cadence®

Virtuoso VHDL Toolbox User Guide

Product Version IC23.1
June 2023

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

1

About the VHDL Integration Environment

VHDL Toolbox is an integrated netlisting and simulation environment that you can use to generate structural VHDL (IEEE93/87) text netlists from hierarchical schematics and run simulations in the Cadence NC environment.

This user guide is aimed at developers and designers of integrated circuits and assumes that you are familiar with:

- The Virtuoso Studio design environment and application infrastructure mechanisms designed to support consistent operations between all Cadence® tools.
- The applications used to design and develop integrated circuits in the Virtuoso Studio design environment, notably, the Virtuoso Layout Suite, and Virtuoso Schematic Editor.

This topic explains VHDL-related concepts and procedures and the conditions that you need to know when netlisting and simulating your VHDL design. It also describes the VHDL Toolbox and explains how to run VHDL netlister in standalone mode using Cadence Simulation Environment (SI) of Open Simulation System (OSS).

In this chapter, you will learn about the following topics:

Licensing Requirements

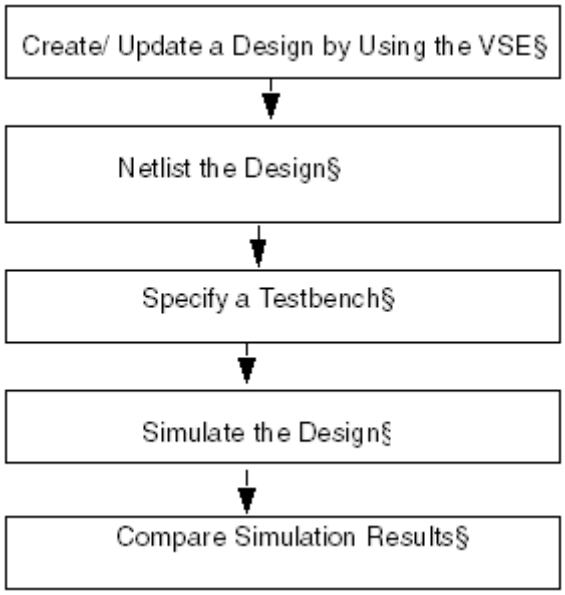
For information about licensing in the Virtuoso Studio design environment, see *Virtuoso Software Licensing and Configuration Guide*.

Schematic Design Process Flow

In a schematic design, you work at the gate level or the Register Transfer Level (RTL). You build the design hierarchy with schematics and VHDL models at the leaf level. The design is simulated and debugged using the VHDL simulator to make it ready for use by a physical layout tool.

As a designer, you need to ensure that the schematic design is up to date and the connectivity is extracted by using the Virtuoso Schematic Editor window. In addition, you need to ensure that the VHDL text files are already in the DFII5.x structure with a valid database reference. For information about organization of data, refer to How Data is Organized.

The flowchart given below provides an overview of the schematic design process flow.



Updating a Design

You use the Virtuoso Schematic Editor to edit your data. The VHDL netlister supports automatic data type propagation from the leaf level VHDL to top level schematic ports, so you do not need to add any VHDL-specific properties to the schematic in the VHDL netlister. However, you can specify VHDL data types for pins and nets, VHDL generics, and netlist options for blackbox devices, which are devices that have no VHDL textviews in DFII.

If you want to be prompted for a pin data type during the creation and editing of a pin, change the pin master map to point to the pins in the `VHDL_PINS` category in the `basic` library and set the *Options – Editor* command option *Add Symbol Pins As Instances* to `yes`. In

addition, you can run the source rule checker using *Check – Current Cellview* to verify that the design is legal VHDL. The checks are grouped together under the VHDL check on the *Check – Rules Setup* command form. You can browse any errors or warnings that generate markers in the schematic using the *Check – Markers* commands.

If you have an existing schematic design, you can add data types to the schematic using the *Edit – Properties – Objects* command. The *Edit – Properties – VHDL* command defines the set of properties that are included in the VHDL design unit. Specify which properties of the cellview or the cellview's CDF should be included. You can reference any existing properties without entering a duplicate set of properties for VHDL netlisting and simulation. You can set the netlister options using the *Edit – Properties – VHDL* command. Use the VHDL package check to ensure that legal VHDL can be generated from the schematic. Fix any errors in naming before proceeding further.

For more information on VHDL netlister properties, refer to Appendix 8, "VHDL Netlister Properties."

Note: You can import text cellviews from Verilog, SystemVerilog, Verilog-AMS, VHDL, and VHDL-AMS text files into the DFII environment using the `cdsTextTo5x` command. This command also lets you generate the symbol views of the imported cellviews. For details, see *Importing Design Data by Using cdsTextTo5x*. You can also use this command to create text cellviews (5x structure), symbol views, and shadow database for SPICE, Spectre, DSPF, and PSpice.

Netlisting the Design

To netlist a design, you need to specify the design as library name, cell name, and view name. The netlister traverses the design hierarchy and converts all the schematics into VHDL design units. The traversal is defined by a switch list, a stop view list, and a stop library list as you specified in the *Setup – Netlist* command in VHDL Toolbox. After netlisting, you have the option to convert the hierarchy specification into a VHDL configuration by using the VHDL netlister.

For each schematic in the hierarchy, the netlister generates a VHDL architecture. The netlister also creates a VHDL entity, once per cell. You can customize the code generation by setting default values in the *Setup – Netlist* command in the VHDL Toolbox. You can override these values on a per-cellview basis or a per-object basis in Virtuoso Schematic Editor L.

Specifying a Testbench

You can specify an existing testbench or create a new using the VHDL Toolbox. To create a testbench for the top-level schematic by using the VHDL Toolbox window, choose *Command*

– *Edit Test Bench*. This command creates an entity and an architecture. The entity has no ports and the architecture instantiates the top-level schematic. The VHDL Toolbox automatically displays the testbench architecture in a text editor window, which is used to enter the stimulus data.

Simulating the Design

Now you are ready to simulate the design using the *Simulate* command in the VHDL Toolbox. The *Simulate* command invokes the VHDL elaborator to create a simulation model. When elaboration completes successfully, the VHDL simulator is run in batch or interactive mode based on options you specify in the *Setup Simulate* command.

In the simulation and debug loop, edit the source and simulate until you obtain correct results. The steps to complete the loop are

1. Access source data using the Library Manager or Browser.
2. Edit the schematics using Virtuoso Schematic Editor L.
3. Use the *Simulate* command to update any modified cellviews by netlisting and analysis. If successful, it also elaborates the design and invokes the simulator.

You use the SimCompare window to compare two simulation results.

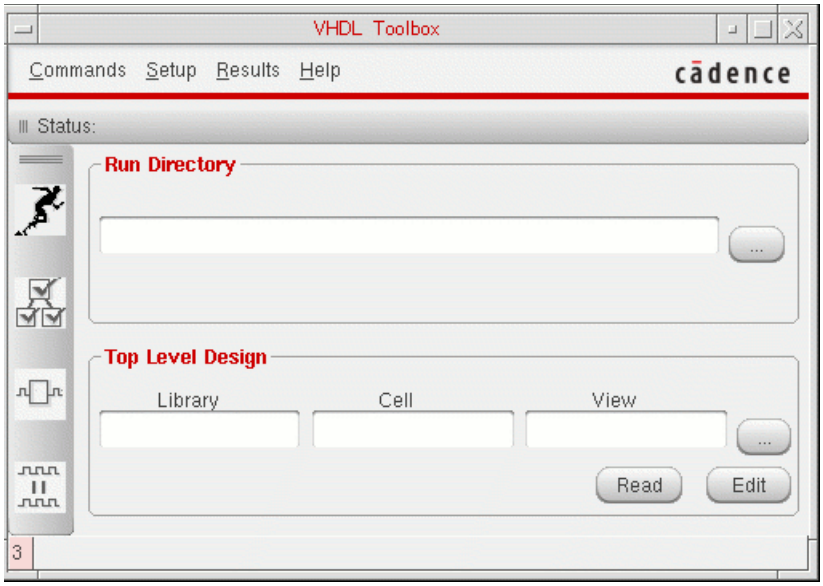
VHDL Integration Environment Tools

The VHDL integration environment consists of the following:

- VHDL Toolbox Window
- Simulation Window
- SimCompare Tool

VHDL Toolbox Window

The VHDL Toolbox window, shown below, appears when you access the VHDL integration environment. This window provides access to simulation commands, command forms, and support tools.



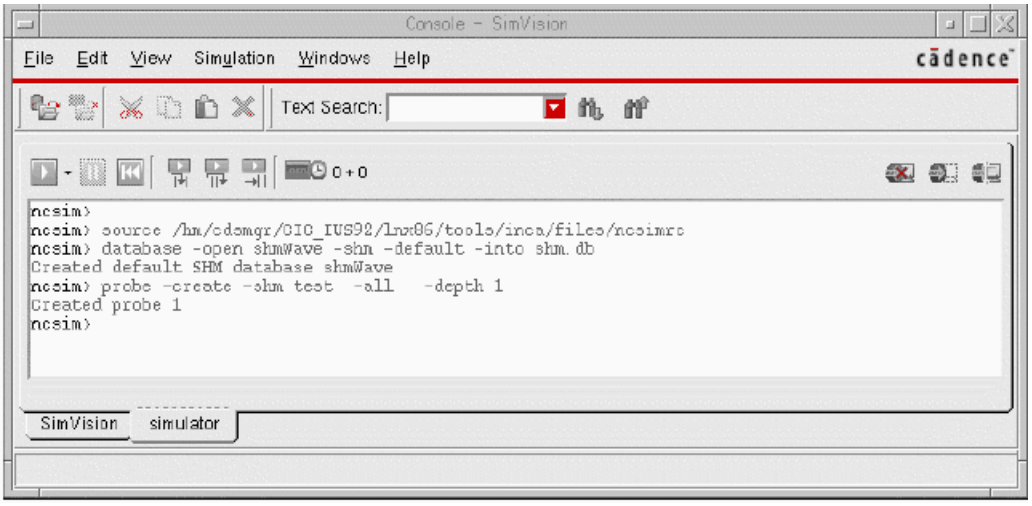
The various parts of the VHDL Toolbox window are described in detail in Chapter 2, "Introducing the VHDL Toolbox."

Simulation Window

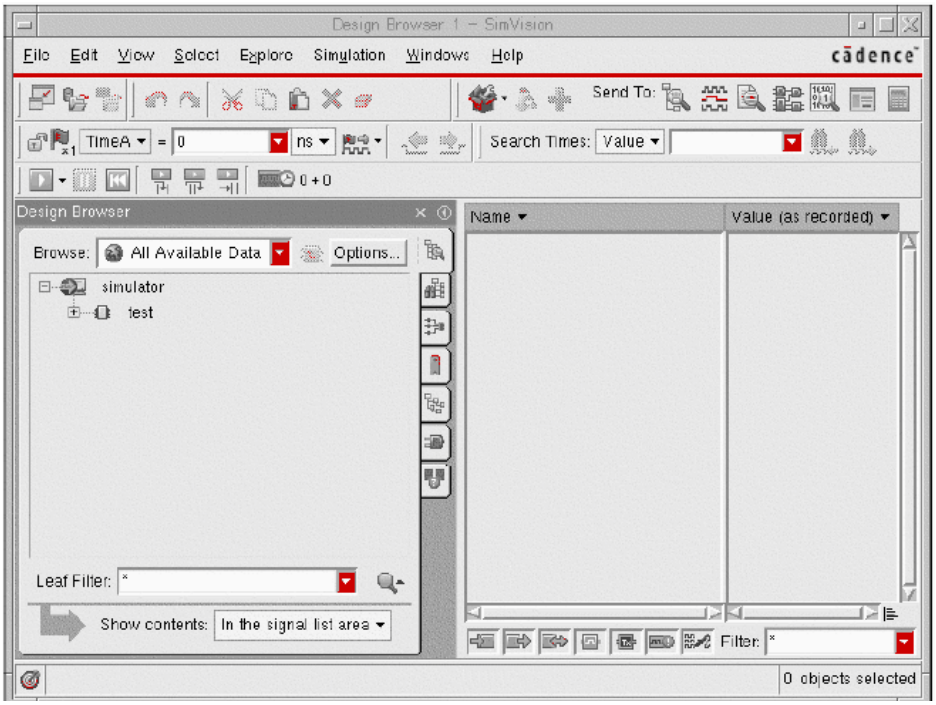
The Simulation window, which you launch from the main VHDL Toolbox window, is used to interactively simulate and debug the design. You use the Simulation window to directly interact with the simulator. You can open a database, trace signals, set breakpoints, observe signals, and perform many other functions to verify your design.

The Simulation window opens when you perform an interactive simulation. In an interactive simulation, you can select in the VHDL Toolbox window the options to compile, elaborate, and simulate the design. The Console - SimVision and Design Browser SimVision windows are displayed after the design has been successfully compiled and elaborated.

Console - SimVision



Design Browser - SimVision



The Simulation window is part of the Cadence® SimVision Analysis environment, which is a unified graphical debugging environment for Cadence simulators.

SimCompare Tool

The SimCompare tool is used by VHDL Toolbox in the background to compare the results obtained from different simulations. It is run when you select the Simulation Compare option from the fixed menu. SimCompare provides a description of any differences that are found.

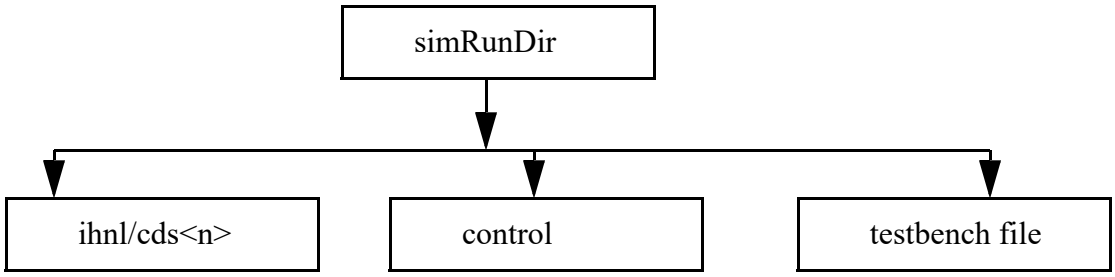
The VHDL Toolbox uses this tool to compare simulation results. The tool runs in the background. To run the SimCompare tool, select the *Compare Simulation* option from the fixed menu. You can also run the SimCompare tool by choosing *Commands – Compare Simulation*.

SimCompare can compare simulation results stored in SST2 or VCD format. For more information about using the SimCompare tool, refer to SimCompare User Guide.

If SimCompare is not available, VHDL uses the Comparescan tool for comparing simulation results. For more information about using the Comparescan tool, refer to Comparescan User Guide.

Managing the Run Directory

You store in the run directory the files that you require to run a simulation, such as the testbench files. Files created during a simulation run, such as the simulation result files, are also stored in the run directory. The following figure shows the structure of a run directory.



For detailed information about testbench files, refer to Chapter 4, “Creating a Testbench.”

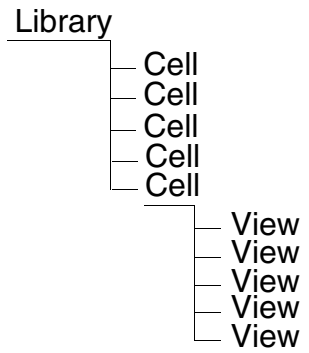
How Data is Organized

You use the Library Manager and Team Design Manager (TDM) to manage your designs. The Library Manager and TDM support browsing, file-locking, and archiving the schematics. You specify the paths to your libraries in the `cds.lib` file.

VHDL design units are stored in Cadence design libraries as views of a cell. Refer to the New Cadence Library Structure section in Chapter 1, Compatibility Guide for details.

Design Library Storage

Cadence design libraries are organized in three levels shown below.



The library is identified by location and name.

A library contains a set of cells – for example – `NAND2` and `NOR`.

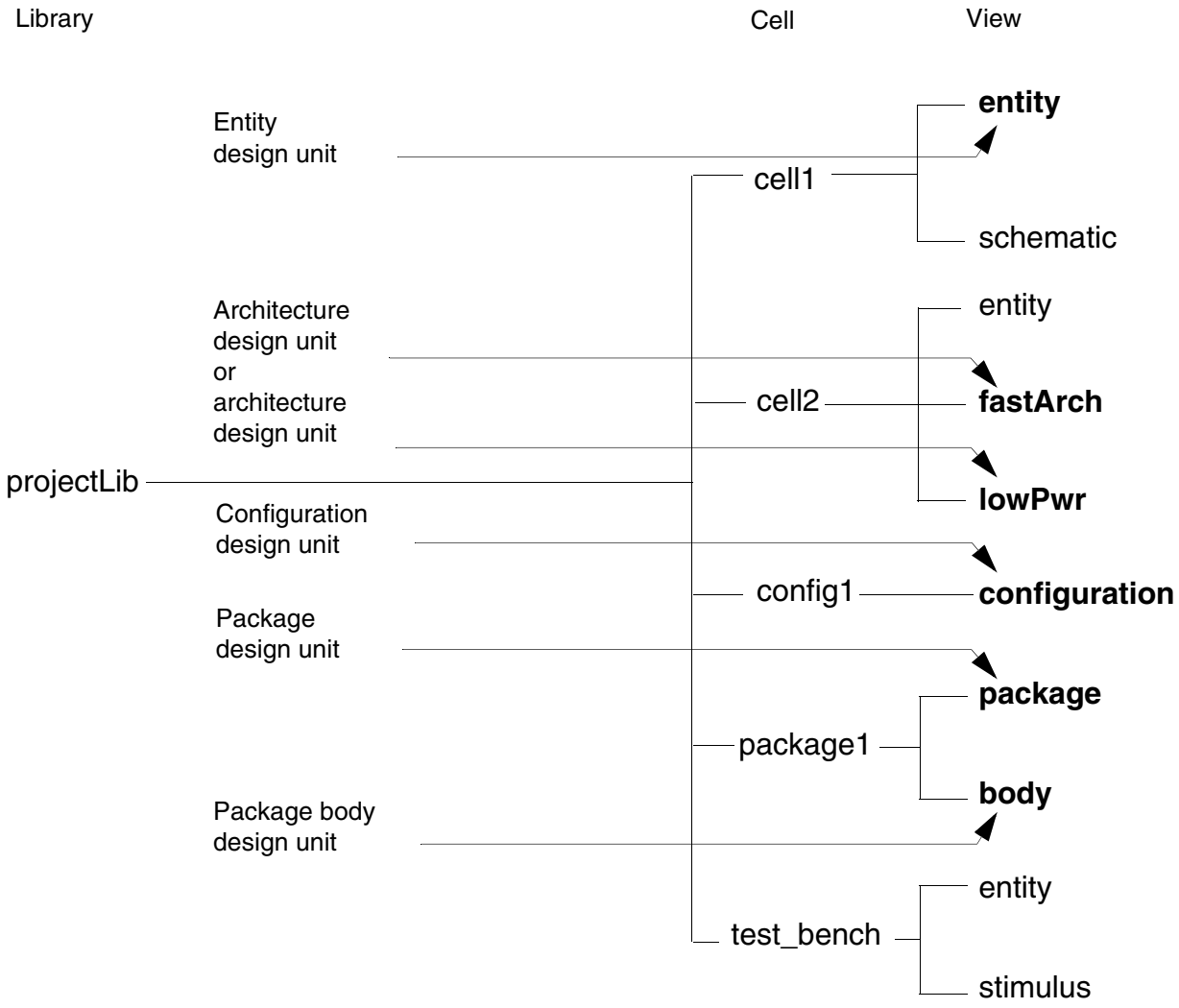
Each cell has a set of views – for example, `schematic`, `entity`, and `behavior` – representing different implementations of the cell.

VHDL Design Units

VHDL defines three primary and two secondary design unit classes.

- Entity declaration (primary unit)
- Architecture body (secondary unit, related to an entity)
- Configuration declaration (primary unit)
- Package declaration (primary unit)
- Package body (secondary unit, related to a package)

The following diagram shows a typical organization for a design library containing a mixture of schematic and VHDL source data. These five sample design units are described in this section.



The five VHDL design units are maintained in a design library, in this case, `projectLib`.

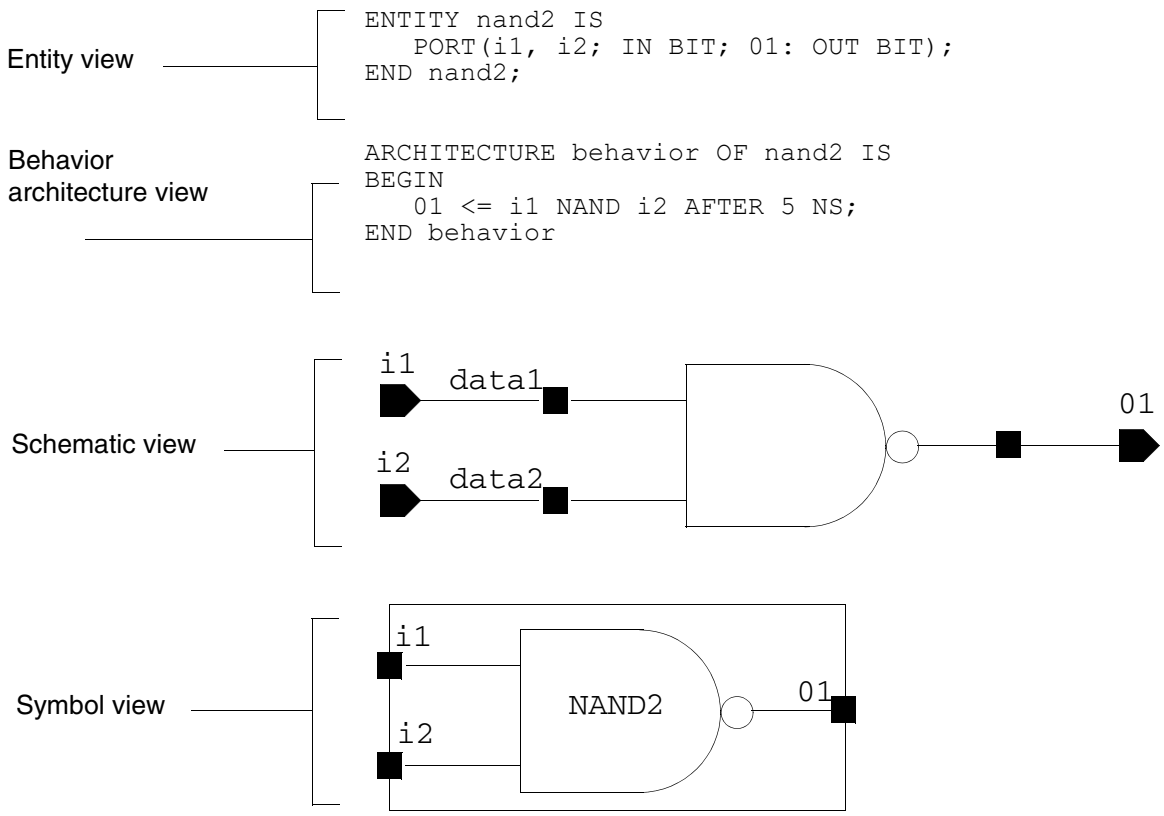
- **Entity** design units are stored as the `entity` view name under the cell name that matches the entity name. In the previous example, the entity for `cell1` is maintained as `projectLib cell1 entity`.
- **Architecture** design units are stored with the view name matching the architecture name and the cell name matching the entity name. In the example, `fastArch` of `cell2` is shown.

