10

Design Compiler Interface

This chapter discusses the Design Compiler interface to Synopsys HDL Compiler for Verilog. It covers the following topics:

- Starting Design Compiler
- Reading In Verilog Source Files
- Optimizing With Design Compiler
- Busing
- Correlating HDL Source Code to Synthesized Logic
- Writing Out Verilog files
- Setting Verilog Write Variables

The Design Analyzer tool provides the graphic interface to the Synopsys synthesis tools. Design Analyzer reads in, synthesizes, and writes out Verilog source files, among others, calling Design Compiler

for these functions. When you view a synthesized schematic in Design Analyzer, you can use the RTL Analyzer tool to see how the Verilog source code corresponds to its synthesized entities and gates. For more information, see the *RTL Analyzer User Guide*.

This chapter describes the commands and variables you use to read Verilog designs. It also explains how to specify synthesis attributes and constraints for compilation and how to write out designs in Verilog format.

Note:

To understand this chapter, you must be familiar with Design Compiler concepts, especially synthesis attributes and constraints. For more information, see the Design Compiler documentation.

Starting Design Compiler

Design Compiler has two interfaces: a command-based interface (dc_shell) and a graphical user interface (Design Analyzer).

Starting the dc_shell Command Interface

Start the Design Compiler command interface by entering the invocation command `dc_shell` at your UNIX prompt.

```
% dc_shell
              Design Analyzer (TM)
            Behavioral Compiler (TM)
              DC Professional (TM)
                 DC Expert (TM)
                 DC Ultra (TM)
               FPGA Compiler (TM)
               VHDL Compiler (TM)
                HDL Compiler (TM)
              Library Compiler (TM)
              Power Compiler (TM)
               Test Compiler (TM)
              Test Compiler Plus (TM)
                  CTV-Interface
                ECO Compiler (TM)
             DesignWare Developer (TM)
                  DesignTime (TM)
                  DesignPower (TM)

          Version 2000.05 -- May 18, 2000
   Copyright (c) 1999-2000 by Synopsys, Inc.
              ALL RIGHTS RESERVED

This program is proprietary and confidential information
of Synopsys, Inc., and may be used and disclosed only as
authorized in a license agreement controlling such use and
disclosure.
Initializing...
```

When Design Compiler has finished initializing, the command-line prompt appears.

```
Initializing...
dc_shell>
```

Starting Design Analyzer

Start Design Analyzer by entering the invocation command `design_analyzer` at your UNIX prompt, in an X windows command window. As in most UNIX programs, you can use the ampersand (&) to execute Design Analyzer in the background.

```
% design_analyzer &
```

The main Design Analyzer window appears. For complete information on using Design Analyzer, see the *Design Analyzer Reference Manual*.

Design Analyzer also provides access to the dc_shell command interface, through the Setup menu's Command Window selection.

The rest of this chapter describes the commands and the menu selections you use when working with Verilog source files and designs.

Reading In Verilog Source Files

Use the Design Compiler `read` command to read in Verilog design files.

```
dc_shell> read -format verilog {file_1, file_2, file_n}
```

Use the Design Analyzer File/Read dialog box to read in Verilog design files.

All of the `read` command options are described in the Design Compiler documentation and in the `read` man page. In the next section, “Reading Structural Descriptions,” however, we include a description of the `-netlist` option for reading structural Verilog files. You might want to use this option to save time.

Reading Structural Descriptions

To read in a Verilog structural description—that is, one that contains only module instantiations and no `always` blocks or continuous assignments—use the `-netlist` option with the `read` command in dc_shell. When the `-netlist` option is present, HDL Compiler reads structural descriptions faster and uses less memory. The syntax is

```
dc_shell> read -f verilog -netlist my_file.v
```

Note:

To use the `-netlist` option with the `read` command, be sure your description is structural only. Do not use this option with any other type of description.

Use the `-netlist` option only with the `read` command. It is not an option for any other command, such as `elaborate`.

Design Compiler Flags and dc_shell Variables

Several dc_shell variables affect how Verilog source files are read. Set these variables before you read in a Verilog file with the `read -format verilog` command or the File -> Read dialog box. You can set variables interactively or in your .synopsys_dc.setup file.

