ESP
User Guide

Version L-2016.06, June 2016

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

8. Verifying Your Design

9. Debugging Your Design

Preface

This preface includes the following sections:

About This User Guide

The *ESP User Guide* describes how to set up the ESP tool, check equivalence between Verilog behavioral models and full-custom digital SPICE implementations, perform power integrity verification, and symbolically simulate your Verilog design and verify its functionality with your custom testbench.

Audience

The ESP tool is intended for designers of high-performance application-specific integrated circuits (ASIC) and structured custom IC. These users must verify that their designs functionally match the design specifications.

Related Publications

For additional information about the ESP tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys products:

- Formality®

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *ESP Release Notes* on the SolvNet site.

To see the *ESP Release Notes*,

1. Go to the SolvNet Download Center located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

2. Select ESP, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Online Man Page Conventions

Throughout the command descriptions in the online man pages, the arguments *containerID*, *libraryID*, *designID*, and *objectID* represent the names of containers, libraries, designs, and design objects, respectively. You specify information for these arguments as follows:

containerID

For *containerID*, you can explicitly specify the container name by supplying a unique character string. You can use any alphanumeric character in the string. Many commands do not require a *containerID* argument.

libraryID

For *libraryID*, specify the logic library name or design library name by supplying a slash character (/) followed by a character string. You can optionally begin the string with the *containerID* and colon (:). Use this form when supplying a *libraryID* argument:

```
[containerID:]/library_name
```

designID

For *designID*, you can explicitly specify the design name by supplying a string or implicitly specify the current design by omitting the *designID* argument. If you specify a design name, you must also supply the logic library or design library name in which it resides. Use this form when supplying a *designID*:

```
[containerID:]/library_name/design_name
```

Many commands do not require a *designID*. In these cases, ESP uses the current design.

objectID

For *objectID*, you must explicitly specify the object name by supplying a string. However, you can indicate the exact location of the object in several ways. You can supply a full path name by specifying container, library and design names. Omitting the *containerID* and *designID* implicitly specifies the current container and design. Use this form when supplying an *objectID*:

```
[[containerID:]/library_name/design_name/]object_name
```

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

ESP Overview

ESP is an equivalence checking tool commonly used for full functional verification of custom designs such as memories, custom macros, standard cell, and I/O cell libraries. It is used to ensure that two design representations are functionally equivalent.

This chapter includes:

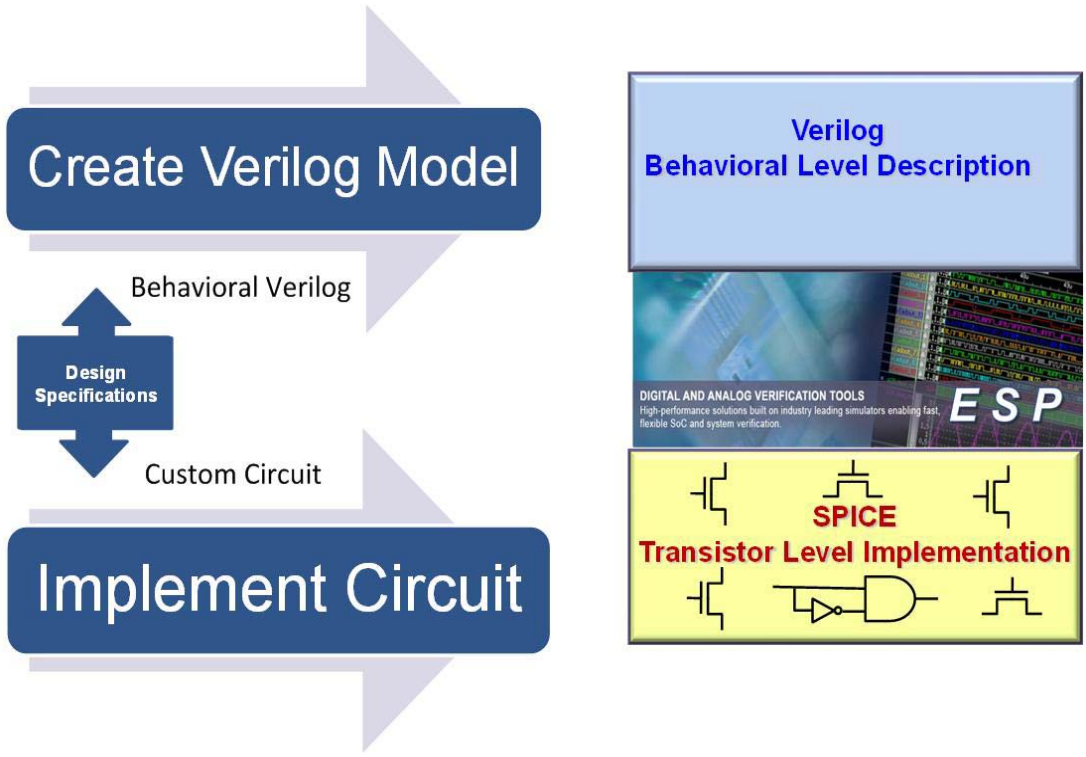
Introduction to ESP

The core technologies used in ESP include:

- Symbolic simulation
- Custom equivalence checking
- Circuit knowledge technology
- Hierarchical compression technology

The ESP tool uses symbolic simulation to verify two design representations. In traditional simulation, the simulator applies specific logic values or voltages to the circuit model to determine the circuit output. In symbolic simulation, the simulator uses symbols that represent values that are simultaneously both a 1 and 0, and stores the values on nets as equations. Equivalence is checked by simulating two designs and verifying that their outputs match over a number of cycles.

Figure 1-1 ESP Basic Diagram



You can use the ESP tool to perform equivalence checking between Verilog behavioral models and full-custom digital SPICE implementations, to perform power integrity

verification, and to symbolically simulate your Verilog or SPICE design and verify its functionality with your custom testbench.

Note:
ESP supports Standard Delay Format (SDF) annotated designs.

You can run ESP in three modes: compare, power integrity verification, and simulate-only. Each mode enables its respective flow. Use the `set_verify_mode` command to set a particular mode. The compare mode is the default.

- Compare Flow (default flow)

  The compare flow is also referred to as the compare mode. Use this flow to create a testbench that determines the functional equivalence between your Verilog reference design and SPICE implementation design. This is the default flow. This flow also supports Verilog-to-Verilog and SPICE-to-SPICE comparisons.

- Power Integrity Verification Flow

  Use this flow to verify power management techniques such as power domains and switching, retention modes, and isolation. The power integrity verification flow is also referred to as the power verification mode. To enable the power verification flow, specify the `set_verify_mode inspector` command. This flow expands the compare flow to include power integrity verification.

- Simulate-Only Flow

  Use this flow with your custom verification testbench to symbolically simulate your Verilog design and verify its functionality. To enable this flow, use the `set_verify_mode simulate` command. This flow also supports symbolic simulation of SPICE designs.

Input Requirements

The ESP tool accepts the following basic files:

- Verilog design

  Language support includes:

  - All syntax constructs from the 1995 Verilog Language Reference Manual (IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language; IEEE Standard number: IEEE Std 1364-1995).
  - Selected syntax constructs from the 2001 Verilog Language Reference Manual (IEEE Standard Verilog® Hardware Description Language; IEEE Standard number: IEEE Std 1364-2001).
  - Selected syntax constructs from the 2012 SystemVerilog Language Reference Manual (IEEE Standard for SystemVerilog®- Unified Hardware Design, Specification, and Verification Language; IEEE Standard number: IEEE Std 1800-2012).

- SPICE netlist (HSPICE and LV SPICE)
- Tcl commands that define the process technology, design constraints, generation of the testbench, and verification and reporting of coverage data

The ESP tool reads in SPICE netlists and Verilog models.

ESP writes out Verilog testbench templates for design types you define in your Tcl command file and SPICE testbenches to verify specific error patterns. After completing the verification run, the outputs from the ESP environment consist of verification reports, simulation log files and the ESP work directory.

User Interface

The ESP tool provides a Tcl shell command line interface (esp_shell). The ESP Tcl shell is used for interactive, script and batch mode operations.

ESP uses the tool command language (Tcl), which is used in many applications in the EDA industry. Using Tcl, you can extend the ESP command language by writing reusable procedures and scripts. For details, see the *Using Tcl With Synopsys Tools* manual.

