To use the Synopsys Design Compiler with VHDL Compiler, Design Compiler calls VHDL Compiler to translate a VHDL description to a netlist equivalent, then synthesizes that logic into gates in a target technology. The synthesized circuit can then be written back out as a netlist (or other technology-specific format), or as a technology-independent VHDL description.

Design Compiler can also call VHDL Compiler to write out designs in VHDL format, regardless of the design’s original format.

To explore the design compiler interface, familiarize yourself with these topics:

- Starting Design Compiler
- Processing the VHDL source files
- Using Analyze and Elaborate with VHDL source files
- Writing out VHDL files
- Optimizing with Design Compiler
Design Analyzer is the Synopsys graphic interface to its tools. Design Analyzer reads in, synthesizes, and writes out VHDL source files, among others. Design Analyzer calls Design Compiler for the functions.

When you view a synthesized schematic in Design Analyzer, you can use the text viewer to see the correspondence between VHDL source code and its synthesized entities and gates. For more information, see the Design Analyzer Reference Manual.

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

Note:
This chapter assumes that you are familiar with Design Compiler concepts, especially synthesis attributes and constraints. For more information, see the Design Compiler Family Reference Manual.
Starting Design Compiler

Design Compiler has two interfaces: a command-based interface (dc_shell), and a graphic interface (design_analyzer).

Starting the Command Interface (dc_shell)

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

% dc_shell

Design Compiler (TM)
ECL Compiler (TM)
VHDL Compiler (TM)
Library Compiler (TM)
Test Compiler (TM)

Version v3.0
Copyright (c) 1988–1992 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 is through 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 Window command window. Like most UNIX programs, you can use the ampersand (&) to run Design Analyzer in the background:

```bash
% 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.

Processing the VHDL Source Files

You can process VHDL source files in one of two ways: use Design Compiler’s `analyze` and `elaborate` commands to read in VHDL design files, or use the `read` command.

Version 3.1 and subsequent versions support common analysis for synthesis and simulation. If you analyze for synthesis, you do not need to reanalyze for the Synopsys simulator.

**WARNING**

If you use design libraries other than WORK, the `read` command is not backward compatible. To make the `read` command compatible, map all design libraries to WORK.
Using Analyze and Elaborate with VHDL Source Files

Use the `analyze` and `elaborate` commands to process VHDL source files. The advantage of using `analyze` and `elaborate` is that the out-of-date source files need to be reanalyzed when changes are made to the VHDL source. Also, the `analyze` and `elaborate` commands are the only way that design libraries other than `WORK` can be used. For more information on design libraries, see Appendix D.

Analyzing and Elaborating a Single VHDL Source File

To process a VHDL design when the entire design is in one file, in this case `alu.vhd`, use the `analyze` and `elaborate` commands as follows:

```
dc_shell> analyze -format vhdl alu.vhd

dc_shell> elaborate alu
```

Analyzing and Elaborating Multiple VHDL Source Files

To process a VHDL design that is described in more than one file, enclose the list of filenames in braces (`{}`).

```
dc_shell> analyze -format vhdl {alu.vhd, control.vhd}

dc_shell> elaborate alu
```
Analyzing HDL Packages

VHDL packages can be analyzed like any other source file. However, if the package is not being changed with every analyze, it is probably more convenient to analyze the package into a design library. After a package is analyzed into a design library, it needs to be reanalyzed only when the package becomes out-of-date.

The following example shows how to analyze packages into a library. The packages in `types.vhd` and `functions.vhd` are analyzed into the `lib1` library.

```
dc_shell> analyze -f vhdl -library lib1
               {types.vhd functions.vhd}
```

To use a package out of a library, you need to include something similar to the following code in the VHDL source:

```
library lib1;
use lib1.types.all;
use lib1.functions.all;
entity ALU is
  . . .
```

The `types` and `functions` packages are then automatically loaded out of library `lib1` if they exist. For information on how to use design libraries, see Appendix D.

**Note:**

IEEE and synopsys library packages have been analyzed for you. You need to reanalyze only if you make changes.
Reading VHDL Source Files

To read VHDL source files into Design Analyzer, use the File/Read dialog.