To list the `hdlin_` variables that affect reading in Verilog, enter

```
dc_shell> list -variables hdl
```

The following are explanations of the Verilog reading variables:

`hdlin_auto_save_templates`

If this variable is set to true, Design Compiler saves templates (designs that use generics) in memory. If this flag is false, it saves templates only as part of the calling (instantiating) design. For more information about templates, see “Using Templates—Naming” on page 3-21 and “template Directive” on page 9-21. Design Compiler automatically generates names for templates that are based on the values of the template naming variables (described later in this chapter).

The default is false.

`hdlin_hide_resource_line_numbers`

When HDL Compiler infers a synthetic library or a DesignWare part, the line number in the HDL source is not appended to the inferred cell’s name if this variable is set to true. (The default setting of `hdlin_hide_resource_line_numbers` is false.) This value makes the results of the Design Compiler `compile` command independent of the location of the inferred synthetic library or DesignWare parts in the HDL source.

To determine the current value of `hdlin_hide_resource_line_numbers`, type

```
dc_shell> list hdlin_hide_resource_line_numbers
```

`hdlin_report_inferred_modules`

If this variable is set to true, Design Compiler generates a report about inferred latches, flip-flops, and three-state and multiplexer devices. Redirect the report file by entering

```
dc_shell> read -f verilog my_file.v > my_file.report
```

`suppress_errors`

Indicates whether to suppress warning messages when reading Verilog source files. Warnings are nonfatal error messages. If this variable is set to true, warnings are not issued; if false, warnings are issued. This variable has no effect on fatal error messages, such as syntax errors, that stop the reading process.

The default is false.

You can also use this variable to disable specific warnings: set `suppress_errors` to a space-separated string of the error ID codes you want suppressed. Error ID codes are printed immediately after warning and error messages. For example, to suppress the following warning

```
Warning: Assertion statements are not supported. They are
ignored near symbol "assert" on line 24 (HDL-193).
```

set the variable to

```
suppress_errors = "HDL-193"
```

`hlo_resource_allocation`

When set to `constraint_driven`, this variable enables automatic resource sharing (see “Resource Sharing Methods” on page 7-11). When it is set to none, each operation in Verilog is implemented with separate circuitry.

Array Naming Variable

The `bus_naming_style` variable affects the way Design Compiler names elements of Verilog arrays.

This variable determines how to name the bits in port, cell, and net arrays. When a multiple-bit array is read in, Design Compiler converts the array to a set of individual single-bit names. The value is a string containing the characters %s and %d, which are replaced by the array name and bit (element) index, respectively. If the value is

```
bus_naming_style = "%s.%d"
```

the third element of an array called X_ARRAY, indexed from 0 to 7, is represented as X_ARRAY.2.

The default is `"%s[%d]"`.

To override the default value, set this variable before you issue the `read` command.

This variable is part of the io variable group; to list its current value, enter

```
dc_shell> list -variables io
```

Template Naming Variables

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

`template_naming_style = "%s_%p"`

This is the master variable for naming a design built from 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"`

This 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 = "_"`

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

When a design is built from a template, only the parameters you indicate when you instantiate the parameterized design are used in the template name. For example, suppose the 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 listed, the template is built with default values and the name of the created design is the same as the name of the template.

Building Parameterized Designs

If your design has parameters, you can change the value of the parameters in a module each time that 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, just as other Verilog files are read. These designs are archived in a design library so they can be built with different (nondefault) values substituted for the parameters. You can also store a template in a design library with the `analyze` command.

If your design contains parameters, you can indicate that the design should be read in as a template in one of three ways:

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

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 it in your Verilog code. Example 10-1 shows how to do this.