With ESP, you can use the following Tcl interface features:

- Tcl command shell extended for Synopsys EDA applications
- ESP shell application commands
- Interactive operation

- Command abbreviation and completion
- Command scripts for batch runs and process automation
- User written procedures to extend the command set
- Command aliases
- Command help
- Command logging to file for replay
- Tcl built-in application and user-defined variables
- Wildcard lookup
- Output redirection to files
- Control over warning and informational messages
- Tcl shell access to man pages for commands, variables, errors, and warnings
- Design object collection mechanism with wildcard and filtering

2

Getting Started

To use the ESP tool, you need to set up the ESP environment. This chapter includes the following sections:

Setting Up the ESP Search Path

Before using the ESP tool, ensure that the search path contains the bin directory under the ESP installation directory. If the ESP tool is installed in the /u/tools/ESP directory, use the following command to add the bin directory to that path:

```
set path = (/u/tools/ESP/bin $path)
```

The ESP tool uses this mechanism to find its executable. It does not use the value of the SYNOPSYS environment variable if the path variable is defined when the `esp_shell` command is called. The ESP tool resets the SYNOPSYS environment variable to the installation path. After you exit ESP, the SYNOPSYS environment variable is reset to its original value.

Use the `env` Tcl array variable to access UNIX shell environment variables in the ESP shell as shown in the following example:

```
esp_shell (setup) > puts $env(PATH)
```

Using the Environment Setup File

You can customize the ESP tool by placing variable settings and initialization commands in the .synopsys_esp.setup environment setup file. The examples in the ESP Help system assume that you invoke the `esp_shell` command to invoke the ESP utilities.

The ESP environment setup file .synopsys_esp.setup, is located in the *esp_installation_directory*/admin/setup directory.

The tool searches for Tcl command files in several locations and processes them in the following order:

1. ESP installation directory
2. Your home directory
3. Your current working directory

For example, if the `set_verify_mode` command appears in each .synopsys_esp_setup file, then the command in the local working directory takes precedence.

Enabling License Queuing

To enable license queuing, set the UNIX environment variable, ESP_WAIT_LICENSE, to the value 1 as shown in the following example:

```
% setenv ESP_WAIT_LICENSE 1
% esp_shell -f run.tcl
```

Starting ESP

To start ESP, enter the following command at the operating system prompt (%):

```
% esp_shell
esp_shell (setup)>
```

The tool returns the ESP copyright or license notice, program header, and the ESP shell prompt. The shell interprets and executes ESP verification commands that are based on Tcl.

Use the following command-line options when you start the ESP tool:

`-file` *`file_name`*

Executes *file_name* (a file of esp_shell commands) before displaying the initial esp_shell prompt. If the last statement in *file_name* is quit, no prompt is displayed and the command shell exits.

```
% esp_shell -file myfilename
```

`-no_init`

Directs esp_shell to ignore the .synopsys_esp.setup files.

The following example ignores the .synopsys_esp.setup file and executes the commands in the esp_shell_command.log.copy file:

```
% esp_shell -no_init -file esp_shell_command.log.copy
```

`-version`

Prints the version number, then exits.

You can use `esp` as an alias for the `esp_shell` command.

Working With ESP

To use the ESP shell, you need to know how to enter commands, supply arguments and set variables.

This section includes:

- Entering Commands
- Supplying Lists of Arguments
- Setting Variables

Entering Commands

The ESP shell considers case when it processes commands. All command names, option names, and arguments are case-sensitive. For example, the following two commands are not equivalent:

```
esp_shell (setup)> read_verilog -r top.v
esp_shell (setup)> read_verilog -R top.v
```

The second command is an error because only `-r` is a valid option. The ESP shell returns the following error message for the `-R` option:

```
Error: Unknown option '-R'
```

Each `esp_shell` command returns a result that is a string, an integer (1 or 0), a Boolean (true or false) value, or a complex data structure. The ESP shell can pass the result of a command directly to another command. For example, the following command uses an expression to derive the right side of the resulting equation:

```
esp_shell (setup)> echo 3+4=[expr 3+4]
3+4=7
```

When you enter a long command with many options and arguments, you can extend the command across more than one line by using the backslash (\) continuation character. During a continuing command input (or in other incomplete input situations), ESP shell displays a question mark (?) as a secondary prompt. Here is an example,

```
esp_shell (setup)> read_verilog -r top.v \
? bottom.v
Loading verilog file...
Current container set to 'r'
1
esp_shell (setup)>
```

Supplying Lists of Arguments

When supplying more than one argument for a given ESP shell command, adhere to Tcl rules. This section highlights some important Tcl language concepts:

- Command arguments and results are represented as strings; lists are represented as strings, but with a specific structure.
- Lists are typically entered by enclosing a string in braces:

  ```
  {file_1 file_2 file_3 file_4}
  ```

  The equivalent Tcl list command is as follows:

  ```
  [list file_1 file_2 file_3 file_4]
  ```

  Note:

  Do not use commas to separate list items. Do not use braces to substitute variables. Use the list format to substitute a variable.

- If you are attempting to substitute a command or a variable, the brace format does not work. For example, the following command reads a single file that contains a single Verilog file. The file is located in a directory that the `DESIGNS` variable defines.

  ```
  esp_shell(setup)> read_verilog -r $DESIGNS/ram_bottom.v
  Information: Analyzing source file ‘/u/project/designs
         ram_bottom.v’
  Information: Analyzing module ( ram1r1w_xtor_orig ).
  Information: 0 parse error(s) and 0 warning(s).
  1
  esp_shell (setup)>
  ```

  You cannot read two files with the following command because the variable is not expanded within the braces.

  ```
  esp_shell(setup)> read_verilog -r {$DESIGNS/ram_top.v \
   $DESIGNS/ram_bottom.v}
  Error: File “$DESIGNS/ram_top.v” does not exist (ESPVER-842)
  0
  esp_shell (setup)>
  ```

The `list` command expands variables as follows:

```
esp_shell(setup)> read_verilog -r [list $DESIGNS/ram_top.v \
 $DESIGNS/ram_bottom.v]
Information: Analyzing source file "/u/project/designs/
ram_top.v"
Information: Analyzing module ( ram1r1w_xtor ).
Information: Analyzing source file "/u/project/designs/     \
 ram_bottom.v"
Information: Analyzing module ( ram1r1w_xtor_orig ).
Information: 0 parse error(s) and 0 warning(s).
1
esp_shell (setup)>
```

You can also enclose the design list in double quotation marks to expand variables as follows:

```
esp_shell (setup)> read_verilog -r "$DESIGNS/ram_top.v    \
 $DESIGNS/ram_bottom.v"
Information: Analyzing source file "/u/project/designs/
ram_top.v"
Information: Analyzing module ( ram1r1w_xtor ).
Information: Analyzing source file "/u/project/designs/     \
 ram_bottom.v"
Information: Analyzing module ( ram1r1w_xtor_orig ).
Information: 0 parse error(s) and 0 warning(s).
1
esp_shell (setup)>
```

Setting Variables

Use the `set_app_var` command to set ESP application variables. For example,

```
esp_shell (setup)> set_app_var testbench_style sram
sram
```

Note that the `set_app_var` command can also be used to set non-ESP variables. However, an error message is issued if you use the `set_app_var` command to set a non-ESP application variable. In Example 2-1, the `set_app_var` command is used to set a non-ESP application variable (`my_var`) and the tool issues a CMD-104 error message.

Example 2-1 Error Message Issued When Setting the set_app_var Command to Non-ESP Variables

```
esp_shell (setup)> set_app_var myvar bar
Error: Variable 'myvar' is not an application variable. Value will still
be set in Tcl. (CMD-104)
bar
```

The ESP tool issues an error message, but also executes the command and sets the `myvar` variable to the value `bar`.

When you invoke the tool, the ESP tool defines the application variables and assigns the defaults.

Specify the `man` command within the ESP shell to view information, including the defaults, for the variables. For example,

```
esp_shell (setup)> man coverage
NAME
   coverage       Enable coverage data gathering during verification.
TYPE
   Boolean
DEFAULT
   false
```

Custom Compiler Integration

You can use ESP as a verification tool from Custom Compiler. You can invoke ESP shell within Custom Compiler, verify your design, debug problems using highlighted error markers on the schematic, and bring up WaveView to check the graphical simulation results.

Note:

Before using Custom Compiler, you need to set up the ESP tool. For more information, see Setting Up the ESP Search Path and Using the Environment Setup File.