```
dc_shell> read -f vhdl alu.vhd
```

The `read` command does not fully support design libraries. If you use the `read` command for your packages, it deposits them in the default `WORK` directory.

**WARNING**

If you use design libraries other than WORK, note that the `read` command is not backward-compatible. To make the `read` command compatible, map all design libraries to WORK. For example:

```
dc_shell> define_design_lib WORK -path ./WORK_DIRECTORY
```

VHDL Input Variables

Several variables affect how VHDL source files are analyzed. Set these flags before you analyze in a VHDL file (`analyze -format vhdl` command, or File/Read dialog). You can set flags interactively or in your `.synopsys_dc.setup` file.

To list the flags that affect analyzing in VHDL (`hdlin_flags`), enter

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

The VHDL reading flags are

- `design_filename_length`
  - 0

- `design_library_file`
  - “.synopsys_vss.setup”
The `hdlin_advisor_directory` variable is used by (V)HDL Compiler during the generation of intermediate files for HDL Advisor. The setting of the `hdlin_advisor_directory` variable specifies the directory in which the intermediate files are placed.

The default value of `hdlin_advisor_directory` is “.”, the current directory. Either an absolute path or a path relative to the current directory are acceptable values for `hdlin_advisor_directory`. A special setting, `hdlin_advisor_directory=“*”`, can be used, which forces (V)HDL Compiler to place the intermediate files into the directory where the corresponding HDL source files reside.

To determine the current value of this variable, type `list hdlin_advisor_directory` in `dc_shell`.

The `hdlin_auto_save_templates` flag is set to true, Design Compiler saves any templates (designs using generics) in memory. If this flag is false, templates are saved only as part of the calling (instantiating) design. The default is false. Design Compiler automatically generates names for templates. The names are based on the values of the template naming variables (described in the next section, “Template Naming Variables”).

Note:
This variable is not recommended for use in Version 3.0a and later. The correct methodology for template usage is described in the section “Using VHDL Templates (Generics)” later in this chapter.
hdlin_hide_resource_line_numbers

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

To determine the current value of this variable, type list hdlin_hide_resource_line_numbers in dc_shell.

hdlin_replace_synthetic

“FALSE”

hdlin_report_inferred_modules

If this flag 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> elaborate -f vhdl > my_file.report
```

hdlin_resource_allocation

When the flag is set to constraint_driven, this flag enables automatic resource sharing (see Chapter 9). When set to none, each operation in VHDL is implemented with separate circuitry. Other VHDL statements can override this flag for specific operators.

hdlin_suppress_warnings

Indicates whether to suppress warning messages when VHDL Compiler is reading VHDL source files. Warnings are nonfatal error messages. If this variable is set to true, warnings are not issued. If it is set to 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 \texttt{true}; warning messages are issued.

You can also use this variable to disable specific warnings: set \texttt{hdlin\_suppress\_warnings} 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 (VHDL-193)

set this variable to

\texttt{hdlin\_suppress\_warnings = "VHDL-193"}

\textbf{Template Naming Variables}

The following three variables affect the way VHDL templates are named. These variables are part of the \texttt{hdl} variable group. To list their current values, enter

\texttt{dc\_shell> list -variables hdl}

The three VHDL template naming variables are

\texttt{template\_naming\_style}

Controls how templates (VHDL generics) are named. The default value is \texttt{%s\_%p}, where \texttt{%s} is the name of the source design and \texttt{%p} is the parameter list. By default, the design name and the parameter list are separated by an underscore (\_). The naming style for the parameter list is the value of the \texttt{template\_parameter\_style} variable.
template_parameter_style
Controls how template parameters are named. The default value is %s%d, where %s is the name of a parameter and %d is the value of that parameter. By default, the parameter name and value are not separated. If there are two or more parameters, each parameter name or value pair is separated by the value of the template_separator_style variable.

template_separator_style
Provides a separator character for multiple parameters in a template name. The default value is _ (underscore).