Example 10-1 Instantiating a Parameterized Design in Verilog Code

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

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

foo #(4,5,4+6) U1(a,b,c); // instantiate foo

endmodule
```

In Example 10-1, the Verilog code instantiates the parameterized design foo, which has three parameters. The first parameter is assigned the value 4, the second parameter is assigned the value 5, and the third parameter takes the value 10.

Because module `foo` is defined outside the scope of module `param`, errors, such as port mismatches and invalid parameter assignments, are not detected until link time. When Design Compiler links module `param`, it searches for template `foo` in the design library `work`. If `foo` is found, it is automatically built with the specified parameters. Design Compiler checks that `foo` has at least three parameters 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 instantiate a parameterized design is with the `elaborate` command in dc_shell. The syntax of the command is

```
elaborate template_name -parameters parameter_list
```

You can archive parameterized designs (templates) in design libraries. To verify that a template is stored in memory, use the `report_design_lib work` command. The `report_design_lib` command lists templates that reside in the indicated design library.

Synthetic Libraries

This section gives only basic information on synthetic libraries. For a complete explanation of how to use synthetic libraries, see the *DesignWare Components Databook*.

A synthetic library contains synthetic cells called operators. Operators resemble generic logic, as they have no netlist implementation and are not linked. Operators are visible from `report_synlib standard.sldb`. Table 10-1 shows all standard operators and a description of each.

Table 10-1 Synopsys Standard Operators

Operator	Description
ADD_TC_OP	Signed adder
ADD_UNS_OP	Unsigned adder
EQ_TC_OP	Signed equality
EQ_UNS_OP	Unsigned equality
GEQ_TC_OP	Signed greater than or equal to
GEQ_UNS_OP	Unsigned greater than or equal to
GT_TC_OP	Signed greater than
GT_UNS_OP	Unsigned greater than
LEQ_TC_OP	Signed less than or equal to
LEQ_UNS_OP	Unsigned less than or equal to
LT_UNS_OP	Unsigned less than
LT_TC_OP	Signed less than
MULT_TC_OP	Signed multiplier

Table 10-1 Synopsys Standard Operators (continued)

Operator	Description
NE_TC_OP	Signed inequality
NE_UNS_OP	Unsigned inequality
SELECT_OP	Selector
SUB_TC_OP	Signed subtracter
SUB_UNS_OP	Unsigned subtracter

The selector operator, SELECT_OP, functions as a multiplexer but has a control input for each data input. When the control input for a corresponding data input is high, that input is passed to the output.

When you issue the `compile` command, Design Compiler determines an appropriate implementation for the operators in your design. Design Compiler implements an operator in three steps:

1. It chooses a module, such as `add`, and its corresponding implementation, such as `rpl_add`. The function of an implementation is determined by the operator type, such as ADD_UNS_OP, and the width of the connections to it (the bit-width).
2. It creates a netlist for the implementation and inserts the netlist in the design.
3. It optimizes the netlist.

For example, HDL Compiler generates an operator called ADD_UNS_OP_3_4_5 when you read in the following code

```
z[4:0] = a[2:0] + b[3:0];
```

One way to implement the ADD_UNS_OP operator is with a 5-bit ripple carry adder. This implementation is called `rpl_add_n5`.

To see a list of modules and their implementations, enter