To set up the environment for ESP integration with Custom Compiler, do the following:

- Create the environment variable for your ESP installation directory:

  ```
  setenv ESP_INSTALL_DIR Your_ESP_installation_directory
  ```

- Create the environment variable to direct Custom Compiler to the ESP Custom Compiler integration environment:

  ```
  setenv SYNOPSYS_CUSTOM_SITE $SYNOPSYS_ESP_ROOT/auxx/esp/cdesigner
  ```

- Add ESP, Custom Compiler, and WaveView to your PATH environment variable

A typical session includes the following steps:

1. Start Custom Compiler.
2. Open a schematic for verification.
3. Choose Tools > Simulation.
4. Choose Simulation > Initialize > New... (or Load...).
5. Set the simulator to ESP and change the run directory, as needed.
6. Click the Create button.

7. Choose Simulation > Inputs > Edit... (or Load...).
8. Modify the control script as needed.
9. Choose Simulation > Verify with ESP.
10. Enter a file name with Verilog Options.

    This file is a Verilog `-f` control file and typically contains the name of the Verilog behavioral model and any other Verilog command options such as `-y` or `-v`.
11. Click OK.
12. The ESP shell appears. Complete your work within ESP as needed.
13. Exit the ESP shell by typing exit.
14. Look at the outputs using the WaveView within Custom Compiler.

For more details, see the `custom_compiler` flow man page.

Useful Commands

The ESP tool includes many useful commands to recall the previous sessions, redirect the output. You can list the commands entered during a session. Command aliases create short forms for the commands you commonly use.

This section includes:

- Listing the Commands Entered During a Session
- Recalling Commands
- Redirecting Output
- Using Command Aliases

Listing the Commands Entered During a Session

The `history` command lists the previous commands that you entered. This command does not have any arguments. The most recent 20 commands are listed by default.

The syntax is as follows:

```
history [keep number_of_lines] [info number_of_entries]
[-h] [-r]
```

`keep` *`number_of_lines`*

Changes the length of the history buffer to the number of lines you specify. The default is a maximum of twenty events returned.

`info` *`number_of_entries`*

Limits the number of lines displayed to the specified number.

`-h`

Shows the list of commands without leading numbers.

`-r`

Shows the history of commands in reverse order. The most recent history entries are displayed first.

You can use the `keep` argument to change the length of the history buffer. To specify a buffer length of 50 commands, enter the following command:

```
esp_shell (setup)> history keep 50
```

You can limit the number of entries displayed, regardless of the buffer length, by using the `info` argument. For example, enter

```
esp_shell (setup)> history info 3
 10  unalias warnings_only
 11  history
 12  history info 3
esp_shell (setup)>
```

You can also redirect the output of the `history` command to create a file and use it as the basis for a command script. For example, the following command saves a history of commands to the my_script file:

```
esp_shell (setup)> redirect my_script { history }
```

Recalling Commands

Use the following UNIX-style shortcuts to recall and execute previously-entered commands:

`!!`

Recalls the last command.

`!-`*`n`*

Recalls the *n*th command from the end of the history list.

`!`*`n`*

Recalls the command numbered *n* from a history list.

`!`*`text`*

Recalls the most recent command that started with *text*; *text* can begin with a letter or underscore (_) and can contain numbers.

The following example recalls and runs the most recent verification command:

```
esp_shell (verify)> !ver
verify
    .
    .
    .
esp_shell (verify)>
```

The following example recalls and starts the last run command:

```
esp_shell (setup)> !!
 1 unalias warnings_only
 2 read_verilog -r top.v
esp_shell (setup)>
```

Redirecting Output

Use the `redirect` command or the > and >> operators to make the ESP shell redirect the output of a command or a script to a specified file.

You can redirect the output to a file using the `redirect` command as follows:

```
esp_shell(setup)> redirect file_name "command_string"
```

To redirect the output to a file, use the > operator:

```
esp_shell(setup)> command > file
```

If the specified file does not exist, the ESP shell creates it. If the file does exist, the ESP shell overwrites the file with the new output.

For more information, see the `redirect` command man page.

Use the following command to append the output to a file:

```
esp_shell (setup)> command >> file
```

Unlike UNIX, the ESP shell treats the > and >> operators as arguments to a command. You must use spaces to separate the operator from the command and from the target file. In the following example, the second line is the correct input:

```
esp_shell (setup)> echo $my_variable>>file.out #incorrect usage
esp_shell (setup)> echo $my_variable >> file.out #correct usage
```

Note:

The built-in `puts` command does not redirect the output. ESP shell provides a similar command, `echo`, that allows output redirection.

Using Command Aliases

You can use aliases to create short forms for the commands you commonly use. For example, the following `check` command creates an alias that invokes the `check_design` command:

```
esp_shell (setup)> alias check "check_design"
```

After creating the alias, you enter the `check` command at the esp_shell prompt.

The following conditions apply to alias behavior and usage:

- Aliases are recognized only when they are the first word of a command.
- Alias definitions take effect immediately and remain in effect for the duration of the ESP shell session.
- The ESP shell runs the commands in the .synopsys_esp.setup file during startup; therefore, you can use this file to define commonly used aliases.
- An alias name cannot be the same as an existing command name.
- Aliases can refer to other aliases.
- Supply arguments when defining an alias by surrounding the entire alias definition in quotation marks.

  Note:

  Use aliases only for interactive work. Scripts should use the complete command name. Use a Tcl procedure to create short forms of commonly used command sequences in a script.

Using the alias Command

The syntax of the `alias` command is

`alias [`*`name`* `[`*`definition`*`]]`

`name`

Represents the name (short form) of the alias you are creating (if a definition is supplied) or listing (if no definition is supplied). The name can contain letters, digits, and the underscore character (_). If no name is given, all aliases are listed.

`definition`

Represents the command and list of options for which you are creating an alias. If an alias is already defined, *definition* overwrites the existing definition. If no *definition* is defined, the definition of the named alias is displayed.

When you create an alias for a command that contains dash options, enclose the whole command in quotation marks.

Using the unalias Command

The `unalias` command removes alias definitions.

The syntax of the `unalias` command is

`unalias [`*`pattern`*`...]`

`pattern`

Lists one or more patterns that match existing aliases whose definitions you want removed. For example, use the following command to remove the *`check_design`* alias:

```
esp_shell (setup)> unalias check_design
```

Accessing Task Man Pages

To access man pages for tasks, do not use “$” in the command. For example, to access the `$esp_equation` man page, use the `man esp_equation` command.

3

Compare Flow

In a standard ESP flow, the tool uses your reference design (Verilog) and implementation design (SPICE) to create a testbench to determine the functional equivalence of the designs. This is referred to as the compare flow. The ESP tool enables the compare flow by default.

This chapter includes:

Compare Flow Overview

In the compare flow, the ESP tool compares the design in the reference container to the design in the implementation container. The tool can automatically create a testbench to compare the two designs or you can specify your own testbench.

The ESP shell prompt indicates the part of the tool that is currently active.

Table 3-1 lists the ESP functions and their prompts.

Table 3-1 ESP Functions and Associated Prompts

Function	Prompt
Setup	esp_shell (setup)>
Match	esp_shell (match)>
Configure	esp_shell (config)>
Verify	esp_shell (verify)>
Debug	esp_shell (explore)>

To enable the compare flow, use the `set_verify_mode compare` command. The compare flow is the default flow.

For more information about advanced flows, see Chapter 11, “Advanced ESP Flows”.

Compare Flow Methodology

The typical ESP flow for memory verification consists of the following steps:

1. Prepare the design for verification.

   For details, see Chapter 4, “Preparing to Verify a Design.”

2. Read the reference design.

   The reference design is typically a Verilog behavioral model. For details, see “Reading in Reference Designs” in Chapter 5.

3. Read the implementation design.

   The implementation design is typically a SPICE netlist. When using ESP device simulation models, run the `set_process` command before specifying the SPICE netlist. For details, see “Reading in Implementation Designs” in Chapter 5.

   The implementation design can also be a Verilog design. See “Verilog-to-Verilog Equivalence Checking” in Chapter 3”.

4. Match the ports between the two designs.

   For details, see Chapter 6, “Matching Reference and Implementation Ports.”

5. Configure the testbench.

   Specify the testbench style, input pin functions, clock pins, supply names, and virtual supply names. Alternatively, specify the custom testbench to use for verification.

   For details, see Chapter 7, “Configuring Supply Nets, Virtual Supplies, and Testbenches.”

