-3 shows some module definition fragments that illustrate the use of port names. The ports for module `ex1`, named a, b, and z, are connected to nets a, b, and z, respectively. The first two ports of module `ex2` are unnamed; the third port is named z. The ports are connected to nets `a[1]`, `a[0]`, and `z`, respectively. Module `ex3` has two ports: the first port, unnamed, is connected to a concatenation of nets a and b; the second port, named z, is connected to net z.

Example 3-3 Module Port Lists

```
module ex1( a, b, z );
    input a, b;
    output z;
endmodule

module ex2( a[1], a[0], z );
    input [1:0] a;
    output z;
endmodule

module ex3( {a,b}, z );
    input a,b;
    output z;
endmodule
```

Renaming Ports

You can rename a port by explicitly assigning a name to a port expression by using the dot (.) operator. The module definition fragments in Example 3-4 show how to rename ports. The ports for module `ex4` are explicitly named `in_a`, `in_b`, and `out` and are connected to nets `a`, `b`, and `z`. Module `ex5` shows ports named `i1`, `i0`, and `z` connected to nets `a[1]`, `a[0]`, and `z`, respectively. The first port for module `ex6` (the concatenation of nets a and b) is named `i`.

Example 3-4 Renaming Ports in Modules

```
module ex4( .in_a(a), .in_b(b), .out(z) );
    input a, b;
    output z;
endmodule

module ex5( .i1(a[1]), .i0(a[0]), z );
    input [1:0] a;
    output z;
endmodule

module ex6( .i({a,b}), z );
    input a,b;
    output z;
endmodule
```

Module Statements and Constructs

The Synopsys HDL Compiler tool recognizes the following Verilog statements and constructs when they are used in a Verilog module:

- `parameter` declarations
- `wire`, `wand`, `wor`, `tri`, `supply0`, and `supply1` declarations

- `reg` declarations
- `input` declarations
- `output` declarations
- `inout` declarations
- Continuous assignments
- Module instantiations
- Gate instantiations
- Function definitions
- always blocks
- task statements

Data declarations and assignments are described in this section. Module and gate instantiations are described in “Module Instantiations” on page 3-16. Function definitions, always blocks, and task statements are described in Chapter 5, “Functional Descriptions.”

Structural Data Types

Verilog structural data types include `wire`, `wand`, `wor`, `tri`, `supply0`, and `supply1`. Although `parameter` does not fall into the category of structural data types, it is presented here because it is used with structural data types.

You can define an optional range for all the data types presented in this section. The range provides a means for creating a bit vector. The syntax for a range specification is

```
[msb : lsb]
```

Expressions for most significant bit (`msb`) and least significant bit (`lsb`) must be nonnegative constant-valued expressions. Constant-valued expressions are composed only of constants, Verilog parameters, and operators.

parameter

Verilog parameters allow you to customize each instantiation of a module. By setting different values for the parameter when you instantiate the module, you can cause constructions of different logic. For more information, see “Parameterized Designs” on page 3-19.

A parameter represents constant values symbolically. The definition for a parameter consists of the parameter name and the value assigned to it. The value can be any constant-valued integer or Boolean expression. If you do not set the size of the parameter with a range definition or a sized constant, the parameter is unsized and defaults to a 32-bit quantity. See “Constant-Valued Expressions” on page 4-2 for a discussion of constant formats.

You can use a parameter wherever a number is allowed, except when declaring the number of bits in an assignment statement, which will generate a syntax error as shown in Example 3-5.

Example 3-5 parameter Declaration Syntax Error

```
parameter size = 4;
assign out = in ? 4’b0000 : size’b0101;  // syntax error
```

You can define a parameter anywhere within a module definition. However, the Verilog language requires that you define the parameter before you use it.

Example 3-6 shows two parameter declarations. Parameters true and false are unsized and have values of 1 and 0, respectively. Parameters S0, S1, S2, and S3 have values of 3, 1, 0, and 2, respectively, and are stored as 2-bit quantities.

Example 3-6 parameter Declarations

```
parameter TRUE=1, FALSE=0;
parameter [1:0] S0=3, S1=1, S2=0, S3=2;
```

wire

A `wire` data type in a Verilog description represents the physical wires in a circuit. A `wire` connects gate-level instantiations and module instantiations. The Verilog language allows you to read a value from a `wire` from within a function or a `begin...end` block, but you cannot assign a value to a `wire` within a function or a `begin...end` block. (An `always` block is a specific type of `begin...end` block.)

A `wire` does not store its value. It must be driven in one of two ways:

- By connecting the `wire` to the output of a gate or module
- By assigning a value to the `wire` in a continuous assignment

In the Verilog language, an undriven `wire` defaults to a value of `Z` (high impedance). However, HDL Compiler leaves undriven `wires` unconnected. Multiple connections or assignments to a `wire` simply short the `wires` together.