```
dc_shell> report_synlib standard.sldb
```

Optimizing With Design Compiler

After HDL Compiler translates a Verilog description, it passes the description to Design Compiler for optimization and synthesis. When you read a Verilog design into Design Compiler, the design is converted to the Design Compiler internal database format. When Design Compiler performs logic optimization on a design, it can restructure all or part of the design. You have control over the degree of restructuring. You can keep your design's hierarchy intact, move modules up or down the design hierarchy, combine modules, or compress the entire design into one module.

After you read your design into Design Compiler, you can write it out in a variety of formats, including Verilog. You can convert existing gate-level netlists, sets of logic equations, or technology-specific circuits to a Verilog description. You can use the new Verilog description as documentation for the original design and as a starting point for reimplementing the design in a new technology. In addition, you can give the Verilog description to a Verilog simulator to extract circuit timing information.

This section describes some uses of the `compile` command in Design Compiler. For a complete description, refer to the `compile` man page.

Flattening and Structuring

Design Compiler uses two optimization strategies: flattening and structuring. Flattening tries to reduce a design's logical structure to a set of two-level logic equations. Structuring tries to find the common factors in the translated design's set of logic equations.

When a design is flattened, the original structure of its Verilog description is lost. Flattening is useful when a description is written at a high level without regard to the use of constructs or resource allocation. Random control logic often falls into this category. In general, flattening consolidates logic; it also often speeds up the final implementation. Not all logic can be flattened: For example, large adders, XOR networks, and comparators of two variables cannot be flattened. If you use these elements in a design, place them in separate modules that will not be flattened.

If you build structure into the Verilog description through user-defined operators (such as carry-lookahead adders) or resource sharing, do not flatten the design. You can still use structuring, which attempts to improve the design's logical structure without destroying the existing structure. The Design Compiler defaults of `-no_flatten` and `-structure` are appropriate for almost all Verilog descriptions. For more information about flattening and structuring a design, see the *Design Compiler User Guide*.

Grouping Logic

Design Compiler performs optimization on designs. All constraints and compile directives are applied at the design level. If you intend to optimize two pieces of logic differently, they must be in separate designs.

Designs in Design Compiler have a one-to-one correspondence with modules in the input Verilog description. Functions and operators in a Verilog module are grouped with that module for optimization. At times, you might regroup logic in a Verilog description to achieve the optimization you want. For example, you might want to optimize part of your design for speed and part for area. You can group the speed-critical logic and optimize it independently. You can regroup logic with the `group` command. For more information on the `group` command, see the Design Compiler documentation or the `group` man page.

Busing

Design Compiler maintains types throughout a design, including types for buses (vectors). Example 10-2 shows a Verilog design read into HDL Compiler containing a bit vector that is NOTed into another bit vector.

Example 10-2 Bit Vector in Verilog

```
module test_busing_1 ( a, b );
  input  [3:0] a;
  output [3:0] b;

  assign b = ~a;

endmodule
```

Example 10-3 shows the same description written out by HDL Compiler. The description contains the original Verilog types of ports. Internal nets do not maintain their original bus types. Also, the NOT operation is instantiated as single bits.

Example 10-3 Bit Blasting

```
module test_busing_2 ( a, b );
input  [3:0] a;
```

```
output [3:0] b;
    assign b[0] = ~a[0];
    assign b[1] = ~a[1];
    assign b[2] = ~a[2];
    assign b[3] = ~a[3];
endmodule
```

Correlating HDL Source Code to Synthesized Logic

By using RTL Analyzer, you can display the text in your source HDL code that corresponds to gates in the synthesized design. For more information, see the *RTL Analyzer User Guide*.

Writing Out Verilog Files

To write out Verilog design files, use the File/Write dialog box or the `write` command.

```
dc_shell> write -format verilog -output my_file.verilog
```

The `write -format verilog` command is valid whether or not the current design originated as a Verilog source file. Any design, regardless of initial format (equation, netlist, and so on), can be written out as a Verilog design.

For more information about the `write` command, see the Design Compiler documentation.

Setting Verilog Write Variables

Several dc_shell variables affect how designs are written out as Verilog files. To override the default settings, set these variables before you write out the design with the `write -format verilog` command or the File/Write dialog box. You can set the variables interactively or set them in your .synopsys_dc.setup file.

To list the current values of the variables that affect writing out Verilog (`verilogout_` variables), enter

```
dc_shell> list -variables hdl
```

The `verilogout_` variables are

`verilogout_equation`

When this is set to true, Verilog assign statements (Boolean equations) are written out for combinational gates, instead of for gate instantiations. Flip-flops and three-state cells are left instantiated. The default is false.

`verilogout_higher_designs_first`

When this is set to true, Verilog modules are ordered so that higher-level designs come before lower-level designs, as defined by the design hierarchy. The default is false.

`verilogout_no_tri`

When this is set to true, three-state nets are declared as Verilog `wire` instead of `tri`. This variable eliminates `assign` primitives and `tran` gates in your Verilog output, by connecting an output port directly to a component instantiation. The default is false.

`verilogout_single_bit`

When this variable is set to true, vectored ports (or ports that use record types) are bit-blasted; if a port's bit vector is `N` bits wide, it is written out to the Verilog file as `N` separate single-bit ports. When it is set to false, all ports are written out with their original data types. The default is true.

`verilogout_time_scale`

This variable determines the ratio of library time to simulator time and is used only by the `write_timing` command. The default is 1.0.