Array Naming Variables
The following three I/O variables affect how elements of VHDL arrays are named. These variables are part of the io variable group; to list their current values, enter

dc_shell> list -variables io

The three VHDL read_array_ naming variables are

read_array_minus_style
Describes how to represent negative indexes in port, cell, and net names. The value is a string containing the characters %d (replaced by the absolute value of a negative index).

If the value is

read_array_minus_style = “M%d”

a negative index value -3 is represented as “M3”.

The default is “-%d”.

See also the read_array_naming_style and read_array_separator_style variables.
read_array_naming_style

Describes how to name the bits in port, cell, and net arrays. When a multiple-bit array is read in, it is converted 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

read_array_naming_style = "%s.%d"

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

The default is "%s[%d]". See also the read_array_minus_style and read_array_separator_style variables.

read_array_separator_style

Describes how to separate index values in a multidimensional port, cell, or net array. The value is a character or string that is inserted between consecutive indexes. A multidimensional array is one whose elements are represented by more than one bit, such as array of enumerated values, or an array of arrays.

If the value is

read_array_separator_style = "::"

in an array of four-bit values indexed from 0 to 7, the first element of that array is represented as X_ARRAY[0:0] through X_ARRAY[0:3] (using the default read_array_naming_style).

The default is "[]". See also the read_array_minus_style and read_array_naming_style variables.
Using VHDL Templates (Generics)

VHDL Compiler supports the use of generic entity declarations, as described in “Entity Generic Specifications,” under “Defining Designs” in Chapter 3.

To analyze a template, use the `analyze` command. For example,

```
dc_shell> analyze -format vhdl template.vhd
```

Templates are stored in design libraries.

For more information about design libraries, see Appendix D.

A template is not a design until it is instantiated, either by a later VHDL design or by using the `elaborate` command. For example, if `TEMPLATE` has one parameter `N` with default 8, the following two `elaborate` commands create two designs from that template.

```
dc_shell> elaborate TEMPLATE
1
dc_shell> elaborate TEMPLATE -parameter "N=3"
1
dc_shell> list -designs
```

<table>
<thead>
<tr>
<th>Design</th>
<th>File</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPLATE_N3</td>
<td>TEMPLATE.db</td>
<td>/usr/designs/</td>
</tr>
<tr>
<td>TEMPLATE_N8</td>
<td>TEMPLATE.db</td>
<td>/usr/designs/</td>
</tr>
</tbody>
</table>

Note that the first `elaborate` command builds the design with default settings (N=8), and the second `elaborate` command builds the design with `N` set to 3.
Design Compiler generates names for the templates; the names are based on the value of the `template_naming_style` variable and the values of the two related variables `template_parameter_style` and `template_separator_style`. These variables are described under “Template Naming Variables” in this chapter.

If `read` is used to read in VHDL files and the VHDL files have generics, the following method enables designs to be analyzed into the work directory.

Set the variable `hdlin_auto_save_templates` to true. The VHDL Compiler directive `--pragma template` also is used to produce the same effect. Entities with generic specifications are read in and analyzed into the `WORK` library. Note that the `WORK` library restriction is true only for the `read` command. The `analyze` command allows you to specify a particular library.

For example, to read in a file `template.vhd` containing one or more templates:

```
dc_shell> hdlin_auto_save_templates = true
    "true"
dc_shell> read -format vhdl template.vhd
```

```
/usr/designs/template.vhd:
No designs were read
{}
Grouping VHDL Designs

When a VHDL design is read into memory, it is represented as one top-level design (the VHDL entity) and a set of subdesigns (the VHDL subprograms and processes). When Design Compiler synthesizes the design, it may optimize the subdesigns into the top-level design.

To create a level of hierarchy for a subdesign during compilation, use the `group -hdl_block` command with the label of a subprogram or process.