In Example 3-7, two `wires` are declared: a is a single-bit `wire`, and b is a 3-bit vector of `wires`. Its most significant bit (`msb`) has an index of 2, and its least significant bit (`lsb`) has an index of `0`.

Example 3-7 wire Declarations

```
wire a;
wire [2:0] b;
```

You can assign a delay value in a `wire` declaration, and you can use the Verilog keywords *scalared* and *vectored* for simulation. HDL Compiler accepts the syntax of these constructs, but they are ignored when the circuit is synthesized.

Note:

> You can use delay information for modeling, but Design Compiler ignores delay information. If the functionality of your circuit depends on the delay information, Design Compiler might create logic whose behavior does not agree with the behavior of the simulated circuit.

wand

The `wand` (`wired-AND`) data type is a specific type of wire.

In Example 3-8, two variables drive the variable `c`. The value of `c` is determined by the logical `AND` of `a` and `b`.

Example 3-8 wand (wired-AND)

```
module wand_test(a, b, c);
     input a, b;
     output c;

     wand c;

     assign c = a;
     assign c = b;
endmodule
```

You can assign a delay value in a `wand` declaration, and you can use the Verilog keywords *scalared* and *vectored* for simulation. HDL Compiler accepts the syntax of these constructs but ignores the constructs during synthesis of the circuit.

wor

The `wor` (`wired-OR`) data type is a specific type of wire.

In Example 3-9, two variables drive the variable `c`. The value of `c` is determined by the logical `OR` of `a` and `b`.

Example 3-9 wor (wired-OR)

```
module wor_test(a, b, c);
     input a, b;
     output c;

     wor c;

     assign c = a;
     assign c = b;
endmodule
```

tri

The `tri` (`three-state`) data type is a specific type of wire. All variables that drive the `tri` must have a value of `Z` (high-impedance), except one. This single variable determines the value of the `tri`.

Note:

HDL Compiler does not enforce the previous condition. You must ensure that no more than one variable driving a `tri` has a value other than `Z`.

In Example 3-10, three variables drive the variable `out`.

Example 3-10   tri (Three-State)

```
module tri_test (out, condition);
        input [1:0] condition;
        output out;

        reg a, b, c;
        tri out;

        always @ ( condition ) begin
             a = 1’bz;            //set all variables to Z
             b = 1’bz;
             c = 1’bz;
                  case ( condition ) //set only one variable to non-Z
                       2’b00 : a = 1’b1;
                       2’b01 : b = 1’b0;
                       2’b10 : c = 1’b1;
                  endcase
        end

        assign out = a;         //make the tri connection
        assign out = b;
        assign out = c;
endmodule
```

supply0 and supply1

The `supply0` and `supply1` data types define `wires` tied to logic `0` (`ground`) and logic `1` (`power`). Using `supply0` and `supply1` is the same as declaring a `wire` and assigning a `0` or a `1` to it. In Example 3-11, `power` is tied to logic `1` and `gnd` (`ground`) is tied to logic `0`.

Example 3-11 supply0 and supply1 Constructs

```
supply0 gnd;
supply1 power;
```

reg

A `reg` represents a variable in Verilog. A `reg` can be a 1-bit quantity or a vector of bits. For a vector of bits, the range indicates the most significant bit and least significant bit of the vector. Both must be nonnegative constants, parameters, or constant-valued expressions. Example 3-12 shows some `reg` declarations.

Example 3-12 reg Declarations

```
reg x;             //single bit
reg a,b,c;         //3 1-bit quantities
reg [7:0] q;       //an 8-bit vector
```

Port Declarations

You must explicitly declare the direction (input, output, or bidirectional) of each port that appears in the port list of a port definition. Use the `input`, `output`, and `inout` statements, as described in the following sections.

input

You declare all input ports of a module with an `input` statement. An input is a type of wire and is governed by the syntax of wire. You can use a range specification to declare an input that is a vector of signals, as in the case of input b in the following example. The `input` statements can appear in any order in the description, but you must declare them before using them. For example,

```
input a;
input [2:0] b;
```

output

You declare all output ports of a module with an `output` statement. Unless otherwise defined by a `reg`, `wand`, `wor`, or `tri` declaration, an output is a type of wire and is governed by the syntax of wire. An `output` statement can appear in any order in the description, but you must declare the statement before you use it.