- **Configuration** design units are maintained as the configuration view name under the cell matching the name of the configuration. In the example, configuration `config1` is stored as `projectLib config1 configuration.`
- **Package** design units are maintained as `package` view name for the declaration. They are stored under the cell name, which matches the package name. For example, `package1` has a declaration and a body. The body is optional.
- **Package body** design units are maintained as `body` view name for the body. They are stored under the cell name, which matches the package name. For example, `package1` has a declaration and a body. The body is optional.

Note: Testbench cellviews are modeled as another entity/architecture pair. By default, `stimulus` is the name of the view for testbenches.

Design Views

The following example shows textual and symbolic views of a `nand2` gate.



VHDL Name Mapping

Cadence developed a consistent strategy for dealing with naming problems between tools. The strategy is called name mapping. Each Cadence tool interprets names according to a consistent set of rules. Data is interoperable across many tools and data formats. Also, data is interoperable between UNIX and NT operating systems.

You can avoid name mapping issues by following some rules, such as always choosing names that use only lowercase letters and digits.

For details on VHDL name mapping, see "Mapping Library, Cell, and Cellview Names to VHDL" on page 84.

For details on name mapping in general, see *Cadence Application Infrastructure User Guide*.

VHDL Text View Database

You can create the database for the VHDL text views in the following ways:

- By calling the `vmsUpdateCellViews` function. For more details of this function, see Appendix D, SKILL Functions and Customization Variables in *Spectre AMS Designer Environment User Guide*.
- Open the VHDL entity in a text editor, save and close it. An `.oa` database will be automatically created.

When the database is created, by default, the names of identifiers are converted to lowercase. To preserve the case of identifiers in VHDL text views:

1. Set the `vhdlKeepCaseAsNC` as `t` in the *.cdsinit* file. For more details, see Appendix D, SKILL Functions and Customization Variables in *Spectre AMS Designer Environment User Guide*.
2. Add the following declaration in the *hdl.var* file:

```
'define xmvhdlopts -keepcase4use5x'
```

If you have set the `vhdlKeepCaseAsNC` as `t`, it is also recommended to set the `hnlVHDLDonotUseCdsNmp` variable as `t` in the *.simrc* file before netlisting.

Netlist and Simulate a VHDL Design

In the Virtuoso Studio design environment, you can netlist and simulate a VHDL design using:

- The command-line interface
- The graphical user interface

Using the Command-Line Interface

From the command-line interface, you can generate a netlist and simulate the VHDL design by using the *xrun utility* in the batch or interactive mode.

Note: The executable and log file names will depend on the simulator being used. For the changes in the executable and log file name when using the Xcelium simulator, see Running Simulations with Xcelium on page 119.

Generating a Netlist

To generate a netlist and to set up your VHDL design for simulation as a batch:

1. Create the `si.env` file in the run directory and add VHDL hierarchical netlister (HNL) variables to the file. For information about VHDL HNL variables that the VHDL netlister uses while netlisting a design, refer to the Setting VHDL HNL Variables section in Appendix B. You can also set these variables in the `.simrc` file.

   Note: For more information about these variables, see "SE Variables" section in Chapter 3, Customizing the Simulation Environment (SW) of Open Simulation System Reference.

2. Run the following `si` command, which represents the OSS binary file, to generate a netlist:

   ```
   si -batch -command netlist -cdslib cds.lib
   ```

   The command-line options used with the `si` command are:

   - `-batch`: Refers to the batch simulation mode
   - `-command`: Specifies the command to generate a netlist
   - `-loadLocal`: Loads the `si.local` file from the simulation run directory
   - `-cdslib`: Specifies the `cds.lib` library, which defines the design libraries and the path to these libraries. You are required to specify the complete path of the `cds.lib` library while running this command.

Note: To maintain the log of the netlisting process, use the `tee` command. For example, the following command stores log entries in `si.log`.

```
si -batch -command netlist -cdslib cds.lib | tee si.log
```

Simulating a Netlisted VHDL Design

To simulate a netlisted VHDL design as a batch process:

1. Add the VHDL *xrun* variables in the `si.env` or `.simrc` file. For information about the SKILL variables that are used for the *xrun utility*, refer to the Setting xrun Variables section in Appendix B.

   If you already have these variables set in the `.vhdlrc` file in the run directory, you do not need to add these variables to the `si.env` or `.simrc` files. The settings from `.vhdlrc` are used.

Note: If you are simulating the netlisted design in the interactive mode, set these variables with the values specified in the *VHDL Setup - Simulation* form. For information about the *VHDL Setup - Simulation* form, refer to Chapter 5, “Simulating a Netlisted VHDL Design.”

Important

Whenever the run directory is re-initialized, the values for variables set in the `.simrc` file are set in the VHDL Toolbox as default values. After you save the values, only the saved settings are used.

2. In the `.simrc` file, add the following information:

   ```
   vhdlSimSimulator="xrun"
   ```

3. Run the following `si` command to generate a netlist:

   ```
   si -batch -command netlist -cdslib cds.lib
   ```

4. After the design is successfully netlisted, run the following `si` command to simulate the design by using the *xrun utility*:

   ```
   si <simRunDirName> -batch -command vosLaunchIrunSimulation
   ```

   The command-line options used with this command are:

   - `<simRunDirName>`: Name of the run directory.
   - `-batch`: Simulation mode
   - `-command`: Command to run the simulation by using the *xrun utility*.

Alternatively, to netlist and simulate the design together, set the `vhdlSimNetlistandSimulate` variable. If this variable is set, you do not need to run step 4, given above.

Important

In the NC mode, you can use the traditional xmsim utility to simulate a netlisted design.

Using the Graphical User Interface

From the graphical user interface (GUI), you can generate a netlist and simulate the VHDL design by using the *VHDL Toolbox,* which is described in detail in the following chapters.

2

Introducing the VHDL Toolbox

VHDL Toolbox is an integrated environment that you can use to generate netlists and run simulations. The toolbox, through a netlister, can generate default VHDL testbenches or you can include external testbenches before running a simulation. The generated netlist can then be simulated and debugged using the toolbox, or in the interactive or batch mode by using the SimVision tool (*xrun utility*).

Note: The SimVision tool is a part of the Cadence SimVision Analysis Environment, which is a unified graphical debug environment for Cadence simulators. For more information on SimVision user interface, see *SimVision User Guide*.

In this chapter, you will learn about the following topics:

VHDL Toolbox Features

VHDL Toolbox uses OSS-based traversal techniques that provide high level of customization. The following are the features the toolbox:

- Ability to set various options from the `.simrc` file.
- Built-in support for pre- and post-netlist functions for user customization. For more information, refer to Customizing Pre- and Post-Processing Functions.
- Automatic data type propagation from the leaf-level VHDL to the top-level schematic ports.
- Improved VHDL netlist generation that provides:
  - Netlisting of HED configuration designs.

- Support for inherited connections.
- Support for proxy symbol views that can be used to resolve inherited connections.
- Support for resolving data type conflicts in hierarchical netlists. For more details, refer to Support for Type Conflict Resolution.
- Support for generating netlists by using Cadence name mapping or by preserving case.
- Ability to generate single or multiple netlist files.
- Support for two types of library bindings–binding the design units to the associated DFII library or to a single logical VHDL library.

For more details, refer to Chapter 3, "Netlisting a VHDL Design."

- Support for simulating Verilog modules at the leaf level. For more details, refer to VHDL Setup - Simulation Form.

Important

The OSS-based netlister requires a *.cdb* or *.oa* database file in the *lib/cell/view* directory. Therefore, before netlisting, ensure that a database is created for the text views used in the design. For more details, refer to the VHDL Text View Database section.

Opening the VHDL Toolbox GUI

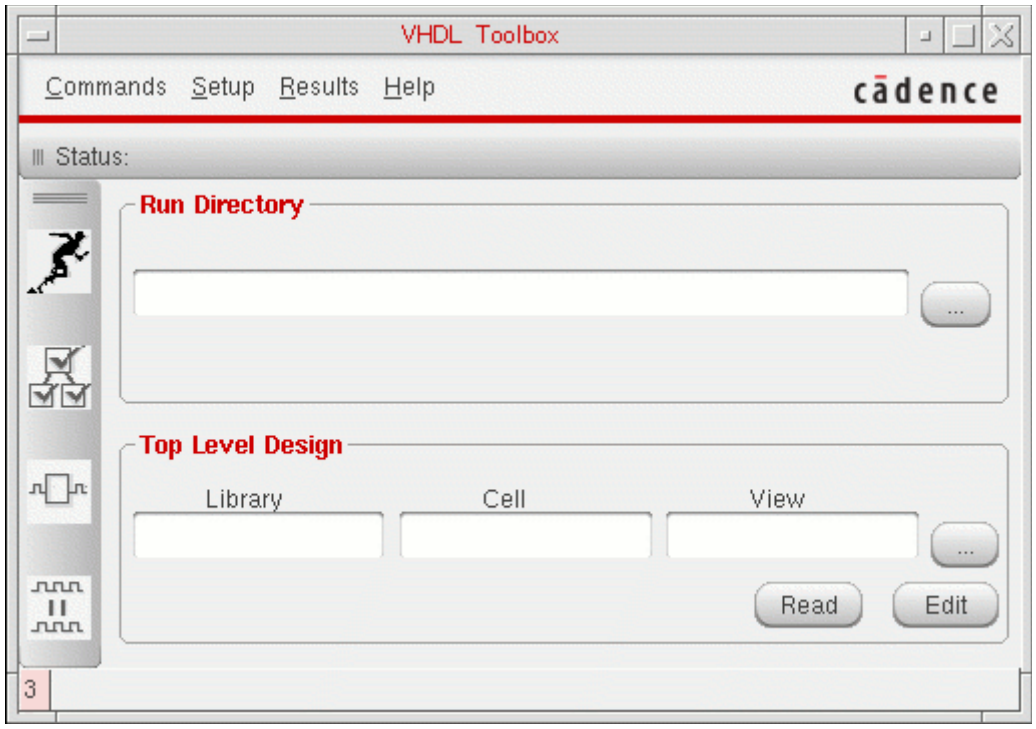
There are two ways to open the VHDL Toolbox:

- From the CIW
- From the Virtuoso Schematic Editor window

To open the VHDL Toolbox from the CIW:

➥ In the CIW, choose *Tools – VHDL Toolbox*.

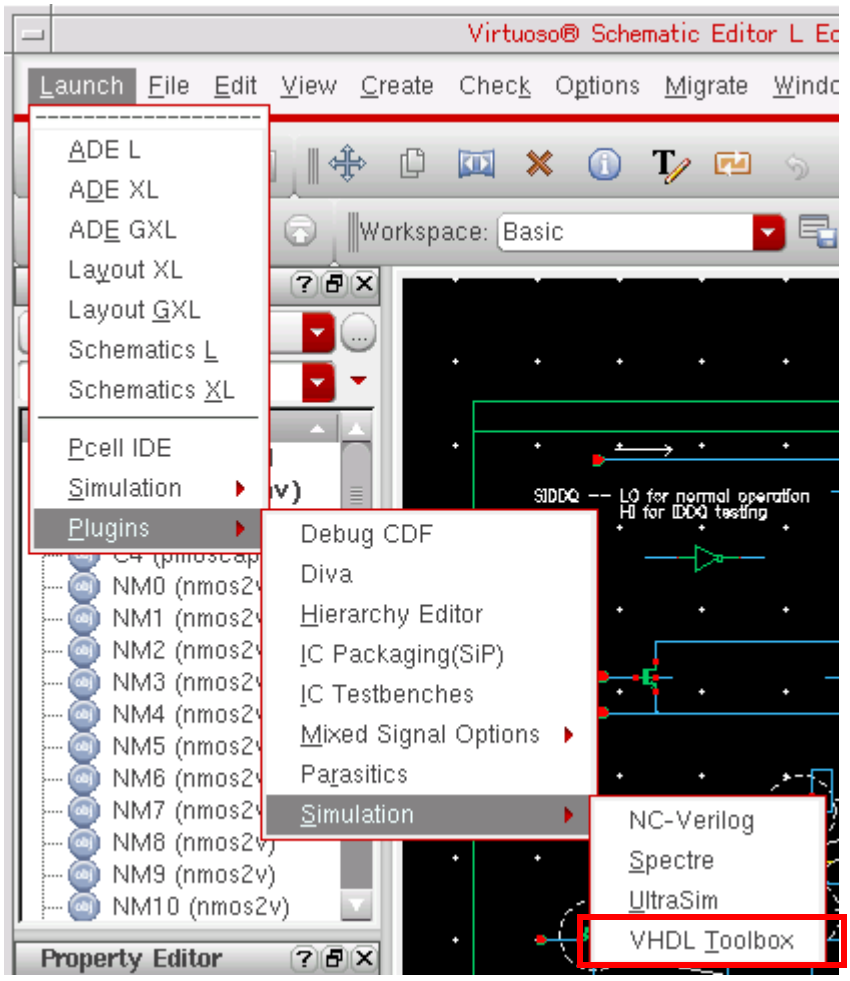
The VHDL Toolbox window is displayed as shown in the figure below.



To open the VHDL Toolbox from the Virtuoso Schematic Editor window:

➡ In the Virtuoso Schematic Editor window, choose *Launch – Plugins – Simulation – VHDL Toolbox*, as displayed in the figure below.

.



The VHDL Toolbox window is displayed.

Alternatively, you can press the `Alt+ l, p, s` and `t` keys to open the VHDL Toolbox.

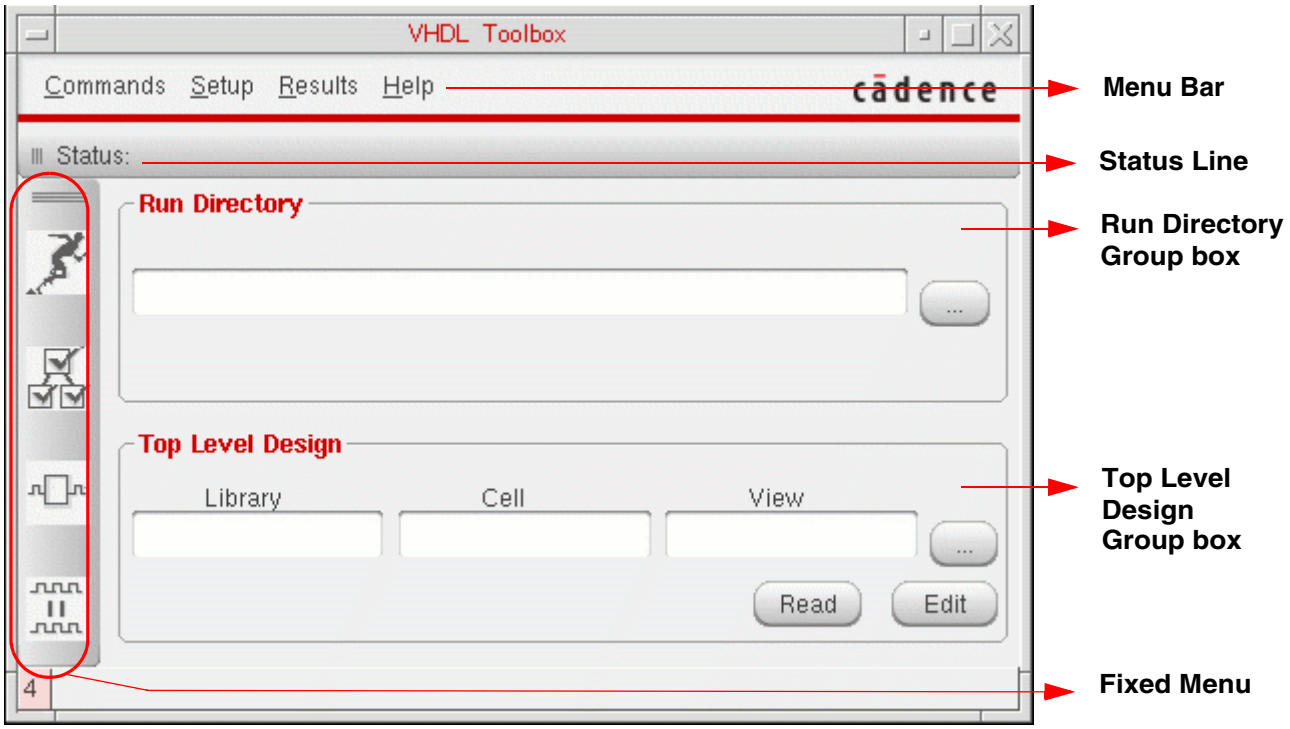
Note: Only a single instance of VHDL Toolbox can run at a time.

VHDL Toolbox GUI

The VHDL Toolbox GUI consists of the following main areas:

- Run Directory Group Box
- Top Level Design Group Box

- Menu Bar
- Status Line
- Fixed Menu
- Command Buttons



Run Directory Group Box

You use the *Run Directory* group box to specify the name of the directory that you want to use for netlisting and simulating the VHDL design and for creating a waveform database. You can create a new directory or specify the name of an existing directory. You can also specify a relative directory name in the *Run Directory* group box instead of specifying the complete path. Relative names are automatically expanded when you move to the next field.

Note: The commands on the *Commands* and *Setup* menus become available after you run the *Initialize Run Dir* command.

Top Level Design Group Box

The *Top Level Design* group box contains fields for specifying the library, cell, and view names of the top-level design. You can either type the library, cell, and view names in the

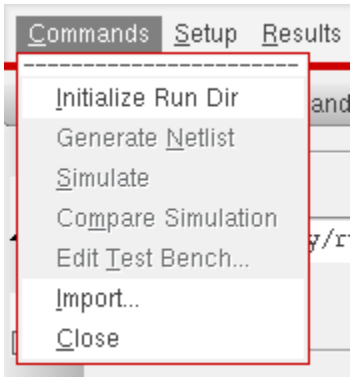
Library, *Cell*, and *View* fields, respectively, or select the required values by using *Library Browse*r. To open *Library Browse*r, click the ellipsis button next to the *View* field. You can specify the HED configuration in a similar way.

Menu Bar

The menu bar provides access to the commands and forms required to generate a netlist and to run the simulation for your VHDL design. The menu bar contains four menus: *Commands, Setup, Results,* and *Help*.

Commands Menu

The *Commands* menu contains commands as shown in the figure below.



- Initialize Run Dir: Initializes the specified run directory for netlisting and simulation and validates the top design. For more information, refer to Chapter 3, “Netlisting a VHDL Design.”
- Generate Netlist: Triggers the netlister to generate the VHDL hierarchical netlist for the design specified in the *Top Level Design* group box. The commands on the *Results* menu become available after a netlist is generated. For more information, refer to Chapter 3, “Netlisting a VHDL Design.”
- Simulate: Triggers the process for simulating a design. For more information, refer to Chapter 5, “Simulating a Netlisted VHDL Design.”.
- Compare Simulation: Compares two waveform databases. For more information, refer to Chapter 5, “Simulating a Netlisted VHDL Design.”.
- Edit Test Bench: Opens the VHDL Create Test Bench form that you use to configure the properties of a testbench. For more information, refer to Chapter 4, “Creating a Testbench.”

- Import: Runs the VHDL import tool that imports existing VHDL text files to a design library. For more information, refer to the *VHDL In for Virtuoso Studio Design Environment User Guide and Reference*.
- Close: Closes the VHDL Toolbox window. For more information, refer to Exiting the VHDL Toolbox on page 31.

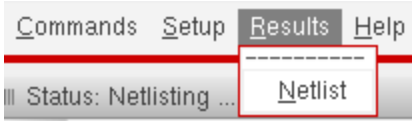
Setup Menu

The *Setup* menu contains the *Netlist*, *Simulation*, and Sim *Comparison* commands that you can use to customize the netlisting and simulation flows. The following figure shows the commands available on the *Setup* menu:

- Netlist: Opens the VHDL Setup - Netlist form. For more information, refer to Chapter 3, "Netlisting a VHDL Design."
- Simulation: Opens the *VHDL Setup - Simulation* form. For more information, refer to Chapter 5, "Simulating a Netlisted VHDL Design."
- Sim Comparison: Opens the VHDL Setup - Sim Comparison form in which you can specify the names of the Simulation History Manager (SHM) databases to compare. In this form, you also specify the name of the output log that contains the results of comparison. For more information, refer to Chapter 5, "Simulating a Netlisted VHDL Design."

Results Menu

The *Results* menu contains the *Netlist* command that you can use to view the netlisting results in a new window. The *Netlist* command becomes available only after a netlist is generated in a session. The following figure shows the *Results* menu:



Note: Netlist results are displayed only if in the *VHDL Setup - Netlist* form, the *Single Netlist File* check box is selected.

Help Menu

The *Help* menu contains options to access help on using VHDL Toolbox and get online support.

Status Line

The status line displays the status of the command that was last run.

Fixed Menu

The fixed menu contains the following frequently used commands:

Initialize Run Dir

Netlist Design

Simulate Design

Compare Simulation

Command Buttons

The two command buttons at the bottom of the VHDL Toolbox window, *Read* and *Edit*, open in Virtuoso Schematic Editor the design that you specified in the *Top Level Design* group box.

Opens in Virtuoso Schematic Editor the design that you specified in the *Top Level Design* group box for viewing.

Opens in Virtuoso Schematic Editor the design that you specified in the *Top Level Design* group box for editing.

Exiting the VHDL Toolbox

To exit the VHDL Toolbox from the CIW:

➡ Choose *Commands – Close*.

The VHDL Toolbox window closes.

To exit the VHDL Toolbox from the Virtuoso Schematic Editor window:

➡ Click the *Close* button in the upper-right corner of the VHDL Toolbox window.

The VHDL Toolbox window closes.

3

Netlisting a VHDL Design

This chapter describes the steps to netlist your VHDL design.

In this chapter, you will learn about the following topics:

Initializing the Run Directory

In the VHDL Toolbox, choose *Commands - Initialize Run Dir* option to initialize the run directory that you specified in the *Run Directory* field and to validate the top design. The run directory is initialized with the default directory structure required to create hierarchical netlists.

The fixed menu commands become available after you initialize the run directory.

Configuring the VHDL Netlister

You can configure the VHDL netlister to customize the netlisting environment. Netlister configuration includes the following tasks:

- Customizing Pre- and Post-Processing Functions
- Setting up Hierarchical Specifications
  - Netlist Options Tab
    - Specifying the Traversal Settings‘
    - Setting up Netlisting Controls

- VHDL Options Tab
- Other Options Tab

- Setting up the VHDL Netlister for Inherited Connections

In addition, following are some additional features of the netlister:

- Support for Generating the Configurations of All the Cellviews in a Design
- Support for Instance Bindings
- Support for External VHDL Source Files
- Support for Type Conflict Resolution
- Support for Shorting Devices
- Support for Shorting Terminals

For information on the SKILL functions associated with VHDL Toolbox, see *Digital Design Netlisting and Simulation SKILL Reference*.