```vhdl
entity DESIGN
  ...
architecture EX of DESIGN is
begin
  MY_PROC: process (SIGNALS)
  ...
  end process MY_PROC;
end EX;
```

dc_shell> analyze -format vhdl DESIGN.vhdl
dc_shell> elaborate design
dc_shell> group -hdl_block MY_PROC
dc_shell> compile

The name of the new design is its label (MY_PROC above) plus one or more digits to make the name unique: MY_PROC1.

Use the following commands for grouping:

- To group each subdesign in a VHDL design into a separate block, use the `group -hdl_all_blocks` command.
- To group only the subdesigns of one labeled subprogram or process, use the `group -hdl_block label` command.
- To group all bused gates (gates connected to a multibit signal), use the `group -hdl_bussed` command.
- To group only bused gates in one labeled subprogram or process, use the `group -hdl_block label -hdl_bussed` command.
Writing Out VHDL Files

To write out VHDL design files, use the write command, or the File/Write dialog.

dc_shell> write -format vhdl -output my_file.vhdl

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

For more information about the write command, see the Design Compiler Family Reference Manual.

VHDL Write Variables

Several dc_shell variables affect how designs are written out as VHDL files. These variables must be set before you write out the design (write -format vhdl command or File/Write dialog). They can be set interactively or in your .synopsys_dc.setup file.

To list the variables that affect writing out VHDL (vhdlout_variables), enter

dc_shell> list -variables vhdlio

vhdlout_equations

When this variable is set to true, combinational logic is written with technology-independent BOOLEAN equations, and sequential logic is written with technology-independent wait and if statements. Three-state drivers are also written with technology-independent code. When this variable is set to false, all mapped logic is written with technology-specific netlists.

The default is false.
vhdlout_local_attributes

This variable is obsolete. Use the dc_shell command write_script instead. Refer to the write_script man page.

vhdlout_upcase

When this variable is set to true, identifiers are written out in upper case to the VHDL file. When this variable is set to false, identifiers are written out with their Design Compiler names. The default is false.

vhdlout_use_packages

A list of package names. A use clause is written into the VHDL file for each of these packages for all entities. library clauses are also written out as needed.

If this variable is not set, or is set to an empty list ({}), it has no effect on the write command.

To use packages from specific libraries, you can prefix the library to the package name. For example:

vhdlout_use_packages = {IEEE.std_logic_1164, 
IEEE.std_logic_arith, 
VENDOR.PARTS.FFD}

becomes

use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use VENDOR.PARTS.FFD;

vhdlout_write_architecture

When this variable is set to true (the default), an architecture definition is written out to the VHDL file.

vhdlout_write_attributes

This variable is obsolete. Use the dc_shell command write_script instead. Refer to the write_script man page.
vhdlout_write_components
Controls whether component declarations for cells mapped to a technology library are written out (if set to true) or not (false).

Component declarations are required by VHDL. If you set this variable to false, make sure a package containing the necessary component declarations is listed in vhdlout_use_packages.

The default is true.

See also the vhdlout_use_packages variable.

vhdlout_write_constraints
This variable is obsolete. Use the dc_shell command write_script instead. Refer to the write_script man page.

vhdlout_write_entity
When this variable is set to true (the default), an entity definition is written out to the VHDL file, and any conversion packages necessary.

See also the vhdlout_write_top_configuration and vhdlout_top_configuration_entity_name variables.

vhdlout_write_top_configuration
When this variable is set to true, a top-level configuration definition is written out to the VHDL file.

The default is false.

See also the vhdlout_top_configuration_name, vhdlout_top_configuration_arch_name, and vhdlout_top_configuration_entity_name variables.
Bit and Bit Vector Variables

These variables define the names of bits, bit vectors, and the associated types.

vhdlout_bit_type
The name of the bit type used for writing out single-bit values, used with the following variables:

vhdlout_one_name
vhdlout_three_state_name
vhdlout_zero_name variables
hdlout_bit_vector_type

The default is std_logic. The previous default was bit. For example, if a simulator prefers a bit type of t_logic, defined as