6. Verify the designs.

   For details, see Chapter 8, “Verifying Your Design.”

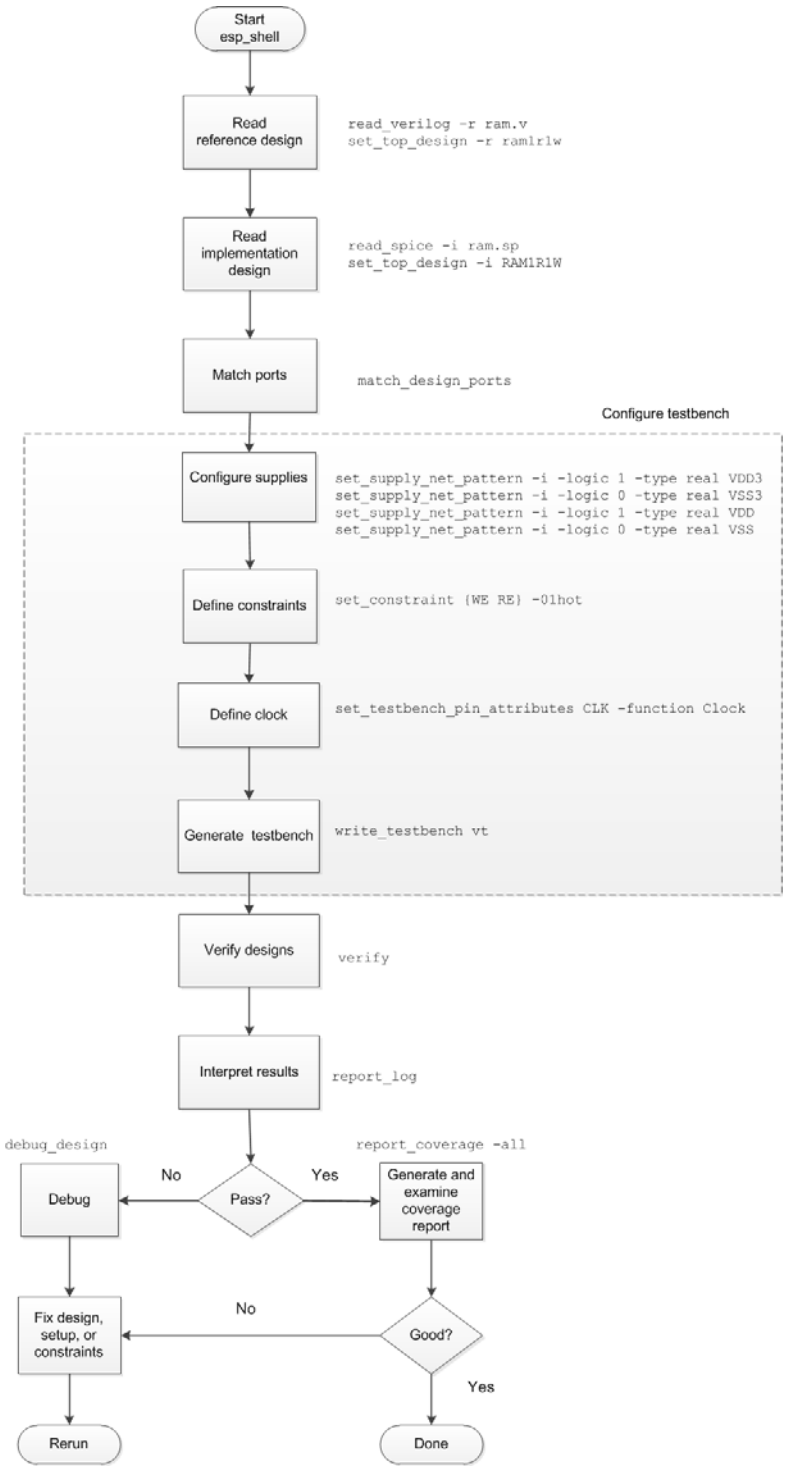
   For reporting options, see “Reporting and Interpreting Results” in Chapter 8.

7. Debug any failures.

   Typical Verilog debugging techniques with a waveform viewer can be used. ESP offers an Interactive Signal Tracing mode that gives you additional visibility into the SPICE netlist. Finally, the ESP tool generates a testbench to verify the failure using HSPICE or a fast-SPICE simulator. For details, see Chapter 9, “Debugging Your Design.”

Figure 3-1 shows the compare flow with the equivalent commands.

Figure 3-1 Compare Flow Flowchart With Command Examples



Verilog-to-Verilog Equivalence Checking

To support the verification flow, the ESP tool uses containers to hold the reference and implementation designs. The ESP shell allows Verilog designs in both the reference and the implementation containers.

Example 3-1 Verilog-to-Verilog Equivalence Checking Script

```
read_verilog -r ram.v
set_top_design -r RAM
read_verilog -i ram_rev21.v
set_top_design -i RAM
match_design_ports
set_testbench_pin_attributes -function read RE
set_testbench_pin_attributes -function address A
set_testbench_pin_attributes -function clock CLK
set_testbench_pin_attributes -function data DI
set_app_var testbench_style sram
verify
```

The script in Example 3-1 performs equivalence checking between two versions (ram.v and ram_rev21.v) of a behavioral Verilog description.

During Verilog-to-Verilog equivalence checking, the default output checker performs a strict comparison where a value X on the reference design only matches to a value of X on the implementation design. Any other value generates a mismatch.

Running the following command at each of the outputs provides the same results:

```
set_testbench_pin_attributes -checker compare pin_name
```

Note:

This is different from the default comparison for design outputs during Verilog-to-SPICE equivalence checking which allows a value of X on the Verilog reference design to match any value (0, 1, X, Z) on the corresponding SPICE implementation design output.

You can still use the `set_testbench_pin_attributes` command to change the `-checker` option to any of its available values.

You can use different compiler directives for the two containers. Therefore, if you have a single set of Verilog source files in which `` `ifdef `` statements determine different behaviors, you can compare the two different behaviors by reading in the same source files using different +define+ directives. For example,

```
read_verilog -r ram.v -vcs {+define+Behav}
set_top_design -r RAM
read_verilog -i ram.v -vcs {+define+RTL}
set_top_design -i RAM
...
```

To perform Verilog-to-Verilog equivalence checking, you must use the VCS-based Verilog language parser. To invoke the ESP shell using the VCS-based parser, use the following command:

```
% esp_shell -parser_vcs ...
```

Alternatively, to access the VCS-based Verilog language parser, set the UNIX environment variable ESP_PARSER to VCS before running the ESP shell. This variable can be set to either ESP or VCS.

For example, you can set the ESP_PARSER variable as follows:

```
% setenv ESP_PARSER VCS
% esp_shell ...
```

For more information, see the `verilog_to_verilog` flow man page.

Specifying the Timescale for Verilog Source Files

To specify the timescale for Verilog source files before the first occurrence of the `timescale directive, use the `-timescale` argument with the `read_verilog -vcs` command as shown in the following example:

```
# run.tcl
# The following command uses the -timescale option
# to specify a time unit of 1ns and a precision of 1fs.
set_verify_mode simulate
read_verilog -r {test1.v test2.v} -vcs { -timescale=1ns/1fs }
verify -verbose

//Contents of the test1.v file
module test1;
initial $printtimescale;
endmodule

//Contents of the test2.v file show how to specify a default timescale
`timescale 100ps/10ps
module test2;
initial $printtimescale;
endmodule

esp_shell -f run.tcl
...
//The output obtained using the run.tcl script
... Starting ESP simulation: November 7, 2014 09:12:57
Time scale of (test1) is 1ns / 1fs
Time scale of (test2) is 100ps / 10ps
```

Parameterized RTL Model Support

The ESP shell supports parameters for top-level Verilog designs. The parameters for a top-level Verilog module are specified using the `-pvalue` option of the VCS simulator. The tool supports parameter overrides for any Verilog design.