You can use a range specification to declare an output that is a vector of signals. If you use a `reg` declaration for an output, the reg must have the same range as the vector of signals. For example,

```
output a;
output [2:0]b;
reg [2:0] b;
```

inout

You can declare bidirectional ports with the `inout` statement. An inout is a type of `wire` and is governed by the syntax of `wire`. HDL Compiler allows you to connect only `inout` ports to module or gate instantiations. You must declare an inout before you use it. For example,

```
inout a;
inout [2:0]b;
```

Continuous Assignment

If you want to drive a value onto a `wire`, `wand`, `wor`, or `tri`, use a continuous assignment to specify an expression for the `wire` value. You can specify a continuous assignment in two ways:

- Use an explicit continuous assignment statement after the `wire`, `wand`, `wor`, or `tri` declaration.
- Specify the continuous assignment in the same line as the declaration for a `wire`.

Example 3-13 shows two equivalent methods for specifying a continuous assignment for `wire a`.

Example 3-13 Two Equivalent Continuous Assignments

```
wire a;             //declare
assign a = b & c;   //assign
wire a = b & c;     //declare and assign
```

The left side of a continuous assignment can be

- A `wire`, `wand`, `wor`, or `tri`
- One or more bits selected from a vector
- A concatenation of any of these

The right side of the continuous assignment statement can be any supported Verilog operator or any arbitrary expression that uses previously declared variables and functions. You cannot assign a value to a `reg` in a continuous assignment.

Verilog allows you to assign drive strength for each continuous assignment statement. HDL Compiler accepts drive strength, but it does not affect the synthesis of the circuit. Keep this in mind when you use drive strength in your Verilog source.

Assignments are done bitwise, with the low bit on the right side assigned to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, operands on the right side are zero-extended.

Module Instantiations

Module instantiations are copies of the logic in a module that defines component interconnections.

```
module_name instance_name1 (terminal, terminal, ...),
            instance_name2 (terminal, terminal, ...);
```

A module instantiation consists of the name of the module (*module_name*) followed by one or more instantiations. An instantiation consists of an instantiation name (*instance_name*) and a connection list. A connection list is a list of expressions called terminals, separated by commas. These terminals are connected to the ports of the instantiated module. Module instantiations have this syntax:

```
(terminal1, terminal2, ...),
(terminal1, terminal2, ...);
```

Terminals connected to input ports can be any arbitrary expression. Terminals connected to output and inout ports can be identifiers, single- or multiple-bit slices of an array, or a concatenation of these. The bit-widths for a terminal and its module port must be the same.

If you use an undeclared variable as a terminal, the terminal is implicitly declared as a scalar (1-bit) wire. After the variable is implicitly declared as a wire, it can appear wherever a wire is allowed.

Example 3-14 shows the declaration for the module SEQ with two instantiations (SEQ_1 and SEQ_2).

Example 3-14 Module Instantiations

```
module SEQ(BUS0,BUS1,OUT); //description of module SEQ
     input BUS0, BUS1;
     output OUT;
     ...
endmodule

module top( D0, D1, D2, D3, OUT0, OUT1 );
     input D0, D1, D2, D3;
     output OUT0, OUT1;

     SEQ SEQ_1(D0,D1,OUT0),  //instantiations of module SEQ
          SEQ_2(.OUT(OUT1),.BUS1(D3),.BUS0(D2));
endmodule
```

Named and Positional Notation

Module instantiations can use either named or positional notation to specify the terminal connections.

In name-based module instantiation, you explicitly designate which port is connected to each terminal in the list. Undesignated ports in the module are unconnected.

In position-based module instantiation, you list the terminals and specify connections to the module according to each terminal's position in the list. The first terminal in the connection list is connected to the first module port, the second terminal to the second module port, and so on. Omitted terminals indicate that the corresponding port on the module is unconnected.

In Example 3-14, `SEQ_2` is instantiated by the use of named notation, as follows:

- Signal `OUT1` is connected to port `OUT` of the module `SEQ`.

- Signal D3 is connected to port BUS1.
- Signal D2 is connected to port BUS0.

SEQ_1 is instantiated by the use of positional notation, as follows:

- Signal D0 is connected to port BUS0 of module SEQ.
- Signal D1 is connected to port BUS1.
- Signal OUT0 is connected to port OUT.

Parameterized Designs

The Verilog language allows you to create parameterized designs by overriding parameter values in a module during instantiation. You can do this with the `defparam` statement or with the following syntax:

```
module_name #(parameter_value, parameter_value,...)
instance_name (terminal_list)
```

HDL Compiler does not support the `defparam` statement but does support the previous syntax.

The module in Example 3-15 contains a `parameter` declaration.

Example 3-15 parameter Declaration in a Module

```
module foo (a,b,c);

     parameter width = 8;

     input [width-1:0] a,b;
     output [width-1:0] c;

     assign c = a & b;

endmodule
```

In Example 3-15, the default value of the `parameter width` is 8, unless you override the value when the module is instantiated. When you change the value, you build a different version of your design. This type of design is called a parameterized design.

Parameterized designs are read into dc_shell as templates with the `read` command. These designs are stored in an intermediate format so that they can be built with different (nondefault) parameter values when they are instantiated.