type t_logic is (U, D, Z, ..., F0, F1, ...);

a bit vector type of t_logic_vector, defined as

type t_logic_vector is array (integer range <>)
of t_logic;

The following dc_shell commands define the appropriate bit and bit vector types and values to write.

vhdlout_bit_type = t_logic
vhdlout_bit_vector_type = t_logic_vector
vhdlout_one_name = F1
vhdlout_zero_name = F0 vhdlout_three_state_name = Z

When writing a generic three-state model, Design Compiler displays an error if vhdlout_bit_type is set to its default value of bit. Set vhdlout_bit_type to a bit type that includes a high-impedance value (‘Z’). For more information, see “Three-State Inference” in Chapter 8.
vhdlout_bit_vector_type
The name of the bit vector type used for writing multiple-bit values, used with the vhdlout_bit_type, vhdlout_one_name and vhdlout_zero_name variables.

The default is std_logic_vector. The previous default was bit_vector.

For an example, see the description of vhdlout_bit_type.

vhdlout_one_name
The name of the enumeration literal that represents a logic 1.

The default is ‘1’.

For an example see the description of vhdlout_bit_type.

vhdlout_three_state_name
The name of the high impedance bit value used for three-state device values.

The default is ‘Z’.

vhdlout_unknown_name
The value used to drive a signal to the unknown state, usually a character literal or an enumeration name.

The default is ‘’X’’.

See also the vhdlout_one_name, vhdlout_zero_name, vhdlout_three_state_name variables.

vhdlout_zero_name
The name of the enumeration literal that represents a logic 0.

The default is ‘0’.

For an example, see the description of vhdlout_bit_type.
Resolution Function Variables

These variables name resolution functions that are written out.

vhdlout_three_state_res_func
Names a three-state resolution function to use instead of the default function. You must supply this function in a package listed in vhdlout_use_packages.

If the variable is set to "" (the default), a resolution function is written out if needed.

See also the vhdlout_wired_and_res_func,
vhdlout_wired_or_res_func, and vhdlout_use_packages variables.

vhdlout_wired_and_res_func
Names a wired-and resolution function to use instead of the default function. You must supply this function in a package listed in vhdlout_use_packages.

If the variable is set to "" (the default), a resolution function is written out if needed.

See also the vhdlout_three_state_res_func,
vhdlout_wired_or_res_func, and vhdlout_use_packages variables.

vhdlout_wired_or_res_func
Names a wired-or resolution function to use instead of the default one. You must supply this function in a package listed in vhdlout_use_packages.

If the variable is set to "" (the default), a resolution function is written out if needed.

See also the vhdlout_wired_and_res_func,
vhdlout_three_state_res_func, and vhdlout_use_packages variables.
Types and Type Conversion Variables

Types and type conversion variables define type conversion functions and how types are written out.

vhdlout_conversion_functions

Overrides the default conversion functions. The value is a list of lists. Each lower-level list is a list of three strings: “from_type”, “to_type”, and “conversion_function”. “from_type” and “to_type” are the base types to be converted, and “conversion_function” is the name of your conversion function from “from_type” to “to_type”.

For example:

vhdlout_conversion_functions = {
    { "INTEGER", "BIT_VECTOR", "MY_INT_TO_BV" },
    { "BIT_VECTOR", "STRING", "MY_BV_TO_STRING" } };

In this example, the function MY_INT_TO_BV is used for conversions from INTEGER to BIT_VECTOR, and the function MY_BV_TO_STRING is used for conversions from BIT_VECTOR to STRING.

You must provide the specified conversion functions in a package listed in vhdlout_use_packages. The default is {}.

See also the vhdlout_single_bit and vhdlout_use_packages variables.

vhdlout_dont_write_types

When this variable is set to true, user-defined types (types not predefined by IEEE VHDL) are not written to the VHDL file. Declare your types elsewhere, probably in your own TYPES package.

When this variable is set to false, user-defined types are written. The default is false.
vhdlout_package_naming_style
Controls how packages of conversion functions are named. The default value is "CONV_PACK_%d", where %d is a number that is incremented as necessary to produce a unique name. By default, the package name and the number are separated by underscores (_).

See also the vhdlout_conversion_functions variable.

vhdlout_preserve_hierarchical_types