Consider the following Verilog design:

```
module test #( parameter width1=4,width2=5) (C, A, B);
input [width1:0] A;
input [width2:0] B;
output C;
wire C;
assign C =1;
endmodule // test
```

The ESP tool performs Verilog-to-Verilog verification by overriding the value of width1 as 2 and width2 as 3 using the following Tcl script:

```
read_verilog -r test.v -vcs "-pvalue+width1=2 -pvalue+width2=3"
read_verilog -i test.v -vcs "-pvalue+width1=2 -pvalue+width2=3"
match
verify
report_log
quit
```

Note:

When you invoke the ESP shell without the `-parser_vcs` option, using the `-pvalue` option generates the following warning message:

```
Warning: Unknown command line option ( -pvalue+width1=4 ) is ignored.
```

The ESP tool accepts the value specified with the `-pvalue` option. You can ignore the warning message. When you invoke the ESP shell tool with the `-parser_vcs` option, the usage of the `-pvalue` option does not generate any warning message.

For reporting options, see “Reporting and Interpreting Results” on page 8-6.

4

Preparing to Verify a Design

Before you verify designs in the compare flow, you must prepare the designs for verification in the ESP environment. To verify a design, you must specify the location of the design files. This chapter discusses how to read Verilog and SPICE design files into the ESP environment.

The following sections describe how to prepare your design for verification:

Using Design Files

In the compare flow, the ESP tool uses automated testbenches to verify a reference or golden design against an implementation design. To support this verification flow, ESP uses containers to hold the reference and implementation designs.

ESP commands such as `current_container`, `current_design`, and `get_designs` operate on the reference or implementation design. These commands use options to indicate the design container. The `-r` option refers to the reference design container. The `-i` option refers to the implementation design container. If you do not specify the`-r` or `-i` option, the tool uses the default container. The default container is the reference container.

To change the default container, use the `current_container` command as follows:

`current_container ref`

Sets the default container to the reference container.

`current_container imp`

Sets the default container to the implementation container.

Within each container, you must designate one module (or for SPICE one subcircuit) as the top design. To designate a specific module as the top design, use the `set_top_design` command.

For SPICE-based designs, you must designate transistor device models before reading in the SPICE design. See “Defining SPICE Device Models” on page 4-4. If you do not define the SPICE device models before reading the SPICE design, the tool issues an error message.

You can read designs into the ESP environment in the following formats:

- Verilog

  Verilog language files that represent a design.

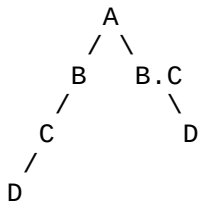
- SPICE

  SPICE netlist files that represent a design.

Using Hierarchical Separators

Consider the design in Example 4-1. This design contains a hierarchical cell A, which contains hierarchical cells B and B.C. In addition, cell B.C contains cell D, cell B contains cell C, and cell C contains cell D.

Example 4-1 Hierarchical Design



The `set_hierarchy_separator` command specifies an ASCII character to delimit hierarchical levels.

In most situations, you should use the default, which is the period character. In some cases where the hierarchy character is embedded, a search through the design might return incorrect results. The `set_hierarchy_separator` command provides a convenient way to deal with this situation.

In this hierarchical design, searching for “A.B.C.D” is ambiguous with the default settings. However, if you modify the hierarchy separator with the forward slash,

```
set_hierarchy_separator /
```

searching for “A/B.C/D” or for “A/B/C/D” is unambiguous.

Configuring SPICE Technology

This section discusses how to configure SPICE model technologies and includes the following:

- Recognizing SPICE Nets as Buses
- Controlling SPICE RC Decay Time
- Defining SPICE Device Models
- Device Model Simulation

Recognizing SPICE Nets as Buses

When you read SPICE netlists into the ESP environment, a pair of angle-bracket characters (< >) is used as the bus dimension separator. For example,

```
BUSA<1>
BUSA<2>
BUSB<1><1>
BUSC<1><2>
```

Your design might not follow the default bus naming scheme. If not, you can change the bus dimension separator by setting the `netlist_bus_extraction_style` environment variable. The values for the `netlist_bus_extraction_style` environment variable are `%s<%d>` (the default), `%s[%d]`, `%s_%d_`, and `%s_%d`, where `%s` is an arbitrary string and `%d` is a decimal integer value.

Controlling SPICE RC Decay Time

The `set_rcdecay_time` command supports infinite decay time, disabling RC decay and treating all SPICE nets as Verilog trireg nodes. The `set_rcdecay_time` command specifies the decay time for all SPICE nets when no driver is active. To enable infinite decay time, set the decay time to infinity (or inf).

For example, to set all SPICE nets to infinite decay time when all the drivers of a given node are turned off, use the following command:

```
set_rcdecay_time inf
```

Defining SPICE Device Models

You can define SPICE device models before reading in a SPICE file. For example, you might want to indicate whether a given transistor model describes an n-channel metal oxide semiconductor field-effect transistor (NMOS) or p-channel metal oxide semiconductor field-effect transistor (PMOS) device.

To define SPICE device models in the ESP shell, use the following example:

```
set_device_model -i -type nmos | pmos SPICE_model_name
```

If you do not provide any SPICE technology files, the ESP tool uses default models based on the Arizona Predictive Technology Models. If both W and NFIN are present in a SPICE netlist and you do not provide a technology file, the ESP tool uses a FinFET technology model and ignores the specified W parameter.

For further information about the Arizona Predictive Technology Models, see the following publications:

- W. Zhao, Y. Cao, "New generation of Predictive Technology Model for sub-45nm early design exploration," IEEE Transactions on Electron Devices, vol. 53, no. 11, pp. 2816-2823, November 2006.
- Y. Cao, T. Sato, D. Sylvester, M. Orshansky, C. Hu, “New paradigm of predictive MOSFET and interconnect modeling for early circuit design,” pp. 201-204, CICC, 2000.

Device Model Simulation

Device model simulation helps you to obtain transistor device information conveniently and precisely. It provides better results when calculating RC delay model simulation and is the preferred method to obtain better device models.

To obtain transistor device information, use the `add_device_model` and `create_model_library` commands and specify the SPICE simulator to simulate and create the SPICE library. Execute this method one time for a library. The tool makes the device model file available during simulation using the `set_process` command.

The `add_device_model` command defines a list of devices for the `create_model_library` command to simulate. You can execute `add_device_model` commands multiple times before running the `create_model_library` command.

The ESP tool selects a variety of length and width value combinations when you specify simulation points using the `-lrange` and `-wrange` options of the `add_device_model` command. The ESP tool selects more points close to the minimum value and fewer points near the maximum value. The circuit simulator simulates all the selected combinations of length and width.

When you specify the `add_device_model -cpoints` command, the ESP tool uses explicit length and width combinations to simulate these points and does not simulate any additional points. You can specify multiple `add_device_model` commands for the same devices. This is useful when most of the length and width values used are in one range and only a few values are significantly outside the general range.

The `create_model_library` command calculates the device model attributes that the `add_device_model` commands define. By default, the ESP tool uses the HSPICE simulator for circuit simulation. To specify how to invoke the SPICE circuit simulator, set the `spice_simulator` environment variable. Use the `create_model_library -simulator` command to specify the circuit simulator type: HSPICE, FINESIM ELDO, or SPECTRE. The netlist format and the file names follow the convention of the specified circuit simulator.

Example 4-2 shows commands that are used to simulate a basic NFET and PFET for a 28 nm library operating at 0.9 V.

Example 4-2 Command Example

```
esp_shell> add_device_model -ntype nch_mac -ptype pch_mac \
           -lrange {30n 1u} -wrange {60n 1u} -voltage 0.9 -macromodel
esp_shell> create_model_library -spicelib mylib.sp -edmfile my28nm_tp.edm
```

Example 4-3 shows an ESP simulation instantiating the transistors by setting the process parameters.

Example 4-3 Command to Set the Process Parameters

```
esp_shell> set_process -i -technology_file my28nm_tp.edm
```

Example 4-4 shows a command to set the RC delay rounding to 1 picosecond.

Example 4-4 Command to Set the RC Delay Rounding

```
esp_shell> set_app_var verify_delay_round_multiple 1
```

Example 4-5 shows how to use FineSim for device model simulation by using the `FINESIM` value with the `create_model_library -simulator` command:

Example 4-5 Command to Use FineSim for Device Model Simulation

```
esp_shell> create_model_library -spicelib lib.l -edmfile ptm20mg.edm \
-simulator FINESIM -replace
```

The ESP tool supports the NF parameter (number-of-fingers) for FinFET designs. The NF parameter is a critical parameter that affects the gate capacitance and drive strength for transistors in FinFET technologies and is interpreted differently for planar designs.