Customizing Pre- and Post-Processing Functions

The VHDL Toolbox allows you to customize and run the pre- and post-processing SKILL functions before you start the VHDL Toolbox. A pre-processing function executes before the netlisting starts while a post-processing function executes after the netlisting completes. VHDL Toolbox supports the `hnlVHDLPreNetlistFunc()` and `hnlVHDLPostNetlistFunc()` pre- and post-processing functions, which you can define in the `.cdsinit` or `.simrc` file. The following example shows how to define the `hnlVHDLPreNetlistFunc()` function:

```
procedure(hnlVHDLPreNetlistFunc()
    printf("Welcome to VHDL Netlister !!\n")
)
```

The following example shows how to define the `hnlVHDLPostNetlistFunc()` function:

```
procedure( hnlVHDLPostNetlistFunc()
    printf("Done with VHDL Netlisting !!\n")
)
```

Setting up Hierarchical Specifications

You need to set up hierarchical specifications before you can netlist your design. The hierarchical specifications specify how the cellviews combine to form a hierarchy. When you

netlist your design, each top-level block is bound to an architecture cellview or schematic cellview. For example, you might bind the memory block to a behavior architecture cellview and a cntl_1 block to a schematic cellview.

The VHDL Setup - Netlist form enables you to specify different component configurations, such as Cadence HED Configuration, or Switch View and Stop View List, for your design at any given time.

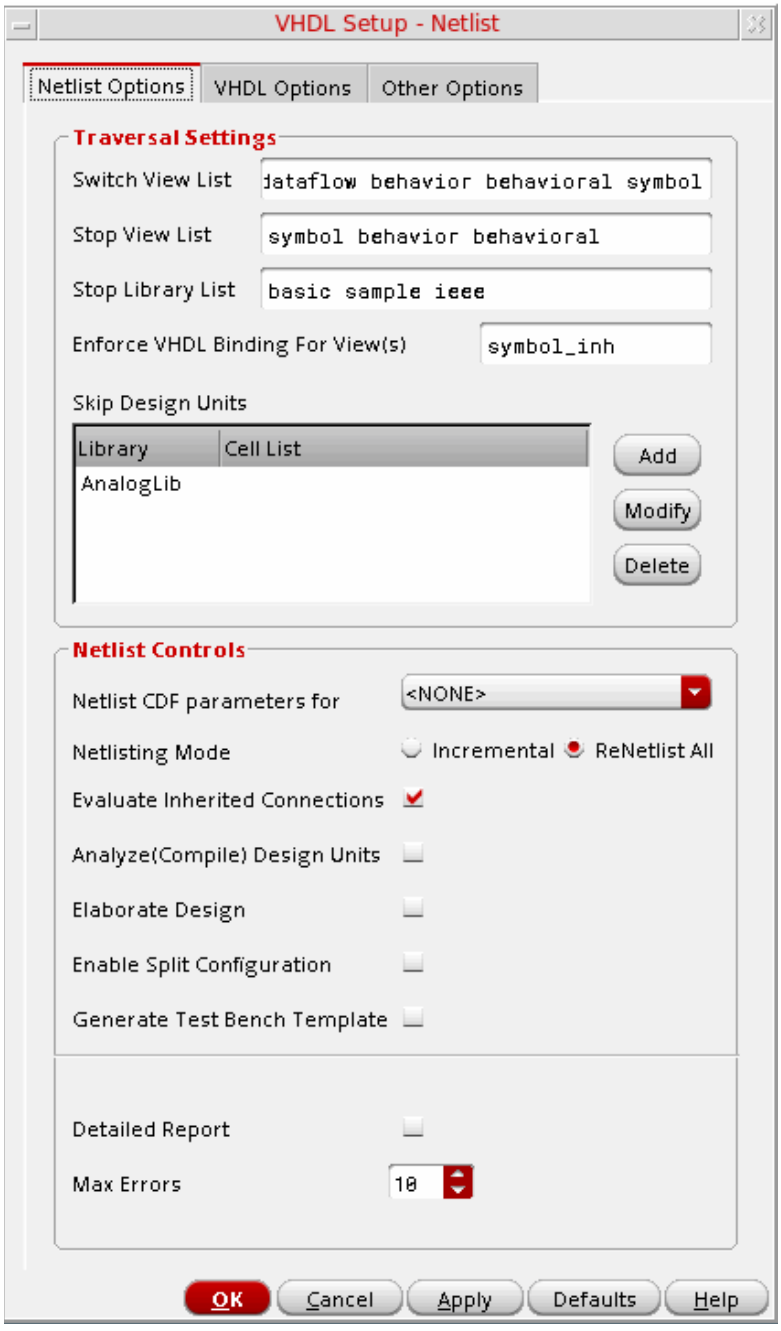
To open the VHDL Setup - Netlist form:

➡ In the *VHDL Toolbox* window, choose *Setup - Netlist*.

The VHDL Setup - Netlist form is displayed, as shown in the figure below. This form consists of three tabs:

- Netlist Options Tab
- VHDL Options Tab
- Other Options Tab

Netlist Options Tab



The following tables describe the various options on the Netlist Options tab.

Specifying the Traversal Settings

Option	Description
Switch View List	Specifies a list of views that define how the top cell/design is traversed. The top/cell design is finally netlisted as per the stop view list. Default: `stimulus schematic structural dataflow behavioral symbol` To generate the entity based on the symbol views, set the following flags in the `.simrc` file: ■ `hnlVHDLGenerateEntityFromSymbol = t` to generate entity and component with view name `symbol` for all cells except stop cells. ■ `hnlVHDLEnableDataTypePropagation = nil` to stop data propagation while netlisting. ■ `hnlVHDLGenerateEntityFromSymbolForStopCell = t` to generate entity and component with view name `symbol` for stop cells.
Stop View List	Specifies a list of views, which indicates the last levels of hierarchy needed for the netlist. If the switch view selected by a netlister is available on the stop view list, the hierarchy stops expanding. This indicates that the entity/architecture for the cellview is not required to be generated. The netlister prints only the instance line in the final netlist. If the current switch view is not available in the list, netlisting continues with the next level of hierarchy. Default: `symbol behavioral`
Stop Library List	Specifies a list of libraries, which contain the cells, which you do not need to traverse while netlisting a design. These cells are treated as stop cell views. If the library of the current instance appears in this list, the instance is not expanded further and only the instance line is printed in the final netlist. Default: `sample basic ieee`

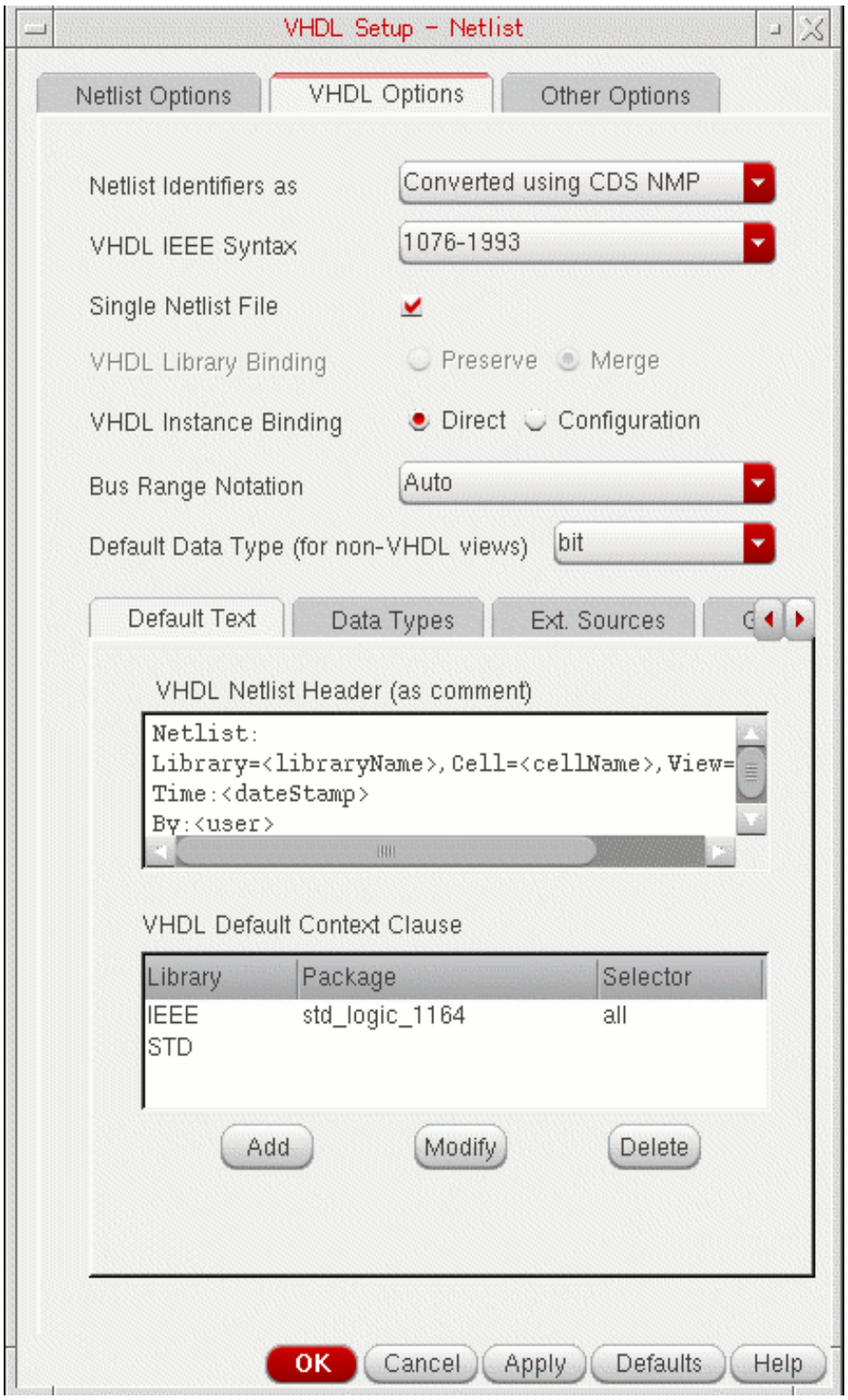
Option	Description
Enforce VHDL Binding for View(s)	Specifies a list of proxy symbol views to be used for resolution of inherited connections. For more details about inherited connections, refer to Setting up the VHDL Netlister for Inherited Connections
Skip Design Units	Specifies a list of libraries or library cells that are to be skipped while netlisting a design. If the library or library and cell name of the current instance appears in this list, the instance is ignored while netlisting and is not printed in the final netlist. To ignore design cells, you do not need to set the design properties as `nlAction=ignore` or `nlIgnore=vhdl`. You can only specify the library cells in the *Skip Design Units* table.

Setting up Netlisting Controls

Option	Description
Netlist CDF parameters for	Lists the simulators, which are used to determine the parameters that need to be translated into VHDL generics. You can select any of these simulators depending on your need. The netlister reads instParameters field of the simulation view simInfo section in the cell CDF to resolve the parameter list. Default: `NONE`
Netlisting Mode	Enables you to re-netlist the entire design. The VHDL Toolbox does not support the *Incremental* netlist mode. The *ReNetlist All* option regenerates files for every design unit in the hierarchy. Default: `ReNetlist All`
Evaluate Inherited Connections	Specifies that the inherited connections should be evaluated before generating netlist. The netlister evaluates the expressions for inherited connections, stored with the `netSet` and `netExpr` attributes, and prints extra cellview terminals or pseudo ports for the explicit and implicit terminals. Default: `On` For more information about support for inherited connections, refer to Setting up the VHDL Netlister for Inherited Connections
Analyze(Compile) Design Units	Specifies whether to compile the design units before elaborating the entire design. Default: `Off`
Elaborate Design	Specifies whether to elaborate the compiled design units. Default: `Off`
Enable Split Configuration	Specifies whether to generate the configurations of all the cellviews in your design, instead of generating a single top-level configuration. Default: `Off`

Option	Description
Generate Test Bench Template	Specifies whether to generate top level VHDL test bench template for the design automatically. A test bench file with a default name test_bench.vhd is saved in the run directory. Default: `Off` To view and edit the auto generated test bench, choose *Commands – Edit Test Bench*. For more details, refer to Chapter 4, “Creating a Testbench.”.
Detailed Report	Specifies whether to display the debug messages generated while netlisting a design on Virtuoso Studio Design Environment workbench. These messages are automatically added to the CDS.log file. The messages include the information on how the netlister has resolved the inconsistencies while netlisting a design. Turn this option On if you need to display the messages. Default: `Off`
Max Errors	Specifies the maximum number of errors after which design compilation or elaboration aborts. Default: `10`

VHDL Options Tab



Option	Description
Netlist Identifiers as	Lists the options to specify whether the identifiers, such as instance names and net names, should be converted using CDS NMP or their case should be preserved. By default, the value of Netlist Identifiers as is set to Converted using CDS NMP. With this option, the netlister uses Cadence name mapping and OSS tabular format. It translates all identifiers that are illegal in VHDL or contain uppercase characters to valid VHDL names using escaped name style. For example, an identifier aBc is translated as \aBc\ to preserve the uppercase character during simulation. To keep the case of the identifiers, set the value of Netlist Identifiers as to Case Preserved. In this case, an identifier aBc is saved with same name as aBc, but the uppercase of characters will not be preserved during simulation.
VHDL IEEE Syntax	Specifies the version of VHDL IEEE standard that is used to generate a netlist. The netlists are generated using any of the two versions, `1076-1993` or 1076-1987.
Single Netlist File	Specifies whether to create a single netlist file or multiple netlist files for VHDL design units. A single netlist file is generated with VHDL description of all non-primitive design units, by default. Turn this option `off` if you need to create multiple netlist files. Default: `On` For more information about analyzing netlists, refer to Analyzing the Netlist

Option	Description
VHDL Library Binding	Specifies the compile library to which the netlister binds the design units. The netlister has two binding options, *Preserve* and *Merge*. The *Preserve* option binds the design units to the associated DFII library and the *Merge* option binds the netlisted design units to a single logical VHDL library, `work`. If the *Single Netlist File* option is on, the *Merge* library binding option is also on by default. However, if you need to switch to the *Preserve* option, you should first turn off the *Single Netlist File* option. When the VHDL library binding is set to *Preserve* and the `hnlVHDLConfigUseLibNameForStoppingCell` variable is set to `t`, the config file prints the specified library name, instead of the single logical VHDL library `work` for stop cells as well.
VHDL Instance Binding	Indicates that the instances are directly bound into the architectural description. The netlister has two instance binding options, *Direct* and *Configuration*. The default instance binding option is *Direct*. For more information, refer to Support for Instance Bindings.

Option	Description
Bus Range Notation	Specifies the direction in which bus ranges should be printed for all vector signals. You can set this option as one of the following values: ■ `Ascending`: Bus ranges should be printed in ascending order. For example, `PORT ( a : IN bit_vector( 1 TO 5 ) )`. ■ `Descending`: Bus ranges should be printed in descending order. For example, `PORT ( a : IN bit_vector( 5 DOWNTO 1 ) )`. ■ `Auto`: Bus direction should be same as specified in the schematic. If the bus ranges are given in both the directions and the Bus Range Notation is set as `Auto`, it is not defined how the bus ranges will be printed in the netlist. The netlister can print the bus ranges in any direction. Therefore, in these cases, it is recommended to set the value of this field as `Ascending` or `Descending`. Default: `Auto` **Note:** For the default mode, to print the vector signals in the order specified in the design, set the `hnlVHDLMergeSignals` variable to `t`. If a design has a descending bus and this variable is set to `t`, then the signal will be printed in descending order, irrespective of whether ascending part has more number of bits. However, if the design contains a complete bus (`a<0:7>`) containing all of the bus bits (`a<0:7> a<1>`, and `a<5:4>`), the signal will be printed in ascending order.

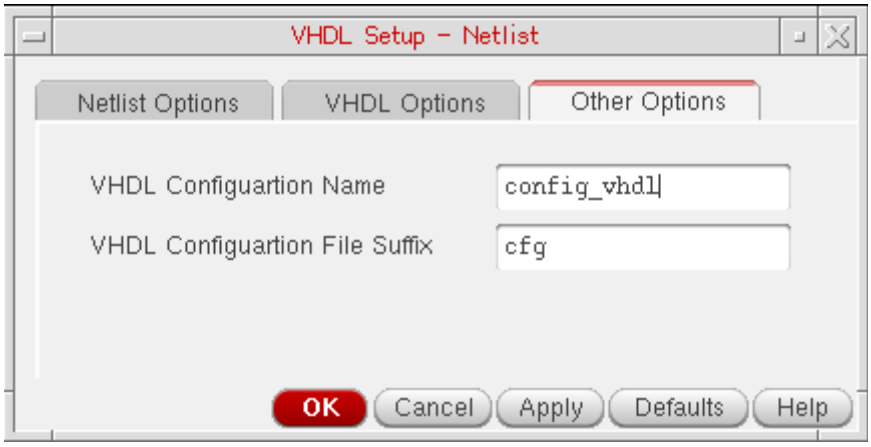
Option	Description
Default Data Type (for non-VHDL Views)	Contains a list of default data types, which can be used for the ports and signals in a netlist, if no information can be extracted from a design. A default signal scalar type (example: STD_ULOGIC) and vector type (example: STD_ULOGIC_VECTOR) is specified. You can modify these values when a signal type cannot be derived from an external VHDL model. Default: `bit` You can configure port type and port direction by using the `.simrc` file. For more information on inherited ports, refer to Configuring Pseudo Ports.
Default Text	Contains the following information: ■ VHDL Netlist Header: Allows you to add a default comment string before each netlisted cellview (entity/architecture/ package/configuration). The comment string is parsed for the keywords `<libraryName>`, `<cellName>`, `<viewName>`, `<user>`, and `<dateStamp>`. When these keywords are in the comment string, these are replaced by their respective values during netlisting. ■ VHDL Default Context Clause: Lists the package names to be inserted in the context clause of each VHDL design unit generated by the netlister. A context clause consists of: ❑ Library ❑ Package ❑ Selector You can add, modify, or delete the value of the context clause. **Note:** You can configure the indentation of the text by setting the `hnlVHDLIndentText` flag to `t`. You can also set the maximum line length by setting the `hnlMaxLineLength` flag.

Option	Description
Data Types	Allows you to specify user-defined data types that may be used in a design. The VHDL netlister need to know the scalar and vector names of a given data type because in VHDL, scalar and vector are different types. The data types can be specified in a tabular format in the following fields: ■ Scalar: Specifies the scalar name of the data type, such as `bit`. ■ Vector: Specifies the vector name of the data type, such as `bit_vector`. ■ Format: Specifies the printing format, such as `LOGIC`, `INTEGER`, `REAL`, `STRING`, and `BOOLEAN`. ■ Def Value: Specifies the default value printed for the ports of this data type. **Note:** During netlisting, this default value is printed for pins if the variable `hnlVHDLPrintPortInitialValue=t` (default value). For example, if you set the default value for a scalar std_logic as `0', then it will be printed as follows: `ENTITY and2_sch IS` `PORT(` `c : OUT std_logic := '0';` However, if `hnlVHDLPrintPortInitialValue=nil`, then the default value is not printed. For example, `ENTITY and2_sch IS` `PORT(` `c : OUT std_logic;` You can add, modify, or delete the data types, if required.
Ext. Sources	Lets you specify external VHDL sources to be used during netlisting. The *Ext. Sources* tab contains the *VHDL External Source Directories* table in which you can specify libraries and their directory paths. For more information, see Support for External VHDL Source Files on page 55.

Option	Description
Generic Defaults	Allows you to add, modify, or delete generics for an instance or cell. For more information about generics, see Modeling Instance Properties as Generics on page 100.

Other Options Tab

The following figure shows the *Other Options* tab on the VHDL Setup - Netlist form:



The various options on the *Other Options* tab are:

Option	Description
VHDL Configuration Name	This option provides the control to specify the configuration name while generating instance binding in `config` mode. For more information, refer to VHDL Instance Binding option. Default: `config_vhdl`
VHDL Configuration File Suffix	This option displays the suffix for the configuration file of your design. However, this feature is no longer supported. Default: `-cfg`

Setting up the VHDL Netlister for Inherited Connections

Inherited connections allow you to create global signals and override their names for selected branches of the design hierarchy. This section explains how VHDL Toolbox provides support to evaluate inherited connections during netlist generation.

An inherited connection is defined using the `netExpr` property in a lower-level schematic, which can also be overridden using the `netSet` property in a higher-level schematic. VHDL language does not support textual attributes such as `netSet` and `netExpr`. Therefore, the netlister evaluates these attributes and prints extra ports, called pseudo ports, in the netlist. The pseudo ports are created in the entity of the cellview being translated and is used to pass the inherited data signal up and down in the hierarchy. The names of these ports are prefixed with `inh_` and are also assigned a default data type, which is obtained from the instance connection in the schematic design. For example,

entity myinv is

```
port (
a : in bit;
y : in bit;
inh_powr : inout std_ulogic );
```

here, `inh_powr` is a pseudo port. The direction of the pseudo ports is always `inout`.

The connection from an instance terminal to an inherited signal is either a resolved `netSet` value or a pseudo terminal net. In the following example, `inh_powr` is a resolved net,

```
inst0: entity work.myinv
port map (
   a => neta,
   b => netb;
   inh_powr => vdd_global
);
```

In the following example, `inh_powr` is a pseudo terminal net,

```
inst1: entity work.myinv
port map (
   a => neta,
   b => netb;
   inh_powr => inh_powr
);
```

At the top level, all inherited connections are resolved to the `netSet` value or the default value of `netExpr`. Therefore, no pseudo ports exist at the entity interface.

For more details on inherited connections, see **Chapter 2, Understanding Connectivity and Naming Conventions of Virtuoso Schematic Editor L User Guide**.

You can specify the choice to evaluate inherited connections in three ways (listed in the order of preference from highest to lowest):

- Setting the Evaluate Inherited Connections option in the *Netlist Options* tab of *VHDL Setup - Netlist* form. This will automatically set the `simPrintInhConnAttributes` SKILL variable as `nil` in the `si.env` file.
- Setting the `simPrintInhConnAttributes` SKILL variable as `nil` in the `.simrc` file
- Setting the `simPrintInhConnAttributes` SKILL variable as `nil` in the CIW

Configuring Pseudo Ports

Pseudo ports are by default prefixed with inh_. You can customize this prefix string as per your requirement using the `hnlInhConnPrefix` variable in the `.simrc` file. If you do not want to prefix the pseudo ports with any string, set this variable to "".

The default port type of the pseudo ports is `INOUT` and their data type is same as the default VHDL data type set using the `hnlVHDLDefaultDataType` variable. However, you can set a different default data type and port type for these ports using the `hnlVHDLDefaultInhPortDataType` and `hnlVHDLDefaultInhPortMode` variables, respectively.

Support for Implicit Inherited Connections for Leaf Cells

By default, the netlister does not consider the implicit inherited net expressions on wires of leaf (stopping) cellviews. To honor such inherited net expressions, set the following variables in the `.simrc` file:

- Set the `simResolveStopCellImplicitConns` skill variable as `t`
- Set the `simPrintInhConnAttributes` variable as `nil`

When these flags are set, the netlister evaluates the implicit inherited net expression on wires of the stopping cellviews and creates pseudo ports at the top level, wherever required.

Support for Proxy Views

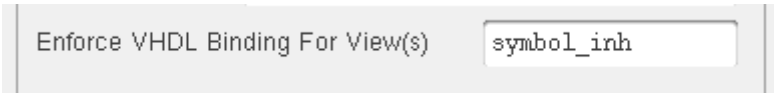
VHDL text views do not support textual attributes for inherited connections. Therefore, to print the inherited connections in the netlist, you need to create proxy views that contain

description of the desired inherited pins and terminals. This section explains how the VHDL Toolbox provides support for printing inherited connections in the netlist by using proxy views.