If your design contains parameters, you can indicate that the design will be read in as a template, in either of two ways:

- Add the pseudocomment `// synopsys template` to your code.
- Set the dc_shell variable `hdlin_auto_save_templates = true`.

  Note:

  If you use parameters as constants that never change, do not read in your design as a template.

One way to build a template into your design is by instantiating the template in your Verilog code. Example 3-16 shows how to do this.

Example 3-16 Instantiating a Parameterized Design in Verilog Code

```
module param (a,b,c);

     input [3:0] a,b;
     output [3:0] c;

     foo #(4) U1(a,b,c); //instantiate foo

endmodule
```

Example 3-16 instantiates the parameterized design foo, which has one parameter, assigned the value 4.

Because module `foo` is defined outside the scope of module `param`, errors such as port mismatches and invalid parameter assignments are not detected until the design is linked. When Design Compiler links module `param`, it searches for template `foo` in memory. If `foo` is found, it is automatically built with the specified parameters. HDL Compiler checks that `foo` has at least one parameter and three ports and that the bit-widths of the ports in `foo` match the bit-widths of ports `a`, `b`, and `c`. If template `foo` is not found, the link fails.

Another way to build a parameterized design is with the `elaborate` command in dc_shell. The syntax of the command is

```
elaborate template_name -parameters parameterized
```

Using Templates—Naming

Templates instantiated with different parameter values are different designs and require unique names. Three variables control the naming convention for the templates:

`template_naming_style = “%s_%p”`

The `template_naming_style` variable is the master variable for renaming a template. The `%s` field is replaced by the name of the original design, and the `%p` field is replaced by the names of all the parameters.

`template_parameter_style = “%s%d”`

The `template_parameter_style` variable determines how each parameter is named. The `%s` field is replaced by the parameter name, and the `%d` field is replaced by the value of the parameter.

`template_separator_style = “_”`

The `template_separator_style` variable contains a string that separates parameter names. This variable is used only for templates that have more than one parameter.

When a template is renamed, only the parameters you select when you instantiate the parameterized design are used in the template name. For example, template `ADD` has parameters `N`, `M`, and `Z`. You can build a design where `N = 8`, `M = 6`, and `Z` is left at its default value. The name assigned to this design is `ADD_N8_M6`. If no parameters are selected, the template is built with default values and the name of the created design is the same as the name of the template.

Using Templates—list -templates Command

To see which templates are available, use the `list -templates` command. The `report_templates` command lists all templates that reside in memory and the parameters you can select for each. The `remove_template` command deletes a template from memory.

Gate-Level Modeling

Verilog provides several basic logic gates that enable modeling at the gate level. Gate-level modeling is a special case of positional notation for module instantiation that uses a set of predefined module names. HDL Compiler supports the following gate types:

- `and`
- `nand`
- `or`
- `nor`
- `xor`
- `xnor`
- `buf`
- `not`
- `tran`

Connection lists for instantiations of a gate-level model use positional notation. In the connection lists for `and`, `nand`, `or`, `nor`, `xor`, and `xnor` gates, the first terminal connects to the output of the gate and the remaining terminals connect to the inputs of the gate. You can build arbitrarily wide logic gates with as many inputs as you want.

Connection lists for `buf`, `not`, and `tran` gates also use positional notation. You can have as many outputs as you want, followed by only one input. Each terminal in a gate-level instantiation can be a 1-bit expression or signal.

In gate-level modeling, instance names are optional. Drive strengths and delays are allowed, but Design Compiler ignores them. Example 3-17 shows two gate-level instantiations.

Example 3-17 Gate-Level Instantiations

```
buf (buf_out,e);
and and4(and_out,a,b,c,d);
```

Note:

HDL Compiler parses but ignores delay options for gate primitives. Because Design Compiler ignores the delay information, it can create logic whose behavior does not agree with the simulated behavior of the circuit. See “D Flip-Flop With Asynchronous Set or Reset” on page 6-28.

Three-State Buffer Instantiation

HDL Compiler supports the following gate types for instantiation of three-state gates:

- `bufif0` (active-low enable line)
- `bufif1` (active-high enable line)
- `notif0` (active-low enable line, output inverted)
- `notif1` (active-high enable line, output inverted)

Connection lists for `bufif` and `notif` gates use positional notation. Specify the order of the terminals as follows:

- The first terminal connects to the output of the gate.
- The second terminal connects to the input of the gate.

- The third terminal connects to the control line.

Example 3-18 shows a three-state gate instantiation with an active-high enable and no inverted output.

Example 3-18 Three-State Gate Instantiation

```
module three_state (in1,out1,cntrl1);
    input in1,cntrl1;
    output out1;

    bufif1 (out1,in1,cntrl1);

endmodule
```