For more information, see the man pages for the following:

- Commands
  - `add_device_model`
  - `create_model_library`
  - `set_process`
- Application variables
  - `spice_simulator`
  - `verify_delay_round_multiple`
- Flow
  - `device_model_simulation`

FinFET Transistor Models

You can create an ESP device model simulation for FinFET transistor models using device model simulation.

Use the NFIN value as the width value for the `-frange` or `-fcpoints` options.

Example 4-6 and Example 4-7 show the edm.tcl and lib.l files used to create ESP device models for FinFET transistors. You can get the SPICElibrary lib.l file from the foundry.

Example 4-6 Example edm.tcl to Create ESP Device Models

```
# edm.tcl creates the ESP device model file
add_device_model -voltage 0.9 -ntype {nfet} -ptype {pfet} \
      -lrange 20n -frange {1 3}
create_model_library -spicelib lib.l -edmfile ptm20mg.edm \
      -simulator HSPICE -replace
quit
```

Example 4-7 Example lib.l Library File

```
# lib.l
*first line comment/title
.inc ‘./PTM-MG/param.inc’
.lib ‘./PTM-MG/models’ptm20lstp
```

Generate the device model simulation file using the ESP shell:

```
% esp_shell -f edm.tcl
```

In ESP shell, use the `set_process` command to read the ESP device simulation model file before you read the SPICE design:

```
% set_process -i -technology_file ptm20mg.edm
```

Power-Up Reinitialization of SPICE Nodes

The ESP tool allows reinitialization of SPICE nodes after simulating a sequence of power-down then power-up.

At the beginning of simulation, the tool initializes all the SPICE nodes to a stable value. If you power down a portion of the design and then power it up, initialization does not occur. In some designs, an X condition might occur, which prevents the circuit from working correctly after power supply sequencing.

The `set_net_value` command forces a value on a net for a given condition and then releases that value after a specified hold time.

The syntax of the `set_net_value` command is as follows:

```
set_net_value
 -i
 -if <if_condition>
 -value <net_value>
 -hold <hold_time>
 -design <SPICE_subcircuit>
net_name
 -code <Verilog_code>
```

If you use the `-code` option, the tool inserts the specified Verilog code into the specified SPICE subcircuit.

Note:
The Verilog code is any Verilog code allowed in a Verilog module.

The net name and the `-if`, `-value`, and `-hold` options are not allowed with the `-code` option. The`-code` option allows flexibility in defining the conditions under which a node is forced and released.

```
set_net_value -i -design CLKGEN NET1 -value {1'b0} \
-if {SWITCH_VDD===1'b1} -hold 0.1
```

To force each instance of MUX2 to zero or one respectively, if the select line is X and both inputs are zero or one, specify the following command:

```
set_net_value -i -design MUX2 \
-code {
always @(*) begin
      case ({SELA,A,B})
            3'bx00: force OUT = 1'b0;
            3'bx11: force OUT = 1'b1;
            default: #0.0 release OUT;
      endcase
end
}
```

Controlling SPICE X Range Thresholds

You can define the node threshold values when transistors connected to both the power and ground drive a node simultaneously. To define the node threshold values, use the following command:

`set_drive_fight_x_range -i -low_threshold` *`value`* `-high_threshold` *`value`*

For example, consider a PMOS pull-up transistor whose source is connected to the power supply, an NMOS pull-down transistor whose source is connected to the ground, and the drains of both the transistors are connected to node A. When both transistors are turned on, they try to pull node A toward power and ground. If the pull-up transistor has an “on”

resistance of 1K ohms and the pull-down transistor has an “on” resistance of 3K ohms, the voltage of node A goes to 0.75 V.

A voltage of 0.75 V is considered a logic 1. The default thresholds are 0.499 V for the low threshold and 0.501 V for the high threshold. However, this 0.75 V voltage is considered logic X if the thresholds are expanded to 0.20 for the low threshold and 0.80 for the high threshold:

```
set_drive_fight_x_range -i -low_threshold 0.20 -high_threshold 0.80
```

However, node A is considered logic 0 if the thresholds are set to 0.79 for the low threshold and 0.81 for the high threshold:

```
set_drive_fight_x_range -i -low_threshold 0.79 -high_threshold 0.81
```

By expanding the drive fight range, you can determine if your design is functionally sensitive to transistor drive fights. For details, see the man page.

Controlling Partial Transitions

The ESP tool allows you to control the handling of large delay nets.

Use the following variables to enable this control:

- `verify_partial_transition_threshold`
- `verify_partial_transition_sensitivity`

By default, ESP considers delays more than three inverter delays as large delays. For large delays, during the transition time, the delays on stages that follow the transitional net become larger than the normal delay.

The `verify_partial_transition_threshold` variable controls the amount of delay that triggers delay modification. A value of 0 means the ESP tool uses the delay value calculated during simulation.

The `verify_partial_transition_sensitivity` variable controls the amount of delay modification for the stage that follows a transitional net. A value of 0 means the tool uses a calculated default. Typical values for the `verify_partial_transition_sensitivity` variable range from 2 to 5.

5

Reading In Designs

This chapter discusses how to read Verilog and SPICE design files into the ESP environment for the compare flow.

The following sections describe how to read in your designs for verification:

- Design Language Support
- Reading in Reference Designs
- Reading in Implementation Designs

Design Language Support

The ESP tool can verify designs written in a subset of SystemVerilog, SPICE, or the Synopsys files compiled for Liberty.

The tool supports Verilog 1995 with some Verilog 2001 extensions as well as the design subset of SystemVerilog and Boolean assertions. “SystemVerilog Support” in Appendix A shows a subset of SystemVerilog that the ESP tool accepts.

The tool supports verification of digital designs described by a SPICE netlist. The ESP tool does not support bipolar junction transistors (BJT), inductors, transformers, current sources, voltage sources or other analog or RF design elements.

The tool supports verification of designs written in Liberty and compiled into the Synopsys binary DB format. The use of Liberty is only supported for cell library verification.

System Tasks

The ESP tool supports a variety of system tasks, including standard Verilog tasks, Verilog tasks enhanced for the ESP tool, system tasks that are specific to the ESP tool, and symbolic coverage system tasks. The following sections describe system tasks that the ESP tool supports:

- Standard Verilog System Tasks
- System Calls
- System Tasks Related to PLAs
- System Tasks Related to the ESP Tool
- ESP Symbolic Coverage System Tasks
- Waveform Dumping Tasks
- Verilog Coding Guidelines and Unsupported Constructs

Standard Verilog System Tasks

The ESP tool supports many standard Verilog system tasks. The tasks that ESP does not currently support are added on a need basis and defined in the System Calls section.

Table 5-1 shows a partial list of important Verilog system tasks and functions that the ESP tool supports:

Table 5-1 Verilog System Tasks and Functions

`$bitstoreal`	`$finish`	`$realtime`	`$timeformat`
`$clog2()`	`$fmonitor`	`$realtobits`	
`$dumpfile`	`$monitor`	`$stime`	
`$dumpvars`	`$readmemb`	`$stop`	
`$fdisplay`	`$readmemh`	`$time`	

System Calls

The ESP tool supports all I/O tasks, simulation control tasks, and `$random`, `$time`, and `$stime` tasks.

The tool does not support the following system tasks and functions:

- Timescale tasks

  The tool does not support the *`path`* option of the `$printtimescale` task, but supports the `$printtimescale` task without options. For example, `$printtimescale()` works whereas `$printtimescale(mytop.u1)` is not supported.

- Stochastic analysis tasks

  Unsupported tasks include:

  ```
  $q_add
  $q_exam
  $q_full
  $q_initialize
  $q_remove
  ```

- Probabilistic distribution functions

  Unsupported functions include:

  ```
  $dist_exponential
  $dist_chi_square
  $dist_erlang
  $dist_normal
  ```

```
$dist_poisson
$dist_t
$dist_uniform
```

- Miscellaneous system functions

  Other unsupported system functions include:

```
$countdrivers
$plus$val
```

Note:

You can find more detailed explanations for these tasks and functions in Section 14 of the 1995 Verilog Language Reference Manual (IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language; IEEE Standard number: IEEE Std 1364-1995).