The *Netlist Options* tab on the VHDL Setup - Netlist form contains the *Enforce VHDL Binding for View(s)* field, shown in the figure below. This field enables you to specify a list of proxy symbol views to be used for resolving inherited connections.

The netlister uses these proxy symbol views, in place of actual VHDL views to resolve inherited connections. After resolving the inherited connections, the netlister uses the actual VHDL text views for data type propagation. If proxy symbol views are not specified, the netlister directly uses actual VHDL text views. In this case, the inherited terminals are not resolved.

Default: `symbol_inh`



The following restrictions apply to the resolution of inherited terminals using proxy views:

- The actual VHDL view must have a valid OpenAccess database.
- If the proxy view has terminals with the `nlAction` property set as `ignore`, those terminals are ignored during netlisting. Such terminals should have a default value in the actual VHDL text view for successful compilation.

The VHDL-OSS netlister supports the use of proxy views by using the same proxy for multiple VHDL architecture views while binding cells in HED. Therefore, you can bind cells or instances to multiple VHDL architectures and specify one proxy view for resolving inherited connections.

To support proxy views, the following prerequisites must be met:

- All architecture-entity pairs are created by using DFII 5x cellviews before binding.
- All architecture-entity views have a valid OpenAccess database.
- Proxy symbol, which include the port name, width, and direction, are in sync with the underlying VHDL text views.

When the netlister finds an instance with VHDL view binding, it uses the first-found view in the proxy view list to resolve inherited connections. If no proxy views are specified in the list, the actual VHDL view is used, which is the same behavior as without proxy views.

Important

Proxy views should not be specified in the Switch View or Stop View lists of hierarchical specifications.

Note: The proxy view is enforced only for VHDL text views. For all non-vhdl views, such as schematic, Verilog, SV, or Verilog-a, the proxy view is not used. Proxy view support is applied on the basis of view types instead of view names. IC 6.1.5 release supports only VHDL-D views.

Additional Features of the Netlister

Support for Generating the Configurations of All the Cellviews in a Design

You can generate the configurations of all the cellviews in your design, instead of generating a single top-level configuration of the design using `hnlVHDLSplitInstsInConfig`. To enable this feature, set `hnlVHDLSplitConfigEnabled` to `t`. For example, you can add the following entry in the `.simrc` file.

```
hnlVHDLSplitConfigEnabled = t
```

Important

The variable `hnlVHDLSplitConfigEnabled` works with VHDL IEEE 1993 format with the configuration mode enabled in the netlister.

The following snippets illustrate parts of a netlist generated when `hnlVHDLSplitConfigEnable` is set to `t`. Note the bold text that illustrates how the configuration of a cellview is included in the netlist.

```
--Netlist:
...
...
LIBRARY IEEE,STD;
USE IEEE.std_logic_1164.all;
CONFIGURATION xor_schematic_config OF \xor\ IS
  FOR schematic
    FOR ALL: myinv_logic
     USE ENTITY WORK.myinv_logic(behavioral);
    END FOR;
    END FOR;
    FOR \I1\: myand2
     USE ENTITY WORK.myand2(behavioral);
    END FOR;
    FOR ALL: myor2
     USE ENTITY WORK.myor2(dataflow);
    END FOR;
    END FOR;
END CONFIGURATION xor_schematic_config;
...
...
-- Netlist:
-- Library=testlib,Cell=top1,View=top1_schematic_config
...
LIBRARY IEEE,STD;
USE IEEE.std_logic_1164.all;
CONFIGURATION top1_schematic_config OF top1 IS
  FOR schematic
    FOR ALL: myxor
      USE ENTITY WORK.myxor(behavioral);
    END FOR;
    FOR \I3\: \generate\
      USE ENTITY WORK.\generate\(behavioral);
    END FOR;
    FOR ALL: \xor\
      USE CONFIGURATION WORK.xor_schematic_config;
    END FOR;
    END FOR;
END CONFIGURATION top1_schematic_config;
```

The following snippets illustrate parts of netlist of the same design when `hnlVHDLSplitConfigEnable` is not set or set to `nil`. Note that the configuration of the cellview is missing and the top-level configuration is in the expanded form.

```
--Netlist:
...
...
-- Netlist:
-- Library=testlib,Cell=top1,View=top1_schematic_config
...
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
...
CONFIGURATION config_vhdl OF top IS
  FOR schematic
    FOR mb1: mid|
      USE ENTITY WORK.mid(behavioral);
      FOR behavioral
        FOR lb1: myor2
          USE ENTITY WORK.myor2(behavioral);
        END FOR;
        FOR lb2: myor2
          USE ENTITY WORK.myor2(behavioral);
        END FOR;
      END FOR;
    END FOR;
    FOR mb2: mid
      USE ENTITY WORK.mid(dataflow);
      FOR dataflow
        FOR lb1: myor2
          USE ENTITY WORK.myor2(dataflow);
        END FOR;
        FOR lb2: myor2
          USE ENTITY WORK.myor2(dataflow);
        END FOR;
      END FOR;
    END FOR;
  END FOR;
END CONFIGURATION config_vhdl;
```

Support for Instance Bindings

You can set VHDL Instance Binding Direct or Configuration option on the VHDL Options tab. If the instance binding is set as *Direct* and the syntax is IEEE 1993, all the instances are bound directly to their library/entity/architecture in the architectural description. If the syntax is IEEE 1987, instances are bound to components and component declarations and configuration specifications are printed in the architecture.

If the instance binding is set as *Configuration*, all instance bindings are deferred in a separate VHDL file called `VHDL config file`. Here, instances are bound to components and only component declarations are printed in the architectural description. With the *Configuration* option, the VHDL netlister enables you to generate two views: *compact* and *detailed*. By default, the VHDL Netlister generates a compact configuration view.

To generate a detailed configuration view such that binding is present for each instance, set the `hnlVHDLSplitInstsInConfig` variable to `t`.

To generate a configuration view that contains detailed binding for specific cells, provide the list of cells to the `hnlVHDLSplitInstsInConfig` variable. For example, if you want to generate instance configuration in the netlist for instances `myinst1` and `myinst2`, set the variable as follows:

```
hnlVHDLSplitInstsInConfig = '("myinst1" "myinst2")
```

Example of a Compact and Detailed Configuration View

Compact configuration view

```
FOR ALL: pm_capa_unitaire
    USE ENTITY
    AFE.pm_capa_unitaire(vhdlrn_functional);
```

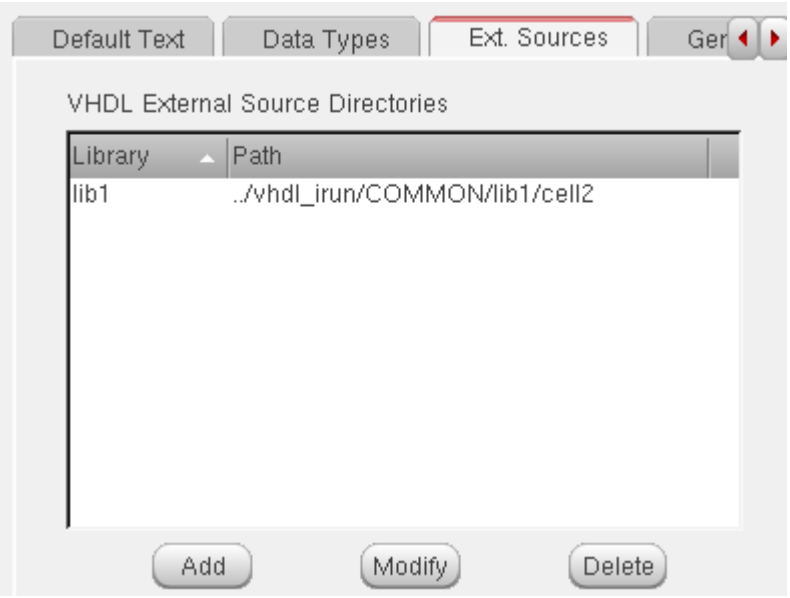
Corresponding detailed configuration view

```
FOR cap_lr_0 : pm_capa_unitaire
USE ENTITY
AFE.pm_capa_unitaire(vhdlrn_functional);
END FOR;
FOR cap_lr_1 : pm_capa_unitaire USE ENTITY
AFE.pm_capa_unitaire(vhdlrn_functional);
END FOR;
FOR cap_lr_2 : pm_capa_unitaire USE ENTITY
AFE.pm_capa_unitaire(vhdlrn_functional);
END FOR;
```

Note: You can print the keyword `OTHERS` instead of `ALL` by setting the SKILL variable `hnlVHDLConfigSpecForCommonMaster="OTHERS"` in the `.simrc` file. The default keyword is `ALL`.

Support for External VHDL Source Files

➡ Click *Ext. Sources*, as shown in the figure below.



Each directory path can have multiple VHDL files. The tool identifies all the VHDL files in these directories as the external source files, parses them to extract the cell and port details, and binds them to the specified library (irrespective of the *VHDL Library Binding* option being set as `Merge` or `Preserve`). By default, the tool identifies files with `.vhd` or `.vhdl` extensions as VHDL files.

You can also specify the external source directories using the SKILL variable `hnlVHDLExternalDirList`. To specify a different set of file extensions, use the SKILL variable `hnlVHDLDefFileExts` .

Important

If the name of an external VHDL cell entity is same as that of a design cell, the schematic of the design cell is treated as stop view and the hierarchy is not traversed further. Therefore, even if the schematic view of the design cell is not specified in the Stop View List, it will be treated as a stop view in this case.

VHDL text views may not exist for some cells, for such cells you need the VHDL Netlister to refer to external text files for specifying the port datatypes. The external VHDL sources can be specified in following ways:

- As a list of directories that contain the VHDL source files where every file in the directory has a `.vhd` or `.vhdl` extension. These files are parsed to gather information, such as cell names and ports data type.

The default extensions are controlled by the `hnlVHDLDefFileExts` skill flag.

- During design traversal, no further hierarchy expansion takes place if there is any DFII cell name that matches the entity name from the external sources.

While generating netlist using the external source files, the VHDL Toolbox considers the following points:

- If Cadence name mapping (CDS NMP mode) is ON (that is, if the `hnlVHDLDonotUseCdsNmp` variable is set to `nil`), while compilation, the tool changes the case of the external entity names to lowercase. While comparing the names of design cells with the names of external entities, the tool considers the changed names. Depending on the CDS NMP mode, the results of cell name comparison will be as shown in the following table:

Cell Names	**Results after Comparison**
VHDL entity: `Mid` DFII cell: `Mid`	Match if the CDS NMP mode is off
VHDL entity: `Mid2` DFII cell: `mid2`	Match if the CDS NMP mode is on
VHDL entity: `mid3` DFII cell: `Mid3`	Always mismatch
VHDL entity: `\Mid4\` DFII cell: `Mid4`	Matches irrespective of the CDS NMP mode because the entity name has been escaped.

- If the name of a design cell matches with the name of an external entity, the netlister honors the external entity. The instance line printed by the netlister contains the instance name bound to the library name specified for the external source cell.
- During the first netlist run, the VHDL Toolbox parses all the files given in the external directories. In the following runs, the tool parses only new files. It parses an existing file only in the following cases:
  - If the file was modified after the previous run
  - If the library binding of the file is modified after the previous run
  - If the file was not parsed in the previous run because of errors
- VHDL Toolbox checks for the presence of multiple declarations for an entity. It reports errors in the following cases:

- If the external source data has multiple files (in same or different libraries) with same entity declaration.
- If a vhdl text view has same name as that of an external entity.
- If multiple stop cells that do not have vhdl text views (blackboxes) with same cell names are present in different DFII libraries. By default, this is not checked. If you want the netlister to check for this case, set the *hnlVHDLCheckSameStopCellFromMultLibs* SKILL variable to t.

- By default, while compiling the external source files, the VHDL Toolbox reports all the compilation errors as errors. You can change the severity of these messages as warning by setting the *hnlVHDLParseExtDataSeverity* SKILL variable to `"warning"`. By default, this variable is set as `"error"`.
- While parsing the external source files, if the VHDL Toolbox finds errors in a file, it reports the errors in the CIW and stops further compilation. This is because, by default, the *hnlVHDLMaxExtFileErrCount* SKILL variable is set to `1`. If you want the tool to report errors for more than one file, say 10 files, set the *hnlVHDLMaxExtFileErrCount* SKILL variable to `10`. In this case, the tool will continue compilation until it finds errors in ten files or it parses all the files successfully, whatever happens first.
- After netlisting, the VHDL Toolbox maintains the compiled data in the cache. You can clear the cache by setting the *hnlVHDLResetExternalVHDLData* SKILL variable to `t`.

Support for Type Conflict Resolution

During hierarchical netlisting, VHDL-OSS netlister propagates data types of intermediate connecting signals and cellview ports by using a process called type propagation. Type propagation can lead to conflicts when:

- A single signal is connected to multiple instance terminals of different data types.
- A vector signal has different data types on multiple bits.

In such cases, the netlister displays warning or error messages, but continues to generate the complete netlist, which is incorrect and cannot be elaborated.

To avoid such conflicts, you need to apply a resolution or a type conversion scheme. A resolution scheme involves providing a priority or type resolution table to find a resolved data type (see Using Type Resolution Tables on page 58). A type conversion scheme involves using a user-specified function that converts a signal of data type 'typeA' to data type 'typeB' (see Using Type Conversion Functions on page 58).

Using Type Resolution Tables

You can define the type resolution table by setting the `hnlVHDLTypeResolverDefs` SKILL variable in `.simrc`. During netlist generation, the tool searches the type resolution table to resolve any type conflicts. However, if any conflict is not resolved, the tool reports a warning message and completes the netlist.

If you want to change this default behavior so that the tool reports errors in case of unresolved type conflicts and discontinues the netlist generation, set the `hnlVHDLTypeConflictSeverity` SKILL variable as `"error"`. By default, this variable is set as `"warning"`.

Following are some examples to show how you can specify entries in the type resolution table:

Example 1:

```
hnlVHDLTypeResolverDefs = '("typeA" "typeB" "typeC")
```

This implies that if the netlister finds a signal or a signal bit with two propagated data types, `typeA` and `typeB`, the resolved data type to be used is `typeC`. Here, `typeC` can be any one of `typeA` and `typeB` or any other data type.

Example 2:

```
hnlVHDLTypeResolverDefs = '(("typeQ" "typeW" "typeQW") ("typeE" "typeR" "typeER") )
```

The above example provides a list of entries for the type resolution table.

Example 3:

If there are more than two conflicting types on a single signal, it is recommended that you provide all type conflicting pairs in the list so that complete type resolution can be done. For example, if signals of three different types: `typeA`, `typeB`, and `typeC` are inflicted upon a single wire, you should provide the type resolution entries for all the following cases:

```
hnlVHDLTypeResolverDefs = '(("typeA" "typeB" "typeR") ("typeA" "typeC" "typeR")
("typeB" "typeC" "typeR"))
```

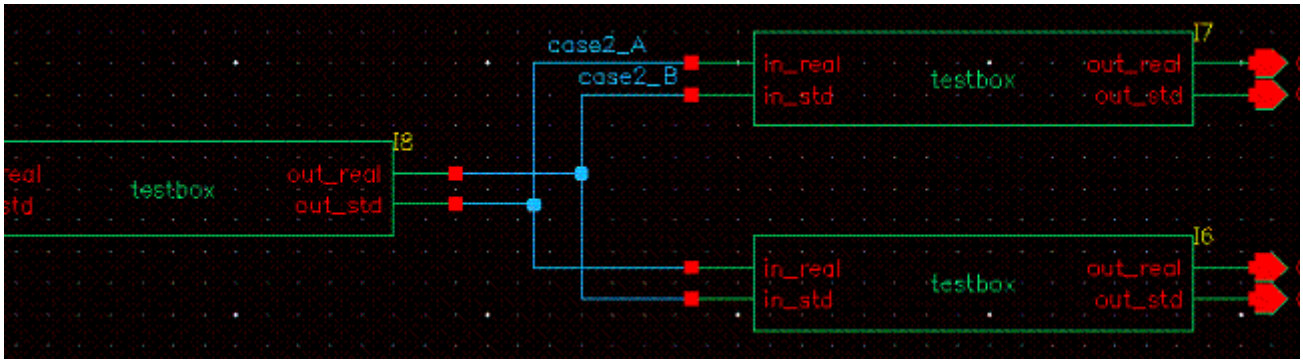
Using Type Conversion Functions

When a netlister finds a signal connected to several instance terminals of different types, it prefers the data type of the output instance terminal and assigns it to the signal. The netlister then adds a conversion function at the instance terminal signal (in PORT MAP section of the netlist) wherever the data type of the instance terminal is different from that of the preferred data type.

For example, in the figure below, signal *case2_A* on the output terminal *I8/out_std* has data type *std_logic* and is connected to an input terminal *I7/in_real* with data type *real*. Therefore, a type conversion function that converts the signal from *std_logic* to *real* is added at the PORT MAP of input terminal *I7/in_real.*

By default, the PORT MAP information is printed using named port mapping. However, this information can also be printed using positional association by setting the `hnlVHDLPositionalAssociation` variable to `t`. For example, if this variable is set, the PORT MAP information would appear as

```
PORT MAP(
OUT2_A,OUT2_B,std_logic2real(\case2_A\),real2std_logic(\case2_B\)));
```




```
---------------------- x ------------------------------
ARCHITECTURE schematic of \TOP_test_vhdl\ IS
-- Local SIGNALS and ALIASES ---
SIGNAL \case2_B\ : real;
SIGNAL \case2_A\ : std_logic;
…
begin
\I7\: ENTITY WORK.testbox(behavior)
PORT MAP(
out_real => \OUT2_A\,
out_std => \OUT2_B\,
in_real => std_logic2real(\case2_A\),
in_std => real2std_logic(\case2_B\)
);

\I6\: ENTITY WORK.testbox(behavior)
PORT MAP(
out_real => \OUT2_C\,
out_std => \OUT2_D\,
```

```
in_real => std_logic2real(\case2_A\),
in_std => real2std_logic(\case2_B\)
);
END schematic;
---------------------- x -------------------------------
```

Every time a conversion function is added at the PORT MAP of the input terminal, the netlister displays the following message in CIW:

“In schematic '<l.c.v>', the signal(net) '<sig>' on instance terminal '<inst>:<termName>' having type '<typeA>' is connected to other instance terminals having type '<typeB>'. Therefore, a conversion function '<func>' will be added to resolve the conflict.”

Note: Conversion functions are applied globally on the complete design. Currently, a local type conversion mechanism for individual instance ports is not available.

For type conversion to work, the following conditions must be satisfied with respect to the conflicting signal:

- The direction or mode of the connected instance ports of the conflicting signals must be input or output. The inout mode is not allowed.
- The connected output instance ports must have the same data type.
- The conversion functions for all type conflicts must be defined.

  Note: The direction of an instance terminal is inferred from its switch master, and not the place master.

You can define a type conversion function by setting the *hnlVHDLTypeConversionFuncs* SKILL variable in CIW or in the `.simrc` or `si.env` files as follows:

```
hnlVHDLTypeConversionFuncs = list( "typeA" "typeB" "fn_typeA2typeB"
"fn_typeB2typeA" )
```

For example, to convert data type *typeA* to data type *typeB*, the conversion function *fn_typeA2typeB* is applied, and vice versa.

For specifying several type conversion functions this list can be changed to a 'list of list' as shown in the following example:

```
hnlVHDLTypeConversionFuncs = list(
    list( "typeA" "typeB" "fn_typeA2typeB" "fn_typeB2typeA" )
    list( "typeC" "typeD" "fn_typeC2typeD" "fn_typeC2typeD" )

)
```

Conflicts.log file

A `conflicts.log` file displaying resolved and unresolved conflicts is created when your design uses a type conversion function.

A typical `conflicts.log` file displaying resolved conflicts is as follows:

```
--------------------------------------------------------------------------------
RESOLVED CONFLICTS (testlib.basic1:schematic):
--------------------------------------------------------------------------------
Type conflict detected on net: 'testlib.basic1:schematic/r2std'. Final type is:
'real'.
On this net types are (inst/term:direction:<type>):
    I0/out_real:OUT:<real>
    I1/in_std:IN:<std_logic> conversion function added: 'real2std_logic'
```

In this file, a conflict (that was resolved) is detected on the net r2std in the lib/cell:view testlib.basic1:schematic. The final resolved type is real.

A typical `conflicts.log` file with unresolved conflicts is as follows:

```
--------------------------------------------------------------------------------
UNRESOLVED CONFLICTS (TOP.TOP_test_vhdl:schematic):
--------------------------------------------------------------------------------
Type conflict detected on net: 'TOP.TOP_test_vhdl:schematic/IN1'. Final type is:
'std_logic'.
On this net types are (inst/term:direction:<type>):
    I38/in_std:IN:<std_logic>
    I37/inout_real1:INOUT:<real>
    I37/inout_real2:INOUT:<real>
Type conflict detected on net: 'TOP.TOP_test_vhdl:schematic/IN2'. Final type is:
'real'.
On this net types are (inst/term:direction:<type>):
    I2A/in_real:IN:<real>
    I2A/in_std:IN:<std_logic>
```

In this file, an unresolved conflict was detected on the net IN1 in the lib/cell: view TOP.TOP_test_vhdl:schematic. The final conflict selected was std_logic.

Note: In case of an internal conflict when only input terminals are connected to a wire, use the following variable to choose one type from the specified types: `hnlVHDLPriorityTypes`.

For example:

```
hnlVHDLPriorityTypes = list(?aType? ?bType?) ; aType is chosen over bType
```

Support for Shorting Devices

For OSS-based netlisters, you can short devices and replace them with a surviving net. For more information about how ports are shorted for devices, see "Removing Devices with Multiple Terminals" in *Open Simulation System Reference*:

To enable shorting for devices in the VHDL netlister, follow one of the methods listed below:

- Set the `lxRemoveDevice` string property at the instance level.

  You can set this property from the Property Editor assistant of Schematic Editor.

  Example:

  ```
  lxRemoveDevice(short(A B) short(C D E F))
  ```

  In this example:

  - `A` is shorted to `B`.
  - `C` is shorted to `D`, `E`, and `F`.

  Note: `lxRemoveDevice` does not work for read-only libraries. Therefore, for read-only libraries, use `hnlUserMultiTermShortCVList` or `hnlUserShortCVList` in the `.simrc` file.

- Use the `hnlUserMultiTermShortCVList` SKILL variable in `.simrc` to specify the cellview names and pin information in a list in the following syntax:

  ```
  hnlUserMultiTermShortCVList  = `(("lib" "cell" "view" "(short(A B) short(C D E)" ))
  ```

  You must provide a value in all the fields of this variable. Otherwise, the netlister reports an error.

  Examples:

  - Multiple cells with multiple ports

    ```
    hnlUserMultiTermShortCVList = `(("sample" "dffpp_c_" "symbol" "(short(in1 out3 out4))") ("testLib" "bottom" "symbol" "(short(in1 out2) short(in2 out3 out4 out5))"))
    ```

  - A cell with two ports

    ```
    hnlUserMultiTermShortCVList = `(("sample" "dffpp_c_" "symbol" "(short(in1 out5))"))
    ```

- To remove devices that have two terminal, use the `hnlUserShortCVList` SKILL variable in the following syntax that specifies the cellview names in a list.

```
hnlUserShortCVList = list(
                ;;; all cells from this library
                "libN"
                ;;; cell1, cell2 and cell3 from lib1
                list("lib1" "cell1" "cell2" "cell3")
                ;;; all cells from this library
                list("libM")  )
)
```

  You can specify any of the elements in the list and keep the remaining elements blank.