This variable affects how ports on lower-level designs are written out. Top-level design ports are controlled by vhdlout_single_bit. (A design is considered lower level if it is instantiated by any of the designs being written out.)

When this variable is set to USER, all ports on lower-level designs are written with their original data types. This option affects only designs that are read in VHDL format.

When set to VECTOR, all ports on lower-level designs are written with their ports bused; ports keep their names (in contrast to ports that are bit-blasted. Bit-blasting is the term for breaking down a bus to its individual bus members). The port types are defined by vhdlout_bit_vector_type, or by vhdlout_bit_type in the case of single-bit ports. This setting is likely to give the most efficient description for simulation. Default is VECTOR. You must ensure that vhdlout_bit_vector_type is an array-type whose elements are vhdlout_bit_type.

When this variable is set to BIT, typed ports are bit-blasted. If the type of a port is n-bits wide, it is written to the VHDL file as n separate ports. Each port is given type as defined by vhdlout_bit_type. This variable has no effect if vhdlout_single_bit = BIT. vhdlout_preserve hierarchical_types is then ignored and the whole design hierarchy is written out bit-blasted.
Also note that this variable cannot take on a higher value than the current setting of \texttt{vhdlout\_single\_bit}. The descending order is \{\texttt{USER}, \texttt{VECTOR}, \texttt{BIT}\}. Thus the combination \texttt{vhdlout\_single\_bit = VECTOR} and \texttt{vhdlout\_preserve\_hierarchical\_types = USER} is not possible.

See also the \texttt{vhdlout\_single\_bit} variables.

\texttt{vhdlout\_single\_bit}

This variable affects how ports on the top-level design are written out. Lower level design ports are controlled by \texttt{vhdlout\_preserve\_hierarchical\_types}. A design is considered lower level if it is instantiated by any of the designs being written out.

When this variable is set to \texttt{USER}, all ports on the top level design are written with their original data types. This option affects only designs that are read in VHDL format. Default is \texttt{USER}.

When this variable is set to \texttt{VECTOR}, all ports on the top level design are written with their ports bused. Ports keep their names (in contrast to bit-blasted ports). Port types are defined by \texttt{vhdlout\_bit\_vector\_type}, or by \texttt{vhdlout\_bit\_type} in the case of single-bit ports. For busses, the range always starts with zero and goes in ascending order, regardless of what the range definition was in the HDL source. It is your responsibility to ensure that \texttt{vhdlout\_bit\_vector\_type} is an array-type whose elements are of \texttt{vhdlout\_bit\_type}.

When this variable is set to \texttt{BIT}, typed ports are bit-blasted. If the type of a port is \text{n}-bits wide, it is written to the VHDL file as \text{n} separate ports. Each port is given the type defined by \texttt{vhdlout\_bit\_type}.
To determine the current value of this variable, use the `vhdlout_single_bit` list.

**VHDL Simulator Variables**

These variables provide the name and time scale for your VHDL simulator.

**vhdlout_target_simulator**

Set to the name of your target simulator. Currently, only one simulator is supported: the Zycad XP hardware accelerator. If this variable is set to "xp", the VHDL output contains a special attribute in architectures, telling the Synopsys simulator to simulate the architecture on the Zycad XP hardware accelerator.

The default is "".

**vhdlout_time_scale**

Determines the ratio of library time to simulator time. Is used only by the `write_timing` command.

The default is 1.0.
Architecture and Configuration Variables

These variables control the names of the architectures, configuration, and entities written to the VHDL file.

vhdlout_architecture_name
Controls architecture naming. The default value is “SYN_%a_%u”, where %a is the name of the architecture and %u is a number that is incremented as necessary to produce a unique name. By default, the architecture name and the number are separated by underscores (_).

See also the vhdlout_write_architecture and vhdlout_top_configuration_arch_name variables.

vhdlout_top_configuration_arch_name
Determines the architecture name that is written out in a configuration definition. The default value is “A”.

See also the vhdlout_write_top_configuration, vhdlout_top_configuration_entity_name, and vhdlout_top_configuration_name variables.

vhdlout_top_configuration_entity_name
Determines the entity name that is written out in a configuration definition. The default value is “E”.