System Tasks Related to PLAs

The ESP tool supports all system tasks related to programmable logic arrays (PLA). The system tasks associated with PLAs use the following format:

```
$arraytype$logic$format(memname,parameters);
arraytype := sync|async
logic := and|or|nand|nor
format := array|plane
```

For example,

```
$async$and$array(mem1,.....);
$sync$nor$plane(mem2,.....);
```

System Tasks Related to the ESP Tool

The following system tasks are specific to the ESP tool:

```
$esp_addrvar          $esp_datavar            $esp_multi_select

$esp_const            $esp_equation           $esp_onehotindex

$esp_const_one        $esp_equation_size      $esp_retire

$esp_const_zero       $esp_equation_symbols   $esp_smminit

$esp_contains         $esp_error              $esp_timevar

$esp_context          $esp_exclusive          $esp_var

$esp_ctrlvar          $esp_gen
```

For details about the tasks, you can

- Access the task man page. For details, see “Accessing Task Man Pages” on page 2-12.
- See the “Frequently Asked Questions > Verilog” section in ESP Help.

ESP Symbolic Coverage System Tasks

The following system tasks assist in gathering symbolic coverage information:

- `$espcovfile`
- `$espcovoff`
- `$espcovon`
- `$espcovscope`
- `$espexclcovscope`

For task details:

- Access the task man page. To access the man page for these tasks, do not use “$” in the command. For example, to access the $espcovfile man page, use `man espcovfile`.
- See the Symbolic Code Coverage section in ESP Help.

Waveform Dumping Tasks

The ESP tool supports a VCD system task that controls output file sizes.

For details, see “VCD Output File Support ($dumpclose)” on page 9-6. The tool also supports FSDB output file system tasks.

For details, see the “Frequently Asked Questions > Verilog” section in ESP Help.

Verilog Coding Guidelines and Unsupported Constructs

For Verilog coding guidelines, including symbolic simulation, and unsupported Verilog constructs, see the “Frequently Asked Questions > Verilog” section in ESP Help.

Note: See the relevant flow sections for more specific flow issues.

Reading in Reference Designs

To read the reference design into the current session in the ESP shell,

```
read_verilog -r file_names
```

You can use the `-vcs` option if you have VCS style options.

The ESP tool supports the `-parameters` option when using the VCS® parser. Use the `read_verilog -parameters` option to specify a parameter file that provides settings for Verilog source parameters. This option is similar to the `-pvalue` option.

A typical parameter file (params.h) contains the following lines:

```
assign 1 ptop.I
assign 2 ptop.R
```

These commands set the value of the `I` parameter in module `ptop` to `1` and the value of the `R` parameter to `2`.

To use the parameter file with the design.v Verilog design file, add the following command to your Tcl script:

```
read_verilog -r design.v -vcs { -parameters+parms.h }
```

The equivalent command using the `-pvalue` option is as follows:

```
read_verilog -r design.v -vcs { -pvalue+ptop.I=1 -pvalue+ptop.R=2 }
```

Though the VCS parser also supports the `-pvalue` option, the `-pvalue` and the `-parameter` options are mutually exclusive.

After reading the reference design, set the top-level design for the reference design in the ESP shell:

```
set_top_design -r design_name
```

where *design_name* is the name of the top-level Verilog module.

Reading in Implementation Designs

Reading the implementation design is similar to reading the reference design.

Note:

You can read only one implementation design during a verification session. To restart the ESP shell and remove the contents of a specified container, use the `reset` command or from the GUI, select Designs > Remove Reference, then select Designs > Remove Implementation.

To read the SPICE file for the implementation design in the ESP shell, use the following steps:

- Set the SPICE bus delimiter. For more information, see “Recognizing SPICE Nets as Buses” on page 4-4.
- For designs that use device models other than PMOS or NMOS, define the mapping of N and P type devices. For more information, see “Defining SPICE Device Models” on page 4-4.
- Specify the following:
  `read_spice -i` *`spice_file`*

  where `-i` signifies the implementation design

After reading the implementation design, set the top-level of the implementation design:

`set_top_design -i` *`design_name`*

where *design_name* is the name of the top-level SPICE subcircuit.

6

Matching Reference and Implementation Ports

This chapter describes port matching for the compare flow, the default ESP flow. For more information about the power intent verification flow and the simulation-only flow, read Chapter 11, “Advanced ESP Flows”.

To generate automatic testbenches, you must match the ports between the top design in the reference container and the top design in the implementation container. The matching process maps a port in the reference container to a port in the implementation container. The generated testbench connects matched ports with a name derived from the top design of the reference container. The ESP tool provides an automatic and manual (user-specified) matching method.

The following sections describe how to match reference and implementation ports:

- Automatically Matching Ports
- Manually Matching Ports
- Removing Matched Ports

Automatically Matching Ports

To automatically match the ports between the reference and implementation top-level designs, use the `match_design_ports` command. If you know that certain ports do not match, use the `set_matched_ports` command to manually match the ports before you run the `match_design_ports` command.

Manually Matching Ports

If automatic matching fails, you can manually match the ports between the reference and implementation designs. This is useful if you need to resolve issues with bit ordering on buses.

To manually match the ports between the reference and implementation top-level designs, use the following example:

```
set_matched_ports -r ref_ports -i imp_ports
```

where *ref_ports* is the reference port that you want to match with the implementation port, *imp_ports*.

Manual matching allows you to resolve the case where port b in the reference design is port a in the implementation and port a in the reference design is port b in the implementation.

If you know that certain ports do not match, use the `set_matched_ports` command to manually match these ports before you run the `match_design_ports` command.

If you manually map all the ports of your design, you do not need to use the `match_design_ports` command.

Removing Matched Ports

You can remove any matched port, from either automatic or manual matching in the reference and implementation designs. For example, if port matching is not correct because automatic matching failed or you made a mistake during the manual matching process, you can remove the match.

To remove a matched port, use the following example:

```
remove_matched_ports
-r ref_port_name
-i imp_port_name
```

where *ref_port_name* is the reference port and *imp_port_name* is the implementation port that you want removed. Specify the `-all` option to remove all the matched ports.

7

Configuring Supply Nets, Virtual Supplies, and Testbenches

The following sections describe how to configure ESP supply nets, virtual supplies, and testbenches for the compare flow:

Defining Supply Nets

To define supply nets for the implementation design in the ESP shell, use the following example:

```
set_supply_net_pattern
-i net_name
-type real | virtual
-logic 0 | 1
```

Recognizing Virtual Supplies

To enable the ESP tool to recognize virtual supplies by their connections in the netlist, enable the `netlist_supply_by_connection` variable:

```
set_app_var netlist_supply_by_connection on
```

For more information, see the `netlist_supply_by_connection` variable man page.

Defining Subcircuit-Based Virtual Supplies

To specify a virtual power or ground pin within the subcircuit of a design, use the `set_supply_net_pattern -design` command:

```
set_supply_net_pattern -i type virtual -logic 1 -design sw_pwr svdd
```

Creating Automatic Testbenches

The ESP tool has a testbench that defines the stimulus and expected responses. The testbench instantiates the reference and implementation designs. A comparison is made between these two designs at select points in time.

The ESP tool allows you to automatically generate testbenches. The following sections discuss the commands and shell variables that affect the content of the automatic testbench:

- Setting Net Delay
- Specifying Input Constraints
- Specifying Testbench Input Delay
- Specifying Output Constraints
- Setting the Testbench Style

- Variables That Affect Testbench Styles
- Configuring Testbench Pins
- Configuring Port Groups
- Defining Clock Objects
- Setting a Simulation Timescale
- Setting Port Group Pins
- Setting a Port Group Constraint
- Generating Automatic Testbenches
- Customizing Your Testbench

Setting Net Delay

To specify that a particular node within an implementation subcircuit has a given delay (regardless of the sizes of the transistors driving it), use the `set_net_delay` command.

For example, the following command causes node “y” within the subcircuit “invnetdelay” to have a delay of 20 ns (20 time units in which a time unit for the implementation design is 1 ns):

```
set_net_delay -i -delay 20 -design invnetdelay y
```

The `set_net_delay` command sets the delay for both the rising and falling transitions of the specified node. In RC mode (the default), all other nodes in the Channel Connected Region (CCR) containing the specified node have their delays set to zero unless they have a `set_net_delay` command applied to them. Note that this command also applies to the net. The specified delay applies for that net regardless of the hierarchical level used to specify the delay or the number of other drivers that connect to it.

For more information, see the `set_net_delay` command man page.

Specifying Input Constraints

You can set common constraints on input pins within the ESP shell. The following commands provide this capability:

- `set_constraint`
- `remove_constraint`
- `report_constraints`

For example, use the following command to declare two port address buses as mutually exclusive in all testbenches:

```
set_constraint -if "ADDR1 === ADDR2" -set "~ADDR2" ADDR1
```

To run a testmode pin named TEST and set it to zero, execute the following command:

```
set_constraint -set 0 TEST
```

For more information, see the `set_constraint`, `remove_constraint`, and `report_constraints` command man pages.