  Examples:

```
hnlUserShortCVList  = '("sample" "dffpp_c" "symbol")
hnlUserShortCVList  = '("sample" "dffpp_c" "")
hnlUserShortCVList  = list( list("analogLib" "res") list("testbottom"))
```

Notes:

- Cadence recommends that you use `hnlUserMultiTermShortCVList` instead of `hnlUserShortCVList`. If required, use `hnlUserShortCVList` for devices that have only two ports.
- OSS gives precedence to `hnlUserShortCVList` or `hnlUserMultiTermShortCVList` over `lxRemovedDevice`.

Regardless of the method you use to short devices, you need to set the SKILL variable `hnlHonorLxRemoveDevice` to `t` in `.simrc`. By default, `hnlHonorLxRemoveDevice` is not set.

Support for Shorting Terminals

You can short two terminals in a schematic by adding a special instance between the terminals. The default instance for shorting terminals is of the cellview `basic/cds_thru/symbol`. The `cds_thru` symbol is used, instead of the patch symbol that is used for other shorts, to avoid any extraction error associated with the connectivity of the shorted terminals.

If you want to use another library and cell for shorting terminals, instead of `basic` and `cds_thru`, use the following flags and insert the symbol between the two terminals:

- `vhdlLibForCdsThru`

  Set this flag to the custom library you want to use.

Example: `vhdlLibForCdsThru = "myLib"`

Default value: `"basic"`

- `vhdlCellForCdsThru`

  Set this flag to the custom cell you want to use.

  Example: `vhdlCellForCdsThru = "my_cds_thru"`

  Default value: `"cds_thru"`

When you generate the netlist of a design where two terminals are shorted, the resulting netlist contains a `cds_thru` instance line, and not an assign statement. This instance line makes the netlist different from the imported netlist, which typically has an assign statement for the shorted terminals. To avoid this mismatch, configure VHDL Toolbox to print the assign statement instead of the `cds_thru` instance line to represent shorted terminals. For this, use the following flag:

- `vhdlUseAssignForCdsThru`

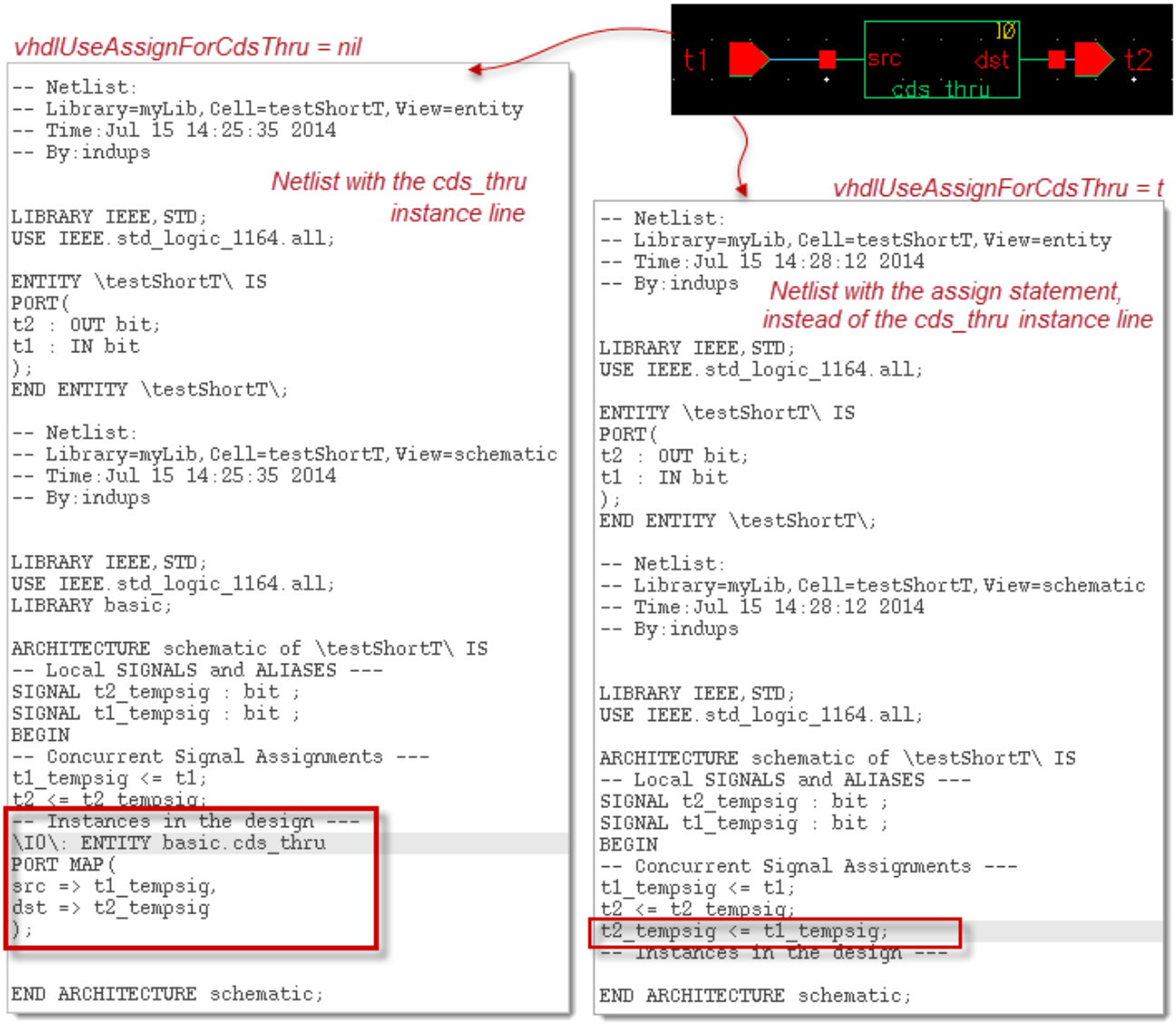
  Set this flag to `t` to use the assign statement to represent shorted terminals. VHDL Toolbox considers this flag even if you have used a custom library and cell for shorting terminals.

  Example: `vhdlUseAssignForCdsThru = t`

  Default value: `nil`

You can set the flags in Virtuoso CIW, `.simrc`, or `si.env`.

The following figure illustrates the netlist with the `cds_thru` instance line and the netlist with the assign statement.



Resolving Floating Nodes

For resolving floating nodes, set the `hnlVHDLPrintOpenForFloatingNode` flag to `t`. When this flag is set, the keyword `open` is printed in the PORT MAP section for nodes that have no explicit net connected.

```
PORT MAP(
    RH_EN => open
);
```

Generating the Netlist

After you have completed the netlist setup, you generate the netlist by choosing *Commands – Generate Netlist*. Alternatively, you can generate a netlist by using the *Netlist Design* option on the fixed menu.

The *Generate Netlist* menu option triggers the netlister to generate the VHDL hierarchical netlist for a design specified in the *Top Level Design* fields. The tool supports incremental netlist generation and also provides the option to analyze and elaborate the generated netlist files. The netlister stops netlisting after the maximum number of errors that you have set is reached. If netlisting is successful, the message *Netlisting Successful* appears on the status line. The variables related to the OSS tabular name mapping are initialized to ensure that the generated netlist contains all the names in the VHDL namespace.

To display a message box on successful completion or failure of a netlist, set the `hnlVHDLPopUpNetlistStatus` flag to `t`.

Viewing Netlist Results

You can view the netlisted file for your VHDL design by choosing *Results – Netlist*. The VHDL netlist is saved as a `.vhd` text file in the run directory.

Note: If you choose to create multiple netlist files for your VHDL design (by clearing the *Single File List* check box on the V*HDL Options* tab in the VHDL Setup - Netlist form), you cannot view the results. To view the results, ensure that the *Single File List* check box is selected.

Analyzing the Netlist

The netlist generated successfully by VHDL Toolbox is saved in a defined directory structure under the Run directory. The netlister creates the following files and directories to save the design details:

A set of `cds<n>` directories in the `<Run Directory>/ihnl` directory, where each `cds<n>` directory contains the entity and architecture description for each netlisted unit. The file containing the design entity is named as `<cell_name>-e.vhd` and the file containing the architecture description is named as `<cell_name>-<view_name>-a.vhd`.

A configuration file, suffixed as `-cfg.vhd`, containing the configuration details.

A global package named as `<top-cell-name>_cds_globals` saved in the `cds_globals` sub directory of the Run directory. This package contains declaration of all the global signals. For example:

```
package WORK.mytop_cds_globals is

  signal vdd_global : std_ulogic;

  signal \gnd!\ : bit;

end package;
```

The architecture design unit refers to this global package, as shown below:

```
use WORK.cds_globals.all;
```

If Cadence name mapping is used, the global signals are printed as escaped names. For example, `\gsig!\`. Otherwise, the `!` character in the global signals is by default mapped to the string `_global`. You can change this mapping by resetting the `hnlMapNetInName` SKILL variable.

If the *Single Netlist File* option on the *VHDL Setup - Netlist* form is turned `on`, the contents of all the files listed above are concatenated and saved in a single file named as `<topcellname>.vhd`. This file is shown in a separate window when you view the netlist using the *Results – Netlist* menu option.

4

Creating a Testbench

A testbench is typically an entity/architecture pair, which instantiates the top-level cell and provides stimulus to the Design Under Test (DUT). By using the VHDL Create Test Bench form, you can:

- Provide details about an external testbench for the top-level design that you want to simulate.
- Provide details to be used during auto-generation of the testbench by VHDL Toolbox.
- Edit the auto-generated testbench.

This chapter describes how you can use the VHDL Toolbox to configure properties for a testbench and the various methods used for creating a testbench.

In this chapter, you will learn about the following topics:

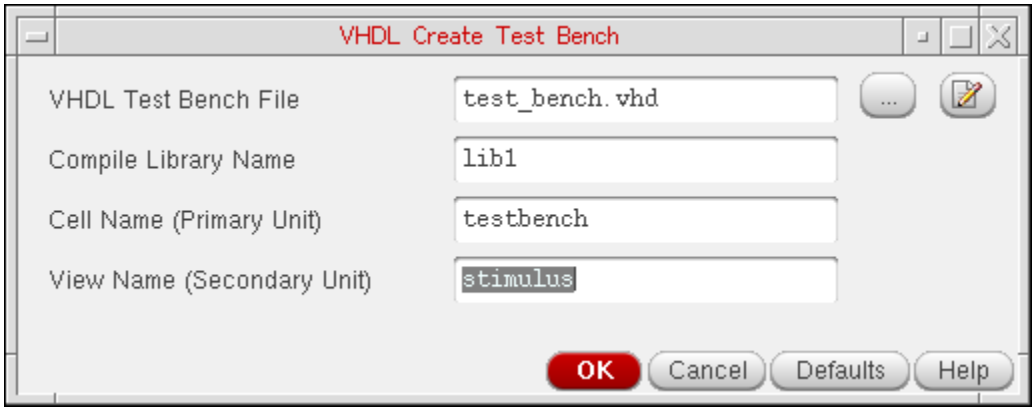
- VHDL Create Test Bench Form on page 69
- Methods to Create a Testbench on page 71

For information on the SKILL functions associated with VHDL Toolbox, see *Digital Design Netlisting and Simulation SKILL Reference*.

VHDL Create Test Bench Form

The VHDL Create Test Bench form contains fields that you can use to configure the properties of a testbench. To open the VHDL Create Test Bench form:

1. In the VHDL Toolbox window, choose *Commands – Edit Test Bench*.
   The VHDL Create Test Bench form appears, as shown in the figure below.



The various options on the VHDL Create Test Bench form are:

- VHDL Test Bench File: Specifies the name of input testbench file, which needs to be compiled with the generated netlists. When you specify the testbench file, you also need to ensure that the *Include the specified Test-Bench in simulation* check box on the VHDL Setup - Simulation form is selected. The current version of the VHDL Toolbox allows you to specify only a single testbench file. If the entity and architecture of a testbench are defined in separate files, you need to combine these files to get a single file.

  Note: If an auto generated testbench exists for current cellview in the specified run directory, the path to that testbench is displayed. To edit the testbench already created, click the edit button  next to the *VHDL Test Bench File* field. The testbench is opened in the default editor where you can make changes.

  Default: `test_bench.vhd`

- Compile Library Name: Specifies the library in which a testbench is compiled. A testbench is usually compiled in the same library as the top level design. However, you can change the value in the textfield to compile a testbench in a different library. The textfield contains the name of the top cell library by default.

- Cell Name (Primary Unit): Specifies the name of the entity in the testbench. The entity name is `testbench` by default. However, you can change the value in the textfield, if the entity in the testbench has a name other than `testbench`.

- View Name (Secondary Unit): Specifies the name of the architecture in the testbench. The architecture name is `stimulus` by default. However, you can change the value in the textfield, if the architecture in the testbench has a name other than `stimulus`.

Methods to Create a Testbench

You can create a testbench for the top level schematic in two ways:

- Generate a testbench automatically during netlist creation
- Attach an existing testbench with the schematic

Automatic Generation of Testbench

To create a testbench automatically, select the *Generate Test Bench Template* option on the *Netlist Options* tab of the VHDL Setup - Netlist form. When this option is used, the VHDL Toolbox creates a testbench and saves it as `test_bench.vhd` in the run directory.

The default structure of the testbench is as given below:

```
<Default Context Clause>

ENTITY <entname>
END [ENTITY] <entname>

<Default Context Clause>
<use lib clause>

ARCHITECTURE <archname> of <entname> IS
< COMPONENT DECLARATION>
< SIGNAL DECLARATION >
BEGIN
<INSTANCE NAME> : < BINDING >
PORT MAP (
        <formal_1> => <actual_1>
        …
    )
END [ARCHITECTURE] <archname>
```

The instances are bound in the testbench in the same way as done during netlist creation, as described below:

- In case of VHDL 93, the direct entity binding is used.
  For example, `<instname> : USE ENTITY <lib>.<topcell>(<view>).`

- In case of VHDL 87, the component and open binding is used.
  For example, `<instname> : <topcell>`.
- In case of config, the instances are bound to the config of the top cell.
  For example, `<instname> : configuration <lib>.<top_config_name>`

You can configure details, such as the file name or the library name of the testbench to be generated, using the VHDL Create Test Bench form. Alternatively, you can also set certain SKILL variables. For more details, refer SKILL Variables to Configure Testbench Creation.

To view and edit the auto generated testbench, choose *Commands – Edit Test Bench*. The VHDL Create Test Bench form appears. The name and path of the testbench file appears in the *VHDL Test Bench File* field. To open the test bench in the default editor and make changes to the file, click the edit button  .

Providing an Existing Testbench

If you have an existing testbench, provide its details to VHDL Toolbox using the VHDL Create Test Bench form. For more details about this form, refer to VHDL Create Test Bench Form.

SKILL Variables to Configure Testbench Creation

You can use the following SKILL variables to configure the testbench:

- `hnlVHDLGenTestBench`: Specifies if the testbench is to be generated automatically during netlist generation. By default, this variable is set as `nil`.
- `vhdlSimTestBenchLCV`: Sets the name of the library, cell and view for which testbench is to be created. By default, this variable is set as `'(<top_cell_lib> "testbench" "stimulus")`.
- `hnlVHDLTestBenchInstName`: Sets the name of instance for the testbench. By default, this variable is set as `dut`.
- `vhdlSimTestBenchFile`: Sets the name of the testbench file. By default, this variable is set as `test_bench.vhd`.

5

Simulating a Netlisted VHDL Design

After netlisting, you can simulate and debug your VHDL design. In the Virtuoso Studio design environment, you can simulate a design by using the `xrun utility` or the XMVHDL (xmsim) simulator. While the `xrun utility` is a quick one-step process, the XMVHDL simulator is a three-step process, which involves compiling, elaborating, and simulating the design.

Note: The executable and log file names will depend on the simulator being used. For the changes in the executable and log file names when using the Xcelium simulator, see Running Simulations with Xcelium on page 119.

You can set up and run the `xrun utility` from both, the command line interface and VHDL Toolbox, however, the XMVHDL simulator can be run only from the VHDL Toolbox.

The XMVHDL simulator enables you to:

- Collect simulation data in a database file
- Create your own programs to directly process simulation results
- Display simulation results using a waveform window
- Save the state of a simulation as a “golden” run so that it can be compared with a subsequent run.

This chapter describes the steps to simulate and debug your netlisted design using the VHDL Toolbox.

In this chapter, you will learn about the following topics:

For more information about using the command-line options for simulating your design, refer to Netlist and Simulate a VHDL Design. For information on the SKILL functions associated with VHDL Toolbox, see *Digital Design Netlisting and Simulation SKILL Reference*.

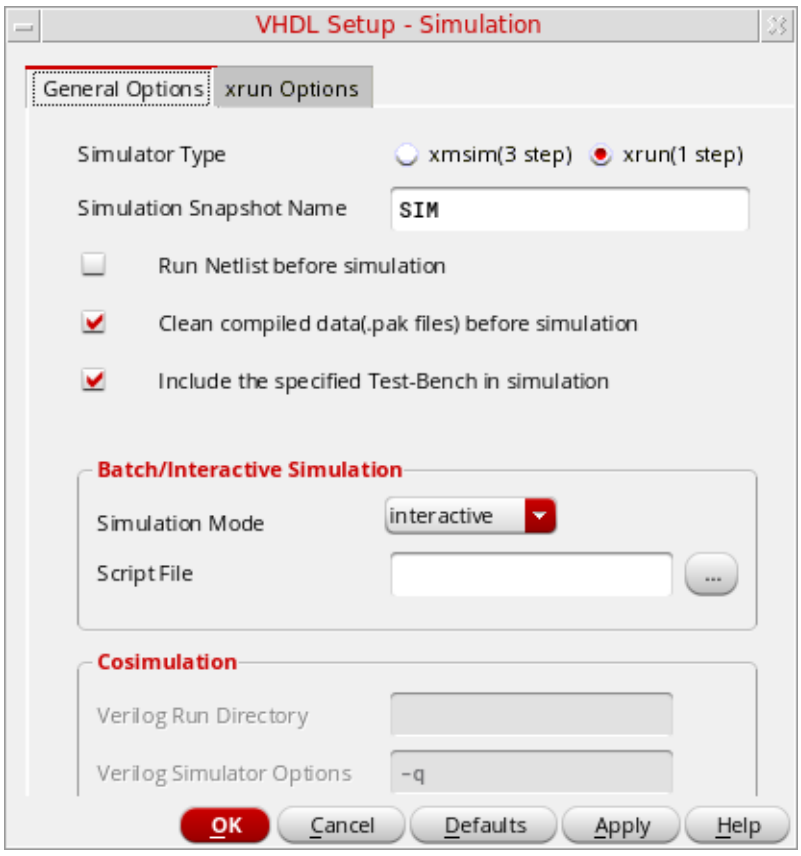
Simulating a Design Using the VHDL Toolbox

Before you can simulate a design by choosing *Commands – Simulate*, you need to set up the simulation options.

VHDL Setup - Simulation Form

To set simulation options for a VHDL design:

1. In the VHDL Toolbox window, choose *Setup – Simulation*.
   The VHDL Setup - Simulation form appears, as shown in the figure below.



The VHDL Setup - Simulation form consists of two tabs:

- General Options
- xrun Options

General Options Tab

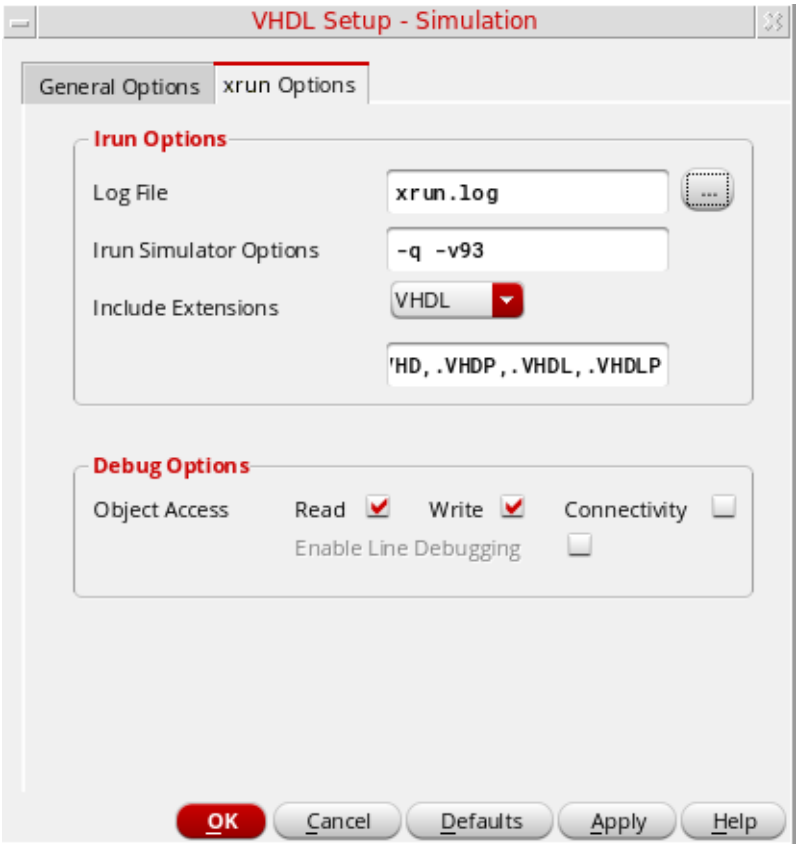
The figure displayed above shows the General Options tab of the VHDL Setup - Simulation form. This tab contains various options to control simulation.

- Simulator Type:
- Simulation Snapshot Name:
- Run Netlist Before Simulation : Ensures the creation of VHDL netlist before running the simulation. The netlister traverses the design hierarchy and converts all the schematics into VHDL design units. The traversal is defined by a switch list, a stop view list, and a stop library list specified in Traversal Settings. After netlisting, the hierarchy is converted into a VHDL configuration. You can choose this option when you are not netlisting the design explicitly before simulation.
- Clean compiled data(.pak files) before simulation: Deletes the elaboration or simulation results of the previous run before starting the simulation. If you do not select this option, the simulator reuses the results of the previous run.
- Include the specified Test Bench in simulaton: Indicates that the user supplied testbench file is also to be added to the list of design files after the specified design gets netlisted. The library, entity name, and architecture can also be specified if their values differ from default values. Once the complete design along with a testbench is compiled and elaborated, the simulator can honor the stimulus in testbench file and perform actions for the users. By default, this option is enabled.
- Batch/Interactive Simulation:
  - Simulation Mode
  - Script File
- Cosimulation:
  - Verilog Run Directory
  - Verilog Simulator Optionsp

Note: When you enable the *Xrun Options* tab, the *Cosimulation* section of the *General Options* tab is disabled.

Xrun Options Tab

The following figure shows the *Xrun Options* tab on the VHDL Setup - Simulation form:



The *Xrun Options* tab provides options for the *xrun utility* integrated with Virtuoso. The xrun utility provides a simple invocation process that lets you run the simulator by specifying all input files and command-line options on a single command line.