See also the vhdlout_write_top_configuration, vhdlout_top_configuration_arch_name, and vhdlout_top_configuration_name variables.

vhdlout_top_configuration_name
Determines the entity name that is written out in a configuration definition. The default value is “CFG_TB_E”.

See also the vhdlout_write_top_configuration, vhdlout_top_configuration_arch_name, and vhdlout_top_configuration_entity_name variables.
Writing out VHDL

Examples 12–1 through 12–7 are explicit examples of how to write out a VHDL source description by using Design Compiler. Text surrounded by /* (slash-asterisk) and */ (asterisk-slash) is ignored by `dc_shell`.

Converting an Existing Design to Technology-Independent VHDL

Example 12–1 shows how to translate a design read in LSI netlist format to a technology-independent VHDL format.

**Example 12–1  Writing Technology-Independent VHDL**

/* Read in the LSI netlist */
`dc_shell>` `read -format lsi ADDER.NET`

/* Set the variable that causes */
/* technology-independent designs to be written */
`dc_shell>` `vhdlout_equations = "true" ;`

/* Now write the design as VHDL source */
`dc_shell>` `write -format vhdl -output ADDER.vhd`

Writing out Finite-State Machines

Example 12–2 shows how to write out the current design in VHDL format as a finite-state machine.

**Example 12–2  Writing out Finite-State Machines in VHDL Format**

/* First, extract the finite state machine */
/* from the current design */
`dc_shell>` `extract`

/* Now write the current design in VHDL format */
`dc_shell>` `write -format vhdl -output outfile.vhdl`

**Note:**

Preserving Port Types

Example 12-3 shows how to write out the current design in VHDL format with port types (vector or record types) preserved.

Example 12-3  Preserving Port Types When Writing VHDL

/* The design must originate in VHDL format */
dc_shell> read -format vhdl my_design.vhdl

/* Set the variable that causes the */
/* port types to be preserved */
dc_shell> vhdlout_single_bit = "user";

/* Now write the current design in VHDL format */
dc_shell> write -format vhdl -output design_out.vhdl

Example 12-4 shows a VHDL input file. Examples 12-5 and 12-6 show the corresponding output files.

Example 12-4  Original VHDL Input File

library IEEE;
use IEEE.std_logic_1164.all;

dc_shell> read -format vhdl my_design.vhdl

entity test_vhdl is
port ( a: in std_logic_vector (3 downto 0);
       b: out std_logic_vector ( 3 downto 0));
end test_vhdl;

architecture structural of test_vhdl is
begin
  b <= not a;
end structural;

Example 12-5 is produced by the following commands, using the default values of the vhdlout_variables (described under “VHDL Write Variables”, earlier in this chapter).
Example 12–5  TEST_VHDL Written Out in Default VHDL Format

dc_shell> analyze –format vhdl TEST_VHDL.vhdl

dc_shell> elaborate TEST_VHDL

dc_shell> write –format vhdl –out OUT.vhdl TEST_VHDL

library IEEE;
use IEEE.std_logic_1164.all;
entity test_vhdl is

    port( a : in std_logic_vector (3 downto 0);
          b : out std_logic_vector (3 downto 0));

end test_vhdl;

architecture SYN_structural of test_vhdl is

begin

    b(3) <= not(a(3));
    b(2) <= not(a(2));
    b(1) <= not(a(1));
    b(0) <= not(a(0));

end SYN_structural;

If you set vhdlout_single_bit to bit, the output file generated is shown in Example 12–6.

Example 12–6  TEST_VHDL Written Out with Port Types in VHDL Format

dc_shell> analyze –format vhdl TEST_VHDL.vhdl

dc_shell> elaborate TEST_VHDL

dc_shell> vhdlout_single_bit = bit

dc_shell> write –format vhdl –output OUT.vhdl TEST_VHDL

...
library IEEE;
use IEEE.std_logic_1164.all;

entity test_vhdl is
  port( a_3_port, a_2_port, a_1_port, a_0_port : in std_logic; b_3_port, b_2_port, b_1_port, b_0_port : out std_logic);
end test_vhdl;

architecture SYN_structural of test_vhdl is
begin
  b_3_port <= not(a_3_port);
  b_2_port <= not(a_2_port);
  b_1_port <= not(a_1_port);
  b_0_port <= not(a_0_port);
end SYN_structural;

VHDL Netlister
This section describes known problems and limitations in the VHDL Netlister.

Limited Conversion of Logic Values
The VHDL Netlister provides limited support for user-defined data types. The following sections describe this support and the requirements for declaring custom data types.
**Built-In Type Conversion Function**

The VHDL Netlister does not use packages and does not check for type equivalence. If you do not provide your own type conversion functions, the VHDL Netlister translates only the logic values 0 and 1. Example 12–7 shows the VHDL Netlister’s built-in type conversion function that converts from type `my_bit` to type `std_logic_vector`.

**Example 12–7  Type Conversion Function**