Note: By default, Virtuoso uses the `xmsim` simulator. Therefore, the *Xrun Options* tab is disabled. To enable the *Xrun Options* tab, set the `vhdlSimSimulator` SKILL variable to `xrun`. When you enable the *Xrun Options* tab, the *Cosimulation* section of the *General Options* tab is disabled.

When using the xrun utility, you need to set xrun specific options in addition to the options given in the settings done in the *General Options* tab. The various options on the *Xrun Options* tab are:

- Xrun Options: Contains the following fields to specify the settings to be used by the xrun utility:
  - Log File: Specifies path for the xrun log file.

Default: `xrun.log`

- Xrun Simulator Options: Specifies simulation options that are to be passed to the simulator and are not present in the Simulation Options form.

  Default: `-q -v93`. Here, `-q` specifies that all informational messages will be suppressed. `-v93` specifies that all VHDL-93 features supported in this release will be enabled.

- Include Extensions: Specifies a list of file extensions to be recognized by the xrun utility. xrun recognizes language of a file by the file extension. Therefore, you need to specify the file types to be recognized for a specific language. Select a language from the pull-down list and specify the file extensions to be recognized for that language. This list overrides the default extensions for VHDL files.

  By default, the xrun utility recognizes files with the following extensions as VHDL files:
  `.vhd,.vhdp,.vhdl,.vhdlp,.VHD,.VHDP,.VHDL,.VHDLP`

  By default, the xrun utility recognizes files with the following extensions as Verilog files:
  `.v,.vp,.vs,.V,.VP,.VS`

- Debug Options: Specifies the debug options. You can specify the type of access required while compiling the design. No special access is required for viewing the hierarchy or for finding the names of objects (nets, regs, variables, or scopes) in the design. However, you need the following types of access for more control on objects:

  - Read access is required to probe nets, regs, and variables (including setting PLI callbacks) and to get the value of these objects.

  - Write access is required to interactively set the value of simulation objects (depositing or forcing variables). Write access automatically provides read access.

  - Connectivity access is required to get driver and load information about a specific net, reg, or other variable. Connectivity access automatically provides write and read access.

  - Enable Line Debugging is required to set breakpoints at source lines or to apply statement callbacks. by default, this option is not checked. Using this option automatically provides read, write, and connectivity access.

Note: When you use the xrun utility, the settings done using the VHDL Setup - Simulation form are saved as a set of SKILL variables in the `.vhdlrc` file in the run directory. Inputs from this file are passed to the xrun utility. When you use the same run directory again, the options set in the .vhdlrc file are restored. For more details on the SKILL variables, refer to the Setting VHDL HNL Variables section in Appendix B.

Simulating the VHDL Design

Simulation first checks the design. If the check is successful, the VHDL simulator starts in batch or interactive mode on the local machine, as specified in the setup. The VHDL simulator triggers a three-step process to compile, elaborate, and simulate a design. When you click *Simulate,* the registered compiler (`xmvhdl`) is invoked on the generated netlist. A testbench, if available, is also compiled with the design. After the design is compiled, it is elaborated using the registered elaborator (`xmelab`). The elaborated design is then passed on to the registered simulator (`xmsim`). You can run the simulation either in interactive or batch mode as required.

To simulate your VHDL design:

➡ Choose *Commands – Simulate*.
This command checks your design and, if successful, starts the XM-VHDL simulator using the setup specified on the VHDL Setup - Simulation form. Alternatively, you can also simulate the design using the *Simulate Design* button on the fixed menu.

For more information on the XM-VHDL simulator, refer to the *XM-VHDL Simulator Help.*

Debugging Your VHDL Design

You can use several tools to help in debugging your design. You can

- Use the VHDL text editor or the schematic editor to edit your design
- Set breakpoints or traces on certain objects to prompt you to specify an object

Editing from the XM-VHDL Simulator

You can edit your design while debugging it. To edit a design unit from the XM-VHDL simulator:

1. Choose *File – Edit File* from the pop-up menu.
   The Edit File form will be invoked.
2. Specify the name of the design unit and click the *OK* button.
   The design will appear in the VHDL text editor.
3. Edit your design.

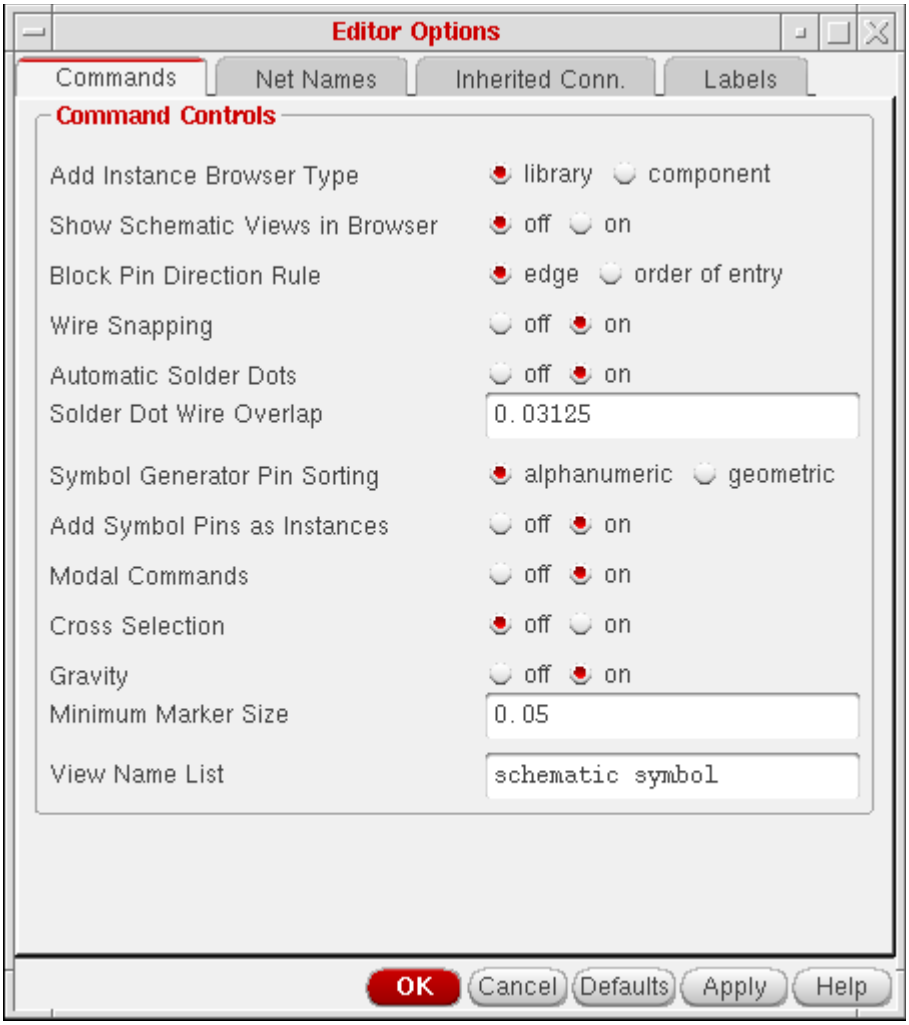
Using Cross Selection

When debugging your design, you can set breakpoints or traces on certain objects. These commands prompt you to specify an object. You can specify an object by doing the following:

- Typing in a name on the command form
- Clicking on an object in the schematic window that automatically fills in the name on the command form

To debug your design using cross selection:

1. From the Visrtuoso Schematic Editor, choose *Options – Editor*.
   The Editor Options form appears.



2. Select *Cross Selection* check box.

3. Click *OK.*

   Note: You can also add the following line to your `.cdsenv` file:

   ```
   envSetVal("schematic" "broadcast" 'boolean t)
   ```

4. From the XM-VHDL simulator, choose *Show – Breakpoint...– Set...* or *Show – Probes...– Set...* .

The following sections show you how to fill in the forms automatically through cross selection.

Setting Break Points

The *Set Breakpoint* command opens a form to create, delete, enable, or disable break points. A breakpoint listing includes the state of the breakpoint.

You can set break points on the form by identifying an object in the context of a schematic or symbol, as opposed to the netlisted VHDL text.

To use cross selection with the *Set Breakpoint* command in the XM-VHDL simulator, do the following:

1. Choose *Show – Breakpoints*... . The *Debug Settings* form appears.
2. Select *Set...* . The *Set Break* form appears.
   The Show form appears.

   Click *Set* to open the Set Break form to add a new breakpoint.
   The Set Break form appears.
   You can use this form to set breakpoints and specify options for breakpoints of various types. The options change, based on the type of breakpoint you set.
3. Click a wire in the Virtuoso Schematic Editor L.
   The Set Break form fills in automatically, showing the breakpoint type name.
4. Click *OK* on the Set Break form.

For details on this command, refer to the *XM-VHDL Simulator Reference Manual.*

Note: The *Show – Set Trace* command is similar. You can select wires in the schematic window to set a trace on a signal.

Comparing Simulations

VHDL uses a tool, SimCompare, to compare two waveform databases and log the results of the simulation comparison. SimCompare is invoked in the background when you select the

Simulation Compare option from the *Fixed Menu* or choose *Commands – Sim Compare*. SimCompare provides a text view of any differences found when two simulation results are compared. You can compare, for example, pre-synthesis and post-synthesis simulation results. For more information on using the SimCompare tool, refer to the SimCompare User Guide.

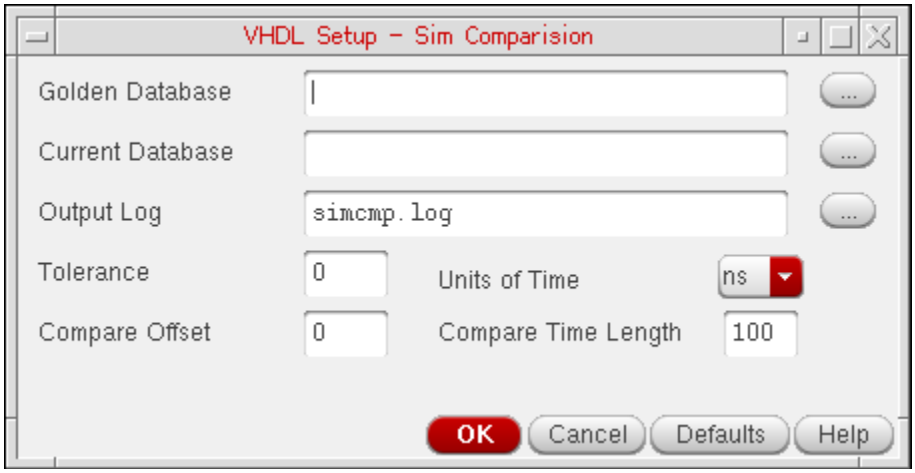
Choose *Setup – Sim Comparison* to open the VHDL Setup - Sim Comparison form in which you can specify names of the Simulation History Manager (SHM) databases that you need to compare and the name of a output log.

During simulation, the simulation database opens and traces signals. The current database becomes the *golden* database. Then, you run another simulation and compare it against the current database. You can compare the results of any two VHDL simulations that are able to generate SHM databases.

VHDL Setup - Sim Comparison Form

To set up the options for comparing two VHDL simulations:

1. Choose *Setup – Sim Comparison* .
   The VHDL Setup - Sim Comparison form appears.

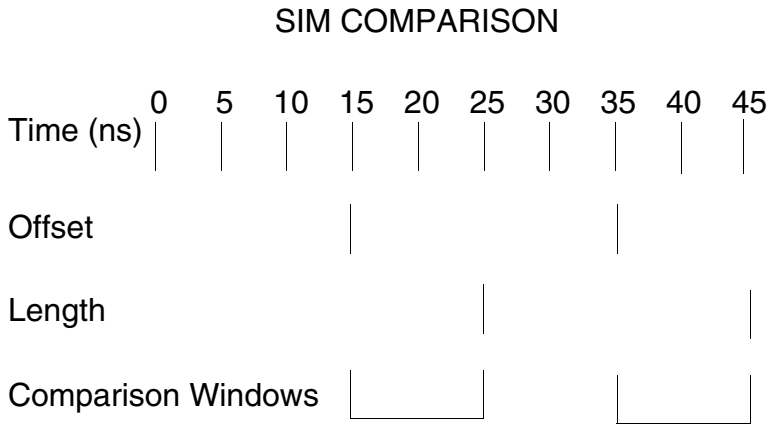


The various options on the VHDL Setup - Sim Comparison form are:

- Golden Database: Enables you to type the name of the first SHM database. Alternatively, you can also select a file by clicking the ellipsis button next to the field.
- Current Database: Enables you to type the name of the second SHM database used for comparison. Alternatively, you can also select a file by clicking the ellipsis button next to the field.

- Output Log: Enables you to specify the name of a log file that should contain the comparison results. Alternatively, you can also select a file by clicking the ellipsis button next to the field.
- Tolerance: Specifies the window of time during which a signal may match
- Compare Offset: Specifies the offset from beginning of the cycle when the comparison is relevant
- Units of Time:. Displays a list box from which you can select the unit of time
- Compare Time Length: Specifies the length of time from offset when comparison is valid

  The following example shows *Compare Offset* set to `15`, and *Compare Time Length* set to `10` nanoseconds.



Comparing VHDL Simulations

To compare simulations:

➡ Choose *Commands – Compare Simulation.*
The system displays the comparison results in a waveform window.

Simulating a VHDL Design Using Non-Cadence VHDL Tools

To use non-Cadence VHDL tools, you need to define your own SKILL procedures and register this information with the toolbox using the SKILL routine, `vhdlRegisterSimulator()`.

To register your callbacks, add the procedures for the callbacks in some file, say myfile.il that is in the /home/xyz directory and add the following lines to the .cdsinit file in your home directory:

```
(loadi "/home/xyz/myfile.il")
vhdlRegisterSimulator("myParserCallBack"
                      "myAnalyzerCallBack"
                      "myAnalyzerFileExt"
                      "myElaboratorCallBack"
                      "mySimulatorCallBack"
                      "myDataDirCallBack"
                      "myDataFileCallBack"
                      "myWorkLibCallBack")
```

If you do not provide your own callback routines to invoke any of the non-Cadence tools, namely, the `parser/analyser/elaborator/simulator`, then by default, the XM-VHDL tools such as the `parser/analyzer xmvhdl`, elaborator `xmelab,` and simulator `xmsim` are run.

Parser CallBack

```
procedure(myParserCallBack((t_filePath t_libName t_errorFile)
          ...
        )
return value:
disembodied Property List(DPL)
```

This procedure takes the VHDL source file and the name of the library in which this file is contained and runs the parser on it. The parser produces a LISP file which can be loaded by the SKILL engine and referenced as the `DPL` disembodied property list. If the parsing was unsuccessful, then errors/warnings from the parser are written to the error file.

Analyzer CallBack

```
procedure( myAnalyzerCallBack(t_filePath t_sourceFileName
          t_workLibraryName x_maxErrors)
       ....
       )
return value:
       t if successful
       nil if fatal error OR
       list( x_errCount t_errFileName ) if errors occurred
```

The procedure invokes the analyzer that analyzes the specified `sourceFileName` which exists in the specified directory `filePath`. The VHDL design unit is compiled into the library specified by the `workLibraryName` variable. The analysis aborts if the number of errors generated exceeds the specified `maxErrors`.

For incremental processing to avoid calling the analyzer if the file has already been analyzed, the analyzed file should exist or be linked to the specified `filePath`.

Analyzed File String

```
analyzerFileExt
```

It is a string representing the name of the analyzed file. For incremental netlisting you need to provide the name of the analyzed file so that the tool does not reanalyze the design.

Elaborator CallBack

```
procedure( myElaboratorCallBack(t_libName t_primaryName
         t_secondaryName t_simModelName x_maxErrors)
         ....
       )
return value:
      t if successful
      nil if fatal error OR
      list( x_errCount t_errFileName ) if errors occurred
```

This procedure invokes the elaborator to elaborate the VHDL design unit specified by *`libName`*, *`primaryName`* and *`secondaryName`* variables. The result of the elaboration is a simulation model whose name is specified by the *`simModelName`* variable. Elaboration aborts if the number of errors generated exceeds the specified `maxErrors`. The VHDL design unit can be an architecture or configuration. In the latter case, the configuration name is `configuration`.

Simulator CallBack

```
procedure( mySimulatorCallBack(t_libName t_primaryName
         t_secondaryName t_simModelName g_batchp
               t_scriptFileName)
         ...
       )
return value:
      t if successful
      nil if unsuccessful
```

The procedure invokes the simulator that simulates the specified simulation model *`simModelName.`* You can either run the simulation in batch or interactive mode using the *`batchp`* variable.

In case of interactive simulation you need to quit any currently running simulations and start the simulator. You can also set the WORK library to specified *`libName`*. If *`scriptFileName`* is not an empty string, then it is sent to the simulator as a setup file.

In case of batch simulation the simulator starts the simulation in background. If scriptFileName is not empty string, then it is sent as the simulation run. Otherwise a scriptFile is created which starts the simulation. In this case the simulation is ended based on the testbench.

Data Directory CallBack

```
procedure( myDataDirCallBack(t_libName t_cellName t_viewName)
        ...
       )
return value:
physical directory where the VHDL text file should be stored
```

If the library, cell, and view name are given this procedure returns the physical directory where the VHDL text file is to be stored.

Data File CallBack

```
procedure(myDataFileCallBack(t_libName t_cellName t_viewName)
        ...
       )
return value:
name of the file in which the VHDL text file should be stored.
```

If the library, cell and view names are given this procedure returns the physical file name under which the VHDL text file is to be stored.

Work Library CallBack

```
procedure(myWorkLibCallBack()
        ...
       )
return value:
name of the work library
```

This procedure returns the library that contains the compiled design unit information.

6

Modeling Schematics as VHDL

This chapter describes key aspects of how the VHDL netlister models Virtuoso schematics and symbols in VHDL. The netlister does this through modeling constructs directly to VHDL, using name mapping, renaming, aliasing, assigning, and converting, or by generating a separate VHDL package.

The following sections describe

Note: For information on the SKILL functions associated with VHDL Toolbox, see *Digital Design Netlisting and Simulation SKILL Reference*.

Mapping Case Sensitivity

Due to differences between the Virtuoso Schematic Editor L software, the Cadence® library structure, and VHDL in the way they handle mixed-case identifiers, it is important to understand how the VHDL netlister handles these differences. The following sections describe these differences, the rules used by the netlister to handle them, and the implications for how design data is created and organized.

Mapping Library, Cell, and Cellview Names to VHDL

VHDL design units are stored in Cadence libraries as cells and cellviews. For example, the `orgate` construct is stored in the gates library as

```
gates/orgate/entity
```

The behavioral architecture of the `orgate` structure is stored as

```
gates/orgate/behavioral
```

The names of libraries and design units in VHDL are case insensitive. This means that two referenced libraries named `Gates` and `gates` refer to the same library. In contrast, the names of libraries, cells, and cellviews in the Cadence library structure are case sensitive; so libraries defined as `Gates` and `gates` in the `cds.lib` file are considered as separate libraries.

To provide a consistent approach to mapping VHDL identifiers (which refer to library elements), Cadence uses the following rules for VHDL design units stored in a Cadence library:

- Any reference to a library element that is *not* escaped, regardless of the case of the identifier, refers to the lowercase equivalent of the identifier. For example, if an identifier refers to `OrGate`, then the equivalent library element is stored as `gates/orgate`.
- Any references to a library element that *is* escaped refers to the case-sensitive equivalent. For example, if an identifier refers to `\OrGate\`, then the equivalent library element is stored as `gates/OrGate`.

  You can avoid name mapping issues by always choosing names that use only lowercase letters and digits. For details on name mapping, refer to the *Cadence Application Infrastructure User Guide.*

When the netlister converts symbols or schematics to VHDL, the rules are applied to name-map all references to libraries, cells, and cellviews. The netlister maps the name of any reference to a library, cell, or cellview in the generated netlist, which contains uppercase characters by adding a backslash “\” to the beginning and end of the name. For example, a

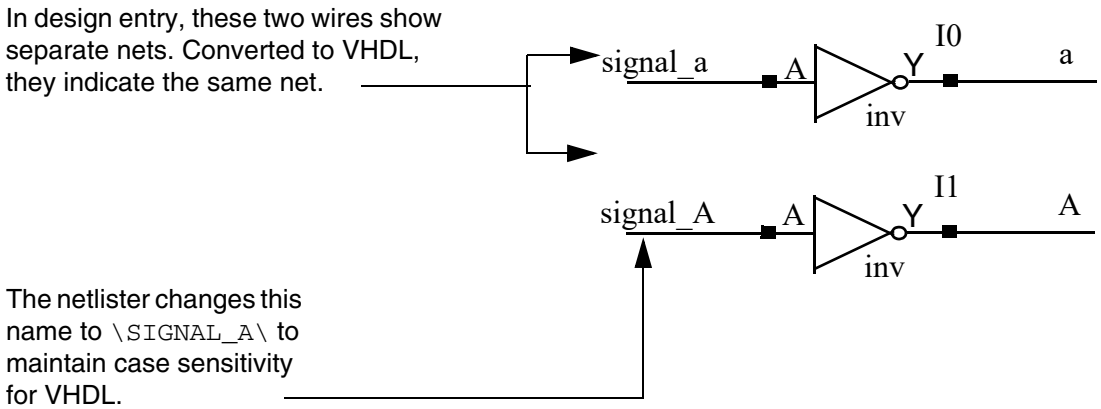
component declaration that refers to the `OrGate` cell in a Cadence library is netlisted as follows:

```
COMPONENT \OrGate\
PORT(
        in1 : IN std_ulogic;
        in2 : IN std_ulogic;
        out1: OUT std_ulogic
   );
END COMPONENT;
```

Avoid naming issues between VHDL and the Cadence library structure by avoiding uppercase characters when you name libraries, cells, and cellviews in a Cadence library.

Mapping Identifier Names to VHDL

The names of objects (identifiers) in VHDL are case insensitive. This means two signals named `signal_a` and `SIGNAL_A` are the same signal. Identifiers in Virtuoso Schematic Editor L are case sensitive; so Virtuoso Schematic Editor L considers wires `signal_a` and `SIGNAL_A` to be separate nets.



When converting the two schematic nets to VHDL, the netlister applies name mapping to one of the nets to model the identifiers correctly. The netlister uses the following rules to perform case-sensitive name mapping of pin, signal, alias, and instance names:

- The netlister maps the name of any identifier that contains uppercase characters by adding a backslash (\) to the front and back of the name. With the backslashes, the identifier becomes an escaped identifier in VHDL, and its case is preserved. For example, `SIGNAL_A` netlists as `\SIGNAL_A\`.

To disable case-sensitive name mapping of pin, signal, alias, and instance names, set the `simVhdlEscapeNameMapping` variable to `t` and select `Case Preserved` from the *Netlist Identifiers as* drop-down in the VHDL Setup - Netlist form.

To generate instance names in 5141 format, set the `hnlVHDL5141LIKEFORMATTING` flag to `t`.

The netlister also escapes any name that is an invalid VHDL identifier or is a VHDL keyword (reserved word).

Matching Compliant and Noncompliant Data

To allow loose case matching among identifiers when the design is a mix of compliant and noncompliant data, do the following:

- Set the *`CDS_ALT_NMP`* UNIX environment variable to a substring of `match` (case insensitive).

  Or, type the following command in the Command Interpreter Window (CIW):

  ```
  setShellEnvVar("CDS_ALT_NMP=match")
  ```

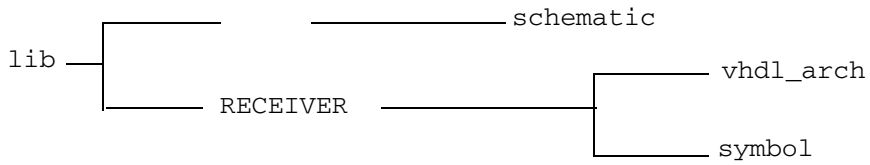
  From the standalone VHDL Netlister, type the following command in a UNIX shell window:

  ```
  setenv CDS_ALT_NMP match ( for C shell )
  ```

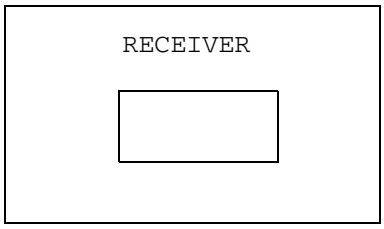
The system preserves identifiers while converting from a case-sensitive name space to a case-insensitive one, such as VHDL.

Note: Alternatively, you can use the `vhdlKeepCaseAsNC` variable to preserve the case of identifiers. For more information, see Appendix E, Customization Variables in *Spectre AMS Designer Environment User Guide*

Example




```
lib:top(schematic):
```




```
lib:RECEIVER(vhdl_arch) :
ARCHITECTURE vhdl_arch OF RECEIVER IS
BEGIN
END vhdl_arch;
```

When you netlist the design hierarchy shown above, the netlister escapes any uppercase identifier producing the following Virtuoso Schematic Editor L symbol `RECEIVER`:

```
ENTITY \RECEIVER\ IS
...
...
END \RECEIVER\;
```

However, in editing the VHDL architecture, if you use compliant data, but then used the following statement during the post netlisting analysis phase,

```
"ARCHITECTURE vhdl_arch OF RECEIVER IS
"
```

the statement would force the analyzer to believe that there is a VHDL entity called `receiver` since VHDL is case insensitive. However, the netlister, creates a VHDL entity named `RECEIVER`. This results in an analysis error. The name-mapping algorithm fails when the design hierarchy is a mixture of compliant and non-compliant data.

The alternative name-mapping algorithm allows a less strict name matching by retaining the case of an identifier when matching between a case-sensitive name space, such as OA, and a case-insensitive one, such as VHDL. However, the limitation in this approach is that all tools who support the VHDL name space need to do case insensitive library lookup.

When you map identifiers from a case-sensitive domain to the file system, the system converts the identifiers to lowercase. The netlisting results are as follows:

```
ENTITY RECEIVER IS
...
...
END RECEIVER;
```

If you netlist a cellview using the VHDL Netlister, and that cellview is on top of the hierarchy, the Netlister refers to the schematic for the port order property, and not to the symbol. Because no pins exist in the schematic, no port declarations exist in the above entity.

Assigning VHDL Data Types

The netlister assigns data types to ports and signals when generating entities and architectures. Data types are values of data that pass through a signal, such as 0 and 1 for bits or red and yellow for color.

The following list shows the rules, in order of priority, that the netlister uses for assigning a VHDL data type.

1. The object has a `vhdlDataType` property already assigned to it, which the netlister uses.
2. The netlister uses one of the objects connected to the data type, such as an instance pin.

   The netlister predefines the `vhdlDataType` property, which you can set on pins and wires by choosing *Edit – Properties – Objects* in the schematic editor. If you use the VHDL pins, then the Edit Object Properties form automatically prompts you for these data types.
3. The netlister uses the default scalar or vector data type specified on the cellview.

   You can specify the default scalar or vector data type by choosing *Edit – Properties – VHDL* in the schematic editor.
4. The netlister uses a default scalar or vector data type specified in the VHDL Toolbox. You can change the default scalar or vector VHDL data type in the *VHDL Set Up Check* dialog box. To display the *VHDL Set Up Check* dialog box choose *Setup – Check* in VHDL Toolbox. The default scalar data type is `std_ulogic` and the default vector data type is `std_ulogic_vector`.

In case of port bundles you can specify the VHDL datatype for individual signals as follows:

```
sigName1:sigType1, sigName2:sigType2, sigName3:sigType3
```

For example, you have defined two signals, sig1 and sig2 as:

```
vhdlDataType : sig1:real
```

In this case sig1 has `real` as the datatype and sig2 has the default scalar datatype.

```
vhdlDataType : real
```

In this case both the signals, sig1 and sig2, have `real` as the datatype.

If you do not specify the `vhdlDataType` then both the signals have the default scalar datatype defined in the *VHDL Set Up Check* dialog box.

Modeling Schematic Pins as VHDL Ports

The netlister models schematic pins as VHDL ports by converting the direction of the pin as follows:

Schematic pins	VHDL ports
input	in
output	out
inputOutput	inout

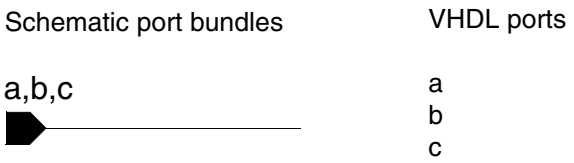
If the schematic pin has the property

```
vhdlPortType = "buffer"
```

the netlister assigns the direction of the port as `buffer`.

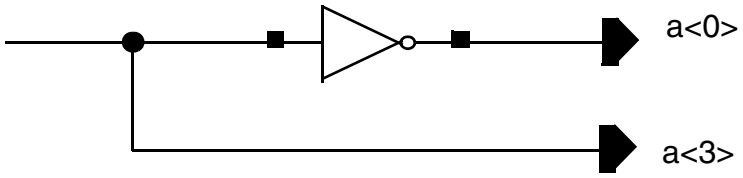
Supporting Port Bundles

VHDL does not support bundles, so the netlister models port bundles as multiple VHDL ports, one for each bundle element. For example,



Discontinuous Ports

VHDL does not support sparse or discontinuous port ranges. That is, if a port is specified with a range, then all members of the range must be declared. Following is a simple example of a schematic that demonstrates a discontinuous port range.



Here the schematic contains two out ports, `a<0>` and `a<3>`, which are discontinuous. The netlister name maps these port names by appending the index number and `_split_%d` to the end of the name.

Following is the entity generated from the previous example, where the `a<0>` and `a<3>` ports are name-mapped to `a_split_0` and `a_split_3`, respectively:

```
ENTITY split IS
   PORT(
      a_split_0 : OUT std_ulogic;
      a_split_3 : OUT std_ulogic
   );
END split;
```

Modeling Schematic Nets as VHDL Signals

The netlister models schematic internal or local nets as VHDL signals. VHDL does not support signal bundles. The netlister models signal bundles as multiple VHDL signals, one for each bundle element, as in the previous VHDL port example.

Supporting Global Signals

The netlister supports global signals by

- Generating a separate VHDL package to declare global signals called `cds_global_signals`
- Gathering global signals while traversing all cellviews in the hierarchy

- Inserting a reference to the `cds_global_signals` packages in the context clause of each architecture that refers to a global signal (the `cds_global_signals` package is part of the top cellview library).
- Removing "`!`" from the end of each global signal name, such as `vdd!`, and appending `_cds_global` to the end of the signal name, such as `vdd_cds_global`.

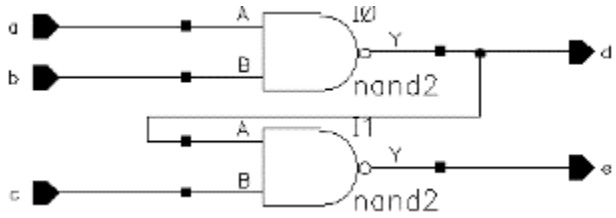
Aliasing Ports and Signals

VHDL supports aliases for ports and local signals in the following ways:

- The netlister declares aliases for all wire names that are marked as an alias using `Add - Wire Name` or patchcords in the Virtuoso Schematic Editor L.
- The netlister declares an alias for a net name that is connected to a pin with a different name.

Modeling Feedback Signals

If a signal is driven (connected to an input pin or output instance pin) and read (connected to an output pin or input instance pin), then the signal is in a feedback loop. The netlister models feedback loops by inserting a temporary signal and connecting it between instances and ports. The following example shows a schematic with a temporary signal that connects instance `I0`, instance `I1`, and pin `d` and its resulting netlist.

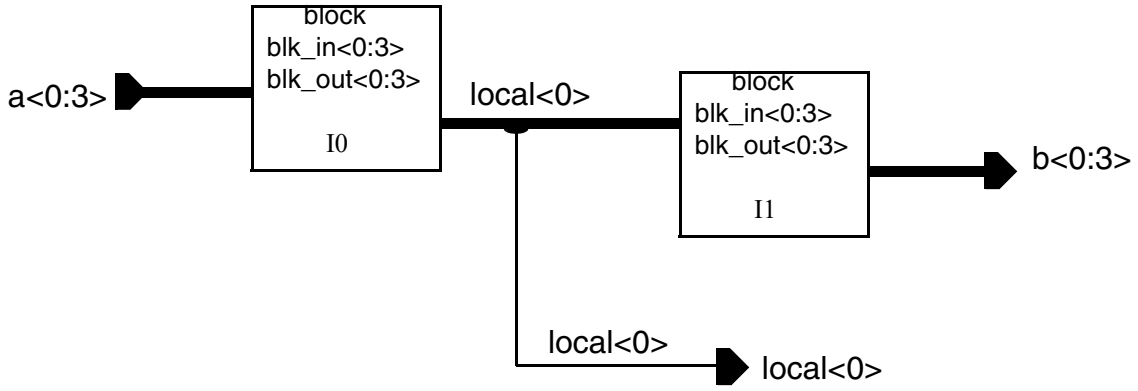



```
ARCHITECTURE schematic OF feedback IS
    COMPONENT nand2
        PORT(
            Y : OUT std_ulogic;
            B : IN std_ulogic;
            A : IN std_ulogic
        );
    END COMPONENT;
    SIGNAL d_tempsig : std_ulogic;
BEGIN
```

```
    d <= d_tempsig;
    I1 : nand2
        PORT MAP(
            Y => e,
            B => c,
            A => d_tempsig
        );
    I0 : nand2
        PORT MAP(
            Y => d_tempsig,
            B => b,
            A => a
        );
END schematic;
```

Signal/Port Name Collisions

VHDL does not allow a port to be named `a<0>` and a signal to be named `a<1>`. Either `a<0:1>` is declared as a port or a signal, but not both. Following is a simple example that demonstrates how a schematic that violates this rule can be drawn:



Here, the schematic contains a signal named `local<0:3>` and a port named `local<0>`. In VHDL, you cannot have a port and a signal with the same name, which in the example is `local`. Normally, the netlister can make the signal an alias of the port, but in this case the width of the signal is greater than the width of the port.

Following is the netlist that is generated from the previous example, where the local signal is name-mapped to be `local_portclash:`

```
ENTITY port_clash IS
   PORT(
      local : OUT std_ulogic_vector(0 DOWNTO 0);
      b : OUT std_ulogic_vector(0 TO 3);
      a : IN std_ulogic_vector(0 TO 3)
   );
END port_clash;

ARCHITECTURE schematic OF port_clash IS
   COMPONENT blk
      PORT(
         blk_out : OUT std_ulogic_vector(0 TO 3);
         blk_in : IN std_ulogic_vector(0 TO 3)
         );
   END COMPONENT;

   SIGNAL local_portclash : std_ulogic_vector(3 DOWNTO 0);
BEGIN
   local(0) <= local_portclash(0);
   I0 : blk
      PORT MAP(
         blk_in(0 TO 3) => a(0 TO 3),
         blk_out(0) => local_portclash(0),
         blk_out(1) => local_portclash(1),
         blk_out(2) => local_portclash(2),
               blk_out(3) => local_portclash(3)
            );
   I1 : blk
      PORT MAP(
         blk_in(0) => local_portclash(0),
         blk_in(1) => local_portclash(1),
            blk_in(2) => local_portclash(2),
         blk_in(3) => local_portclash(3),
            blk_out(0 TO 3) => b(0 TO 3)
      );
END schematic;
```

Modeling Schematic Instances as VHDL Instances

The netlister models schematic instances as VHDL component instances.

Ignoring Instances

The netlister skips over those instances that have the `nlAction` property with the value of `ignore` on either the instance or the instance master. If you set the `nlAction` property, then the instance is ignored by all Cadence netlisters. Examples of instances that are ignored are pins, sheet borders, and power and ground symbols.

```
"nlAction" = "ignore"
```

If you want to prevent only the VHDL netlister from netlisting an instance, then you can set the string property, `nlIgnore` to `vhdl` on the design object as shown below:

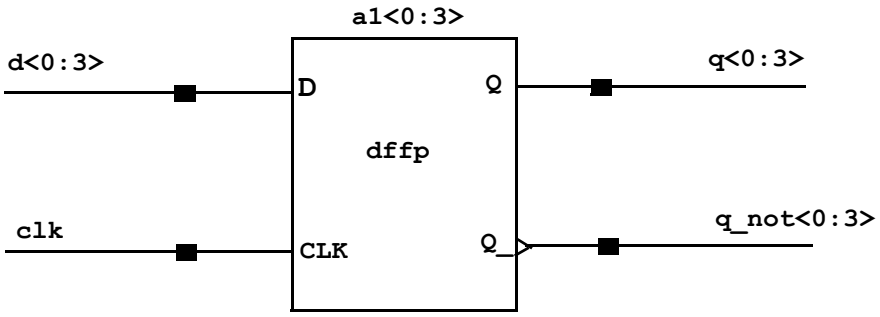
```
nlIgnore = "vhdl"
```

If a design object has `nlAction` set to `ignore` then irrespective of the value of `nlIgnore`, the design object will not be netlisted. This is because, `nlAction` has precedence over `nlIgnore`.

Note: Although the `nlIgnore` property accepts space-separated values, each value being the netlister name that should ignore the design object, currently only the VHDL netlister implements the `nlIgnore` property.

Modeling Iterated Instances

The netlister models iterated instances by splitting the iterated instances into individual instances. It names the individual instance using a base name appended with an iteration index. The following example shows a schematic containing an instance of a D flip-flop iterated from 0 to 3 and its resulting netlist.




```
ARCHITECTURE schematic OF iterated IS
    COMPONENT dffp
        PORT(
            Q : OUT std_ulogic;
            Q_ : OUT std_ulogic;
```

```
            CLK : IN std_ulogic;
            D : IN std_ulogic
        );
    END COMPONENT;

    SIGNAL q : std_ulogic_vector(0 TO 3);
    SIGNAL q_not : std_ulogic_vector(0 TO 3);
    SIGNAL clk : std_ulogic;
    SIGNAL d : std_ulogic_vector(0 TO 3);
BEGIN
    a1_0 : dffp
        PORT MAP(
            Q => q(0),
            CLK => clk,
            Q_ => q_not(0),
            D => d(0)
        );
    a1_1 : dffp
        PORT MAP(
            Q => q(1),
            CLK => clk,
            Q_ => q_not(1),
            D => d(1)
        );
    a1_2 : dffp
        PORT MAP(
            Q => q(2),
            CLK => clk,
            Q_ => q_not(2),
            D => d(2)
        );
    a1_3 : dffp
        PORT MAP(
            Q => q(3),
            CLK => clk,
            Q_ => q_not(3),
            D => d(3)
        );
END schematic;
```

Modeling Instance Properties as Generics

Generics in VHDL let an instance pass information to its component. The netlister provides the ability to define generics of a component and to map the values of schematic instance properties to generics. The netlister accomplishes this through the use of the `vhdlGenericDefList` property.

The `vhdlGenericDefList` property defines a list of properties that represent the generics and their types for a component.

You can store the property in the cellview defining the interface for the component from Virtuoso Schematic Editor or Virtuoso Symbol Editor.

To store `vhdlGenericDefList` in a cell through Virtuoso Schematic Editor or Virtuoso Symbol Editor:

1. Open the cell in the editor and select *Edit – Properties – VHDL*.

   The VHDL Properties form appears.

2. Click the *Add* button.

   The Add Generic form appears.

3. Specify the following information in their respective fields:

   - The name of the generic, for example, `MIN_DELAY`.
   - The valid VHDL data type, for example, `INTEGER`.
   - The value, for example, `10`.

4. Click *OK*.

   The added property appears in the generics section of VHDL Properties form.

5. Choose *File — Check and Save* in the editor.

6. Use VHDL Toolbox to regenerate the netlist of the design.

Note: By default, VHDL Toolbox prints all generics. To print only those generics for which the default value is specified in the VHDL netlist, set `hnlVHDLSkipGenericWithNoDefaultValue` to `t` in Virtuoso CIW, `.simrc`, or `si.env`. You can set the default values of generics from the VHDL Setup - Netlist form. For details, see the *Generic Defaults* tab.

You can also store the `vhdlGenericDefList` property in the cell or library-level CDF of the component.

To store `vhdlGenericDefList` as a CDF parameter of a component:

1. Choose *Tools – CDF – Edit* from Virtuoso CIW.

   The Edit CDF form appears.

2. Choose where you want to edit the CDF property, which can be at the *Cell* or *Library* level.
3. Choose *Base* in the *CDF Layer* area.
4. Specify the library and cell of the component in their respective fields.
5. Add the component parameter in the component parameters table. Use the following guidelines:
   - Specify the parameter *Name* as `vhdlGenericDefList`.
   - Select the parameter *ype* as `string`.
   - Specify the *Default Value* of the parameter as shown in the following example:

     ```
     ((min_delay time 5 ns) (max_delay time 10 ns))
     ```

     The value specified in this example is an `iLList` type list.
6. Click *Apply*, and then *OK* to save the CDF parameter and exit the Edit CDF form.
7. In the design, replace the entity so that it has the CDF parameter.

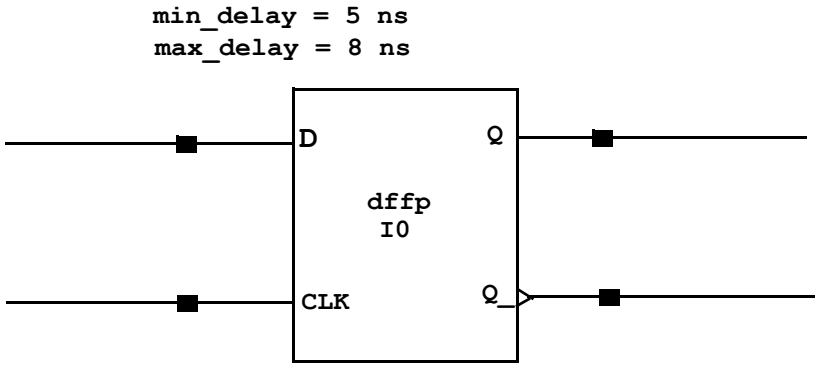
   The VHDL generics become available in the newly created entity. You can then use VHDL Toolbox to regenerate the netlist.

VHDL Toolbox uses the `vhdlGenericDefList` property when generating entities from schematics or symbols, and for generating component declarations when generating architectures. The property is stored as a SKILL list of lists, with each sublist containing the generic name, type, and an optional value. For example:

```
vhdlGenericDefList = (("min_delay" "time" "5 ns")
                      ("max_delay" "time"))
```

The following schematic and resulting netlist show an instance of a D flip-flop with the two properties `min_delay` and `max_delay` representing generics. The D flip-flop symbol

contains the `vhdlGenericDefList` property, which defines these properties as generics with the type of `time`.




```
ARCHITECTURE schematic OF generic_example IS
    COMPONENT dffp
        GENERIC(
            max_delay : time;
            min_delay : time
        );
        PORT(
            Q : OUT std_ulogic;
            Q_ : OUT std_ulogic;
            CLK : IN std_ulogic;
            D : IN std_ulogic
        );
    END COMPONENT;
BEGIN
I0 : dffp
        GENERIC MAP(
            max_delay => 8 ns,
            min_delay => 5 ns
        )
        PORT MAP(
            Q => q,
            CLK => clk,
            Q_ => q_not,
            D => d
        );
END schematic;
```

It is possible that the same property is set in the VHDL file and a CDF parameter of a cell. In such a case, VHDL Toolbox determines the property to print in the `GENERIC` section of the netlist using the SKILL variable `hnlVHDLoverCDFGenerics`. By default, this variable is set to `nil` and VHDL Toolbox prints the CDF parameter. To configure VHDL Toolbox to print the VHDL property, instead of the conflicting CDF parameter, set `hnlVHDLoverCDFGenerics` to `t` in Virtuoso CIW, `.simrc`, or `si.env`.

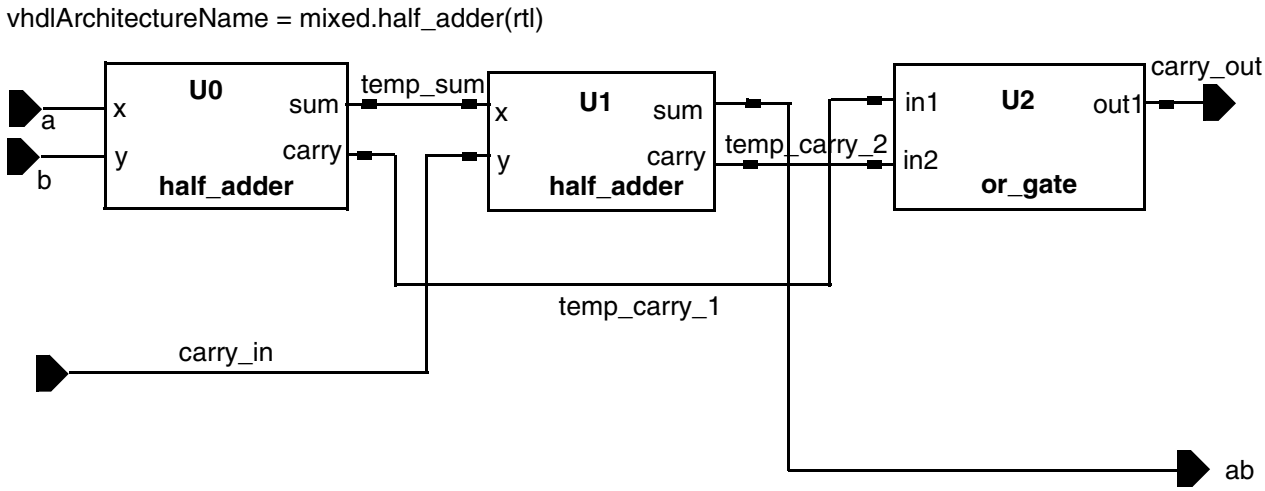
When you create entity view for a cell using cv2cv, the properties added as `Generics` through `vhdl` properties are printed in the `entity` generics. But, if a property with same name is added on the cellview through cellview properties, then the value corresponding to this property gets printed. If you do not wish to override this generic value, set the `vhdlDoNotUseCVPropForGenerics` flag to `t`.

Specifying Explicit Component Binding

Normally, the netlister determines a component's binding while traversing a design hierarchy. However, it can be useful to explicitly specify the binding for a component instance. The netlist provides the following property for this purpose

The `vhdlArchitectureName` property can be placed on an instance in a schematic to explicitly specify the component binding for the instance. The property is stored as a string and specifies the library, entity, and architecture names of the component to use when generating a configuration specification for the instance. An example is `mixed.half_adder(behavior)`.