```vhdl
-- User-defined type declaration
attribute ENUM_ENCODING : STRING;
type my_bit is (A, B, C);
attribute ENUM_ENCODING of my_bit : type is “00 01 11”;

-- std_logic_vector to enum type function
function std_logic_vector_to_my_bit(arg : in std_logic_vector ( 1 to 2 ))
return my_bit is
-- synopsys built_in SYN_FEED_THRU;
begin
  case arg is
    when “00” => return A;
    when “01” => return B;
    when “11” => return C;
    when others => assert FALSE -- this should not happen.
        report “un-convertible value”
        severity warning;
      return A;
  end case;
end;
```

**Workaround:**

Use the `vhdlout_conversion_functions` variable to define the functions to use for type conversions. Use the `vhdlout_use_packages` variable to indicate the packages that contain the conversion functions. If type declarations are contained in packages, set the variable `vhdlout_dont_write_types` to true.
How the Netlister Handles Custom Types

All types you use should be resolved. If types are not resolved, the VHDL Netlister uses built-in resolution functions to resolve conflicts between multiple drivers on the same signal. Use the following functions to specify your own resolution function to the VHDL Netlister:

vhdlout_three_state_res_func
vhdlout_wired_and_res_func
vhdlout_wired_or_res_func

Example 12–8 shows the resolution function that the VHDL netlister writes out. This resolution function is used to resolve the value for multiple sources driving a signal, port, or pin.

Example 12–8  VHDL Resolution Function

function X( inputs : in vhdlout_bit_vector_type )
    return vhdlout_bit_type is
    -- synopsys resolution_method three_state
    variable retval: vhdlout_bit_type;
    begin
        retval := vhdlout_three_state_name;
        for i in inputs’range loop
            if inputs(i) /= vhdlout_three_state_name then
                if ( retval = vhdlout_three_state_name ) then
                    retval := inputs(i);
                else
                    retval := vhdlout_unknown_name
                    exit;
                end if;
            end if;
        end loop;
        return retval;
    end X;
Example 12-9 shows a simplified description of the process flow for the resolution function in Example 12-8.

**Note:**

The `vhdlout_three_state_name` and `vhdlout_unknown_name` variables use the default values `z` and `x` (respectively) for brevity in this example. You can set the values for both of these variables.

**Example 12-9  Pseudocode of VHDL Resolution Function:**

```vhdl
if the only logic values are 'z'
    return 'z'
if there are 'z's and another logic value
    return the other logic value
if there are non-'z' logic values that are different
    return 'x'
else
    return the common logic value
```

**Workaround:**

Use resolved types to prevent VHDL Netlister from using the pessimistic resolution function described in these examples. Set the `vhdlout_bit_type_resolved` to `true` if all of your types are resolved. The VHDL Netlister will not write out the resolution function.

### Optimizing with Design Compiler

Design Compiler optimizes (compiles) a design into a technology-specific circuit that reflects the attributes and constraints you require in your design. You use the `compile` command to produce an optimized circuit and a status report. For more information about optimizing with Design Compiler, see the *Design Compiler Family Reference Manual*. For information about the `compile` command, see the `compile` man page.