The following schematic contains two instances of the `half_adder` component, each containing a different value for `vhdlArchitectureName`. The resulting netlist follows the schematic.




```
ARCHITECTURE schematic OF full_adder IS
    COMPONENT or_gate
        PORT(
            in1 : IN std_ulogic;
            in2 : IN std_ulogic;
            out1 : OUT std_ulogic
        );
    END COMPONENT;

    COMPONENT half_adder
        PORT(
            x : IN std_ulogic;
            y : IN std_ulogic;
            sum : OUT std_ulogic;
            carry : OUT std_ulogic
        );
    END COMPONENT;

    SIGNAL temp_carry_2 : std_ulogic;
    SIGNAL temp_carry_1 : std_ulogic;
    SIGNAL temp_sum : std_ulogic;

    FOR U0 : half_adder
```

```
        USE ENTITY mixed.half_adder(rtl);
    FOR U1 : half_adder
        USE ENTITY mixed.half_adder(behavior);
BEGIN
    U2 : or_gate
        PORT MAP(
            in2 => temp_carry_2,
            out1 => carry_out,
            in1 => temp_carry_1
        );
    U0 : half_adder
        PORT MAP(
            x => a,
            y => b,
            sum => temp_sum,
            carry => temp_carry_1
        );
    U1 : half_adder
        PORT MAP(
            x => temp_sum,
            y => carry_in,
            sum => ab,
            carry => temp_carry_2
        );
END schematic;
```

Specifying Components Declared in External Packages

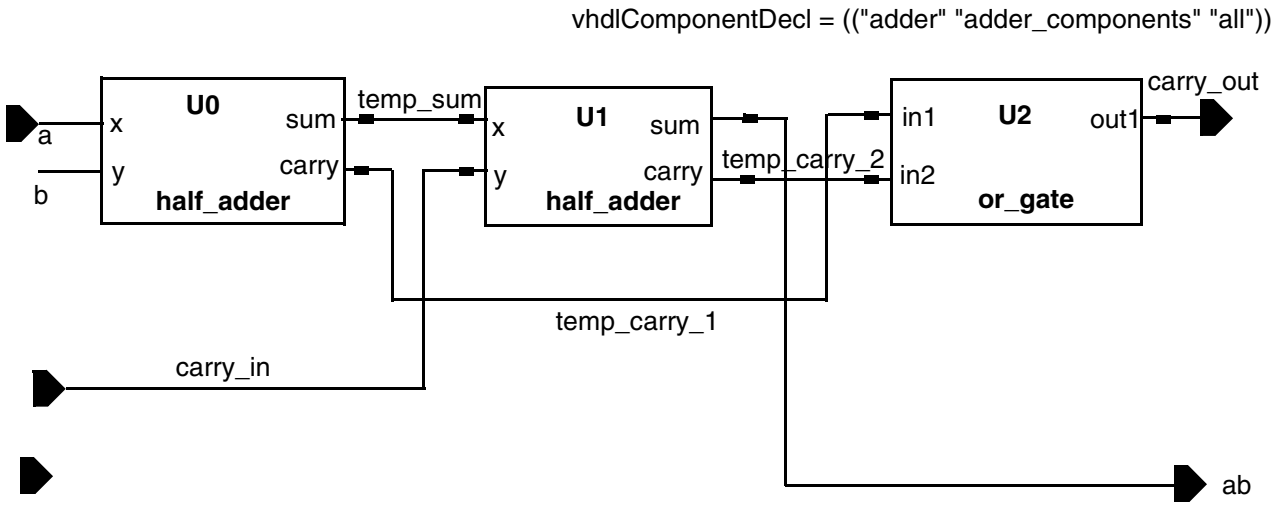
The netlister generates component declarations for all components that are instantiated in the schematic being netlisted. However, it is common to specify component declarations in a package that can be referenced by architectures that instantiate these components. The netlister provides the following properties for specifying components declared in an external package.

- You can place the `vhdlComponentDecl` property in a symbol to specify the name of the external package containing its component declaration. When generating a netlist for a schematic that instantiates the symbol, the netlister does not generate a component declaration.

  However, the component declaration won't be skipped when you place the `vhdlComponentDecl` property on a symbol (for example, `library:cell::L1:C1`)

and specify an external `vhdl` text file containing the definition for a cell with the same name in a different library (for example, `library:cell::L2:C1`).

The following schematic demonstrates the use of this property:



The schematic contains two instances of `half_adder` and one of `or_gate`. The schematic also contains the property

specifying that the component declaration for the `half_adder` component exists in an external package. The `or_gate` symbol contains the property

```
vhdlComponentDecl=(("adder" "adder_components" "all"))
```

specifying that its component declaration exists in the `adder_components` package in the `adder` library. Following is the resulting netlist:

```
LIBRARY ieee, adder;
USE ieee.std_logic_1164.all;
USE adder.adder_components.all;

ARCHITECTURE schematic OF full_adder IS
    SIGNAL temp_carry_2 : std_ulogic;
    SIGNAL temp_carry_1 : std_ulogic;
    SIGNAL temp_sum : std_ulogic;
BEGIN
    U2 : or_gate
        PORT MAP(
            in2 => temp_carry_2,
            out1 => carry_out,
```

```
            in1 => temp_carry_1
        );
    U0 : half_adder
        PORT MAP(
            x => a,
            y => b,
            sum => temp_sum,
            carry => temp_carry_1
        );
    U1 : half_adder
        PORT MAP(
            x => temp_sum,
            y => carry_in,
            sum => ab,
            carry => temp_carry_2
        );
END schematic;
```

Modeling Multisheet Schematics

The netlister models a multisheet schematic as a single design unit (architecture) by

- Generating a single entity for the schematic index to declare hierarchical pins
- Defining each sheet in the architecture as a separate VHDL block
- Matching the name of the block to the instance name of the sheet
- Maintaining separate name scopes within a block that directly map to the use model of separate cellviews for each sheet

7

Customizing Your Environment

This appendix covers the following topics:

- Setting VHDL HNL Variables on page 109
- Setting VHDL CDSENV Variables on page 112
- Setting xrun Variables on page 114

For information on the SKILL functions associated with VHDL Toolbox, see *Digital Design Netlisting and Simulation SKILL Reference*.

Setting VHDL HNL Variables

The following table shows VHDL HNL variables, which VHDL netlister uses while netlisting a design. You can also set these variables in the `.simrc` file.

Variable Name	Type	Value	Description
`hnlInhConnPrefix`	`string`	`""`	Support for Inherited Connection
`hnlUserSimViewName`	`string`	`""`	Netlist CDF parameters for
`hnlUserStopCVList`	`list`	`("sample" "basic" "ieee")`	Stop Library List
`hnlVHDLBusRangeNotation`	`string`	`"Ascending"/ "Descending"/ "Auto"`	Bus Range Notation
`hnlVHDLMergeSignals`	`boolean`	`t/nil`	Bus Range Notation
`hnlVHDLCheckSameStopCellFromMultLibs`	`boolean`	`t/nil`	Ext. Sources
`hnlVHDLConfigSpecForCommonMaster`	`string`	`"OTHERS"/"ALL"`	Instance Bindings
`hnlVHDLDefaultClauseList`	`list`	`(("IEEE" "std_logic_1164" "all") ("STD"))`	VHDL Default Context Clause

Variable Name	Type	Value	Description
hnlVHDLDefaultDataType	string	"bit"	Default Data Type
hnlVHDLDefaultInhPortDataType	string	"bit"	Support for Inherited Connections
hnlVHDLDefaultInhPortMode	string	"IN"/"OUT"/"INOUT"	Support for Inherited Connections
hnlVHDLDefFileExts	list	("vhd" "vhdl")	Ext. Sources
hnlVHDLDonotUseCdsNmp	boolean	t/nil	Netlist Identifiers as
hnlVHDLEnforceVHDLViewList	list	("symbol_inh" "symbol_xxx")	VHDL Enforce Binding for Views
hnlVHDLGenTestBench	string	t/nil	Generate Test Bench Template
hnlVHDLIEEESyntax	string	"1076-1993"	VHDL IEEE Syntax
hnlVHDLInstanceBinding	string	"Direct"/"Configuration"	VHDL Instance Binding
hnlVHDLLibraryBinding	string	"Merge"/"Preserve"	VHDL Library Binding
hnlVHDLConfigUseLibNameForStoppingCell	boolean	t/ni	VHDL Library Binding
hnlVHDLMaxErrors	int	10	Max Errors
hnlVHDLMaxExtFileErrCount	int	1	Ext. Sources
hnlVHDLParseExtDataSeverity	string	"error"	Ext. Sources
hnlVHDLPrintPortInitialValue	boolean	t/nil	Print Default Value of Ports
hnlVHDLPriorityType	list	(?aType? ?bType?)	Type Conversion Functions
hnlVHDLResetExternalVHDLData	boolean	t/nil	Ext. Sources
hnlVHDLSingleNetlist	boolean	t/nil	Single Netlist File
hnlVHDLSkipGenericWithNoDefaultValue	boolean	t/nil	Modeling Instance Properties as Generics
hnlVHDLSplitConfigEnabled	boolean	t/nil	Generating the Configuration of All Cellviews
hnlVHDLSplitInstsInConfig	string	""	Instance Bindings
hnlVHDLTestBenchInstName	string	"dut"	SKILL Variables for Test Bench
hnlVHDLTypeConflictSeverity	string	"error" | "warning"	Type Conflict Resolution

Variable Name	Type	Value	Description
hnlVHDLTypeConversionFuncs	list	("typeA" "typeB" "fn_typeA2typeB" "fn_typeB2typeA"	Type Conversion Functions
hnlVHDLTypeResolverDefs	list	"typeA" typeB" "typeC"	Type Conflict Resolution
hnlVHDLUserComment	string	""	VHDL NetList Header
hnlVHDLUserDataTypeList	list	(("std_logic" "std_logic_vector" "LOGIC" "'0'") ("std_ulogic" "std_ulogic_vector" "LOGIC" "'0'") ("bit" "bit_vector" "LOGIC" "'0'"))	Data Types
hnlVHDLVerboseMode	boolean	t/nil	Detailed Report
simPrintInhConnAttributes	boolean	t/nil	Evaluate Inherited Connections
simReNetlistAll	boolean	t/nil	Netlisting
simResolveStopCellImplicitConns	boolean	t/nil	Support for Implicit Inherited Connections for Leaf Cells
simStopList	list	("symbol" "behavioral")	Stop View List
simViewList	list	("stimulus" "schematic" "structural" "dataflow" "behavioral" "symbol")	Switch View List
vhdlSimTestBenchFile	string	"test_bench.vhd"	SKILL Variables for Test Bench
vhdlSimTestBenchLCV	list	("library-name" "cell-name" "view-name")	SKILL Variables for Test Bench

Note: In the `.simrc` file, you can also specify pre and post-processing functions `hnlVHDLPreNetlistFunc()` and `hnlVHDLPostNetlistFunc()`. For details, see "Customizing Pre- and Post-Processing Functions" on page 34.

Setting VHDL CDSENV Variables

The following table shows VHDL variables, which VHDL netlister uses while netlisting a design. You can set these variables in the `.cdsenv` file.

Variable Name	Type	Default Value	Possible Values	Description
analyzeMode	boolean	nil		Analyze(Compile) Design Units
busRangeNotation	string	"Auto"	"Ascending" "Descending" "Auto"	Bus Range Notation
caseSensitivity	string	"Converted using CDS NMP"	"Case Preserved" "Converted using CDS NMP"	Netlist Identifiers as
configName	string	"config_vhdl"		VHDL Configuration Name
configSuffix	string	"cfg"		VHDL Configuration File Suffix
defComment	string	"Netlist: Library=<libraryName>, Cell=<cellName>,View=<viewName> Time:<dateStamp>By:<user>"		Default Text
defaultDataType	string	"bit"	"bit" "std_logic" "std_ulogic" "real"	Default Data Type (for non-VHDL Views)
detailReport	boolean	nil		Detailed Report
elaborateMode	boolean	nil		Elaborate Design
enforceVHDLBindingView	string	"symbol_inh"		Enforce VHDL Binding for View(s))
evalInheritedConn	boolean	t		Evaluate Inherited Connections
generateTestBench	boolean	nil		Generate Test Bench Template
instanceBinding	string	"Direct"	"Direct" "Configuration"	VHDL Instance Binding
libraryBinding	string	"Merge"	"Preserve" "Merge"	VHDL Library Binding
maxErrors	int	10		Max Errors

Variable Name	Type	Default Value	Possible Values	Description
netMode	cyclic	"ReNetlist All"	"Incremental" "ReNetlist All"	Netlisting Mode
simViewName	string	"<None>"		Netlist CDF parameters for
singleNetlist	boolean	t		Single Netlist File
skipLibList	string	"analogLib"		Skip Design Units
splitConfiguration	boolean	nil		Generating the Configuration of All Cellviews
stopLibList	string	"basic sample ieee"		Stop Library List
topLibraryName	string	""		Top Level Design
topCellName	string	""		Top Level Design
topViewName	string	""		Top Level Design
runDir	string	""		Run Directory
stopViewList	string	"symbol behavior behavioral"		Stop View List
switchViewList	string	"stimulus schematic structure structural dataflow behavior behavioral symbol"		Switch View List
vhdlIEEESyntax	string	"1076-1993"	"1076-1993" "1076-1987"	VHDL IEEE Syntax
defGenericsList	string	"'((\"REAL\" \"1.0\") (\"INTEGER\" \"1\") (\"STRING\" \"\") (\"TIME\" \"1 ns\") (\"BOOLEAN\" \"true\"))"		Generic Defaults

Variable Name	Type	Default Value	Possible Values	Description
defDataTypeMappingList	string	"'((\"std_logic\" \"std_logic_vector\" \"LOGIC\" \"'0'\") (\"std_ulogic\" \"std_ulogic_vector\" \"LOGIC\" \"'0'\") (\"bit\" \"bit_vector\" \"LOGIC\" \"'0'\"))"		Data Types
defClauseList	string	"'((\"IEEE\" \"std_logic_1164\" \"all\") (\"STD\"))"		VHDL Default Context Clause

Note: The `defClauseList`, `defDataTypeMappingList`, `defGenericsList`, and `skipLibList` CDSENV variables also accept values without a backslash and double quote \". For example, both the settings below are valid:

```
envSetVal("vhdl.oss" "skipLibList"  'string  " '(( \"analogLib\" )) ")
envSetVal("vhdl.oss" "skipLibList"  'string  "'(( analogLib))")
```

Setting xrun Variables

The following table shows the SKILL variables that are used for the *xrun utility*. You can also set these variables in the `.simrc` file.

Note: If you already have these variables set in the `.vhdlrc` file in the run directory, you do not need to add these variables. The settings from `.vhdlrc` are used.

Variable Name	Type	Value	Description
vhdlSimSimulator	string	"xrun"	Specifies the simulator to be used by VHDL Toolbox. You can set it to `xrun` to use the xrun utility or to `xmsim` to use the xmsim simulator.

Variable Name	Type	Value	Description
vhdlSimVhdlFileExt	string	".vhd,.vhdp,.vhdl,.vhdlp,.VHD,.VHDP,.VHDL,.VHDLP"	Include Extensions
vhdlSimVerilogFileExt	string	".v,.V,.vp,.VP,.sv,.SV"	Include Extensions
vhdlSimLogFile	string	"file-name"	Log File
vhdlSimOptions	string	"-ieee364"	Simulation Options
vhdlSimDebugObjectAccess	string	"r"/ "rw"/ "rwc"	Debug Options
vhdlSimEnableLineDebug	boolean	t/nil	Debug Options
vhdlSimSnapShot	string	nil	Specifies name of the simulator snapshot.
vhdlSimNetlistandSimulate	boolean	t/nil	Specifies that the design should be netlisted before simulation.
vhdlSimCleanCompiledData	boolean	t/nil	Cleans the compiled data.
vhdlSimMode	string	"interactive"/ "batch"	Specifies whether the simulation is run in interactive or batch mode.
vhdlSimScriptFile	string	"file-name"	Specifies name of the script file to be run by the simulator.
vhdlSimIncludeTestBench	boolean	t/nil	Specifies that the testbench specified in the simulation is to be included.

8

VHDL Netlister Properties

For information on the SKILL functions associated with VHDL Toolbox, see *Digital Design Netlisting and Simulation SKILL Reference*.

Following are the properties used by the VHDL netlister:

vhdlDataType specifies the VHDL data type for the netlister to use when generating a port in a VHDL generic from a pin.

Type	string
Value	<dataType>
Example	“integer”

vhdlPortType specifies the VHDL port mode for the netlister to use when generating a port in a VHDL generic from a pin. The netlister searches for the port mode value in the vhdlPortType, vhdlPortMode, and dbDirection properties, in this order.

Type	string
Value	“in”	“out”	“inout”	“buffer”
Example	“in”

vhdlInitialValue specifies the initial value for the netlister to assign when generating a port in a VHDL generic from a pin.

Type	string
Value	<expression>
Example	“1”

Note: During CV2CV the lookup for VHDL properties `vhdlDataType`, `vhdlInitialValue`, `vhdlResolveFunction`, and `vhdlPortMode` happens on the port,

port's pins and pin fig. The property's value on the port have the ighest priority followed by that on port's pin and pin fig.

9

Running Simulations with Xcelium

You can also simulate and debug your designs using the Xcelium simulator. With this simulator in place, all the executable names, log file names, and output directory names have been changed. However, the old executable and log file names will continue to work for other simulators.

The following table lists the changes in the executable and log file names when using the Xcelium simulator:

Old Executable	Old log file name	New Executable	New log file name
irun	irun.log	xrun	xrun.log
iprof	iprof.log	xprof	xprof.log
ncsim	ncsim.log	xmsim	xmsim.log
ncelab	ncelab.log	xmelab	xmelab.log
ncvlog	ncvlog.log	xmvlog	xmvlog.log
ncvlog_cg		xmvlog_cg	
ncvhdl	ncvhdl.log	xmvhdl	xmvhdl.log
ncvhdl_cg		xmvhdl_cg	
ncsc	ncsc.log	xmsc	xmsc.log
ncsc_run	ncsc.log	xmsc_run	xmsc_run.log
ncls	ncls.log	xmls	xmls.log
nchelp	nchelp.log	xmhelp	xmhelp.log
ncdc	ncdc.log	xmdc	xmdc.log
ncverilog	ncverilog.log	xmverilog	xmverilog.log
ncprep	ncprep.log	xmprep	xmprep.log

Glossary

A

alias
Specifies an alternate name for a signal.

analysis
Compilation of the VHDL source code.

architecture
Describes the implementation of an entity. A single entity can have several different architectures.

.ast, AST
Abstract syntax tree. An intermediate representation of a design produced by a VHDL analyzer.

B

behavioral constructs
VHDL expressions that describe the behavior of a component.

binding specification
The assignment of specific components to specific entities. Configurations control binding.

bundles
A collection of signals with different names. A bundle is represented by names separated by a comma, for example `A,B,Data<0:15>`.

C

Component Description Format (CDF)
A system for dynamically changing and storing parameters and displaying information for components and sets of components for different versions (levels) of designs.

cell

The Cadence software representation of a design element or component. It can be viewed as a collection of views that describe an individual building block of a chip or system.

cellview

The Cadence software representation of a design unit. Views can be used to delineate between design representations, such as entity, schematic, or layout. Or they can be used to specify levels of abstraction; for example, behavior, RTL, or synthesis.

CIW

Command Interpreter Window. The primary user interface for launching Cadence software. The CDF editor starts from the *Tools* menu in the CIW. You can enter SKILL commands in the CIW command line.

.cod, COD

Short for code; machine instr

compilation

Also called *analysis*.

component

A fundamental unit within a system that encapsulates behavior and structure. Also known as an *element*. A cell, with cellviews and associated CDF.

component declaration

Identifies the name, ports, and generics of a component. A component must be declared before it is instantiated in an architecture. The component declaration can appear within the architecture or in a package.

component instantiation

Describes an instance of a component contained within an architecture.

configuration declaration

Describes how component instances are bound to design entities and how design entities are put together to form a complete design.

configuration specification

Binds component instances to entity architecture pairs.

construct

A statement or basic element in a programming language.

context clause

A statement in a design unit that identifies which elements in which libraries the design unit references.

CSI

Cadence-to-Synopsys Interface.

D

design unit

The basic building blocks of a VHDL design, including entity, architecture, package, package body, and configuration declarations.

design library

A library containing cells that describe components of a single design.

E

elaboration

Binds analyzed VHDL design units into a design that can be simulated.

entity declaration

Describes the interface to a component. Entities communicate through generics and ports.

environment variables

Values in the UNIX operating system that you set in UNIX files, such as the `.cshrc` file to control how the shell works.

escaped names

Identifiers that begin and end with a backslash are called escaped identifiers. For example, `\Name\`. Unlike normal VHDL names, escaped VHDL identifiers are case sensitive. This means that the identifier `\AbC\` and the identifier `\aBc\` refer to two different objects. An identifier in the VHDL normal name space and the same identifier in the VHDL escaped name space do not represent the same object. For example, the VHDL identifiers `abc` and `\abc\` refer to two different objects. To embed a backslash in an escaped identifier, use double backslashes. If the original identifier was in the VHDL escaped form even though it was legal in the VHDL normal form, it needs to be returned to the escaped form, not the normal form. For example, `\abc\` maps to `OA ESC_abc`, and it maps back to `\abc\` in VHDL. All alphanumeric characters and symbols, as well as spaces, are allowed as VHDL escaped identifiers.

F

form field

The area on a form where you indicate values, names, and selections.

G

generic

A constant declared in a component declaration or an entity declaration. The value of a generic can be supplied externally either in a component instantiation or in a configuration specification.

golden database

The complete and correct results of a simulation used as a measure against the success of other simulations.

I

IR

Intermediate representation.

L

LISP

An artificial intelligence language.

LSE

Language Sensitive Editor. A text editor with features specific to one programming or design language, such as syntax checking.

N

name mapping

To make data interoperable among Cadence tools, Cadence developed a common naming convention called name mapping. When tools use data from other applications with noncompatible naming conventions, the name mapping mechanism converts the names to a recognizable language that the tool understands.

name space

A set of rules – for example, a VHDL name space – for creating legal names that a particular tool or language uses for determining the types of identifiers and keywords that are legal for that tool.

ncvhdl

Cadence's NC-VHDL parser.

O

OSS

Open Simulation System.

P

package declaration

Describes information common to multiple design units.

package body

Describes the implementation of the declarations found in a package.

PI

Procedural Interface.

port

A signal declared in the interface list of an entity declaration or in the interface list of a component declaration. A port corresponds to a pin in a symbol or schematic.

primary design unit

Entity, package, and configuration declarations. Primary design units must be analyzed (compiled) before secondary design units.

R

reference library

A library containing cells that describes common components potentially used in many designs.

RTL

Register Transfer Level.

S

secondary design unit

Architecture and package body declarations. Secondary design units must be analyzed (compiled) after primary design units.

signal

Connections between component instances, providing for communication of dynamic data between components.

SHM

Simulation History Manager or Simulation History Management.

SKILL

A proprietary Cadence high-level interactive programming language based on the artificial intelligence language, LISP.

T

TDM

Team Design Manager.

type

Defines a set of values and a set of operations, such as bit, std_logic, or integer.

V

view

See cellview.

VHSIC

Very High Speed Integrated Circuit.

VHDL

VHSIC Hardware Description Language.

W

working library

The library where the result from analyzing a VHDL design unit is placed. There can be only one working library.