Specifying Testbench Input Delay

You can apply an input stimulus at different times compared to other input stimuli. Use the `set_input_delay` command to delay an input by a specified amount from when it would otherwise be applied. The delay can be as large as 50 percent of the clock period.

The `set_input_delay` command has the following syntax:

```
set_input_delay -delay timeValue pinName
```

The `timeValue` argument is in time units. The default time unit is nanoseconds. If the default clock period is 40 time units (40 ns), the delay value can be as large as 20 time units, which is 50 percent of the clock period. Larger values might result in the input change not occurring.

To specify a 300 ps input delay on the write-enable port WE, relative to the data port D, use the following command:

```
set_input_delay -delay 0.3 WE
```

The data port, D, changes at the default time of 10 ns before the active edge of the clock. The write enable port WE changes 0.3 ns later, or 9.7 ns before the active edge of the clock.

For more information, see the `set_input_delay` command man page.

The `report_log` command generates a report that distinguishes inputs delayed by the `set_input_delay` command from inputs that change at the default times, such as the start of a testbench cycle. The report displays `@+<delay>` after the input value of the signal.

For example, consider the signal WE that is delayed by 5 time units using the following command:

```
set_input_delay {WE} -delay {5.0}
```

The `report_log` command generates a report that contains lines similar to the following:

```
...
Checking DOUT 110
...
=========Input Signal Values========== 120
BINCYCLE 2
A = 7
DI = 80
WE = 0 @+5
RE = 1
=====================================
...
```

The report shows the values of the input signals A, DI, WE, and RE at time 120. For a symbolic style testbench (the default), the values of 7, 80, and 1 were applied to A, DI and RE at time 110, but the value 0 was applied to the signals WE_ref and WE_xtor at time 115 (WE_ref and WE_xtor are the signals directly feeding the reference and implementation designs).

Specifying Output Constraints

To create output constraints, use the `set_constraint -ignore` command.

Input constraints allow you to limit the possible values that are applied to the inputs of your design. Output constraints limit the situations in which an output's reference and implementation values are compared, but they do not limit the output values that result from your design. It is possible to implement output constraints by modifying the testbench through explicit edits or redefining tasks.

For example,

```
set_constraint DOUT -ignore -if {(RE !== 1'b1) || (OE !== 1'b1)}
```

For more information, see the `set_constraint` command man page.

Setting the Testbench Style

To configure the testbench style used during verification, specify the following:

```
set_app_var testbench_style testbench_type
```

For a list of testbench types that you can use, see the man page.

You can set constraints on various styles of testbenches. To enable this capability, use the `-style` option with the `set_constraint` command. The `-style` option can have the following values: `symbolic`, `binary`, `dataint`, `protocol`, `dualphs`, `clkenum`, `clkwave`, `portcov`, `cammatch`, `holdchk`, `rom`, `romhold`, or `romprot`.

For details about the various testbench styles, see the man page for the `testbench_style` variable.

For example, to set a constraint of logic 0 on the SE signal for just the binary testbench, use the following command:

```
set_constraint SE -style "binary" -set {1'b0}
```

The `set_constraint` command creates the following Verilog code within the `apply_global_constraints` tasks for all the testbenches generated:

```
if( (esp_testbench_style == "binary")) begin
        {SE} = 1'b0 ;
```

If you do not use the `set_constraint` command with the `-style` option, the constraint applies to all testbenches. For example, the following command sets the SE signal to logic 0 in all testbenches:

```
set_constraint SE -set {1'b0}
```

The `set_constraint` command automates testbench creation by translating Tcl commands into the Verilog code required for testbenches. You do not need to use the `set_constraint` command to set testbench constraints. Instead, you can manually create your own constraints by writing Verilog code. Your manual constraints should be in a file. To use this file in verification, set the `testbench_constraint_file` variable to the name of the file. You can also add your constraint code to a separate constraints file and include that file in your testbench.

Variables That Affect Testbench Styles

To control the type of testbenches you can generate, use the following variables:

- `testbench_binary_cycles`
- `testbench_constraint_file`
- `testbench_declaration_file`
- `testbench_dump_symbolic_waveform`
- `testbench_flush_cycles`
- `testbench_initialization_file`
- `testbench_symbolic_cycles`
- `testbench_implementation_instance`
- `testbench_module_name`
- `testbench_output_checks`

- `testbench_reference_instance`
- `testbench_style`

For more information, see the associated variable man page.

Configuring Testbench Pins

To configure the testbench pin attributes at the ESP shell prompt, use the `set_testbench_pin_attributes` command.

The syntax of the command is as follows:

```
set_testbench_pin_attributes
         tbpin
         [-checker Compare | Comparex | Comparez | Disable]
         [-function string]
         [-value Low | High]
         [portgroup list]
         [direction string]
         [-allow {0 | 1 | X | x | Z | z}]
```

where `tbpin` is the name of the port, `string` is a legal value, such as `clock`.

To specify the symbolic values `X` and `Z` in addition to `0` and `1` at the primary inputs of the design with the automatically generated testbenches, use the following command:

```
set_testbench_pin_attributes pin -allow [list]
```

The `-allow` option accepts a list with the following possible values:

- `0`
- `1`
- `X`
- `x`
- `Z`
- `z`

Note:
`X` and `Z` are not case-sensitive. The default is `{0 1}`.

For example, the following command stimulates the RESET pin with the logic values, zero and one, and the symbolic values, X and Z:

```
set_testbench_pin_attributes RESET -allow {0 1 x Z}
```

The `report_testbench_pins -all` command reports all the settings applied by the `set_testbench_pin_attributes` command as shown in Example 7-1:

Example 7-1 Reporting Symbolic Values With the report_testbench_pins -all Command

```
esp_shell> report_testbench_pins -all
**************************************
Report : report_testbench_pins
...
**************************************
-------------------------------------------------------------------------
PortName     Function     Checker      Value     Allow   Port Groups
-------------------------------------------------------------------------
OUT1         Other        Compare      None      N/A
OUT4         Other        Compare      None      N/A
IO1          Other        Compare      High      0,1,x,z
IO4          Other        Compare      High      0,1,x
IN1          Other        None         High      0,1,x
IN4          Other        None         High      0,1,z
ME           Other        None         High      0,1
OE           Other        None         High      0
*************************************************************************
```

Example 7-1 shows the port ME with the default values in the Allow column of the report, while the port IO1 shows all possibilities. For the output-only signals and ports, the Allow column always reports the value as `N/A`.

For more information, see the `set_testbench_pin_attributes` command man page.

Latch-Based Designs

The ESP shell provides additional symbolic testbench support for latch-based designs and designs that associate some signals with one phase of the clock and other signals with the other phase of the clock. To specify the signals whose values are to be changed before the appropriate clock edge, use the `set_input_delay` command.

The `4check` value of the `testbench_output_checks` application variable enables a consistent timing relationship between output checks for values that propagate from the opposite edges of a clock.

When you specify the following command, the generated testbench checks the outputs four times per cycle:

```
esp_shell> set_app_var testbench_output_checks 4check
```

The four checks occur at specific times and are described with respect to an active high clock in a symbolic testbench:

- `'SETUPTIME` after start (just before the rising edge of the clock)
- `'CHECKTIME` later (middle of the clock period)

- `'SETUPTIME` later (just before the falling edge of the clock)
- `'CHECKTIME` later (end of the testbench cycle)

Without the `4check` value, the tool performs only checks 1, 3, and 4 and they are numbered 1, 2 and 3. Therefore, the `4check` setting renames the checks 2 and 3 of the default `3check` setting to be the third and fourth checks of the `4check` setting respectively.

For example, consider that a default (`3check`) script had the following constraint:

```
esp_shell> set_constraint signal_name -ignore -check_num {2 3}
```

If you specify the `4check` value, you need to change it to the following:

```
esp_shell> set_constraint signal_name -ignore -check_num {3 4}
```

If you specify the `4check` value, the dual phase testbench (`dualphs` which has the `.2ph` suffix applies two independent symbols per cycle) will have four checks. Therefore, the tool observes the input changes that propagate to the outputs before the clock edges occur and checks them for equivalence. If you do not specify the `4check` value, the dual phase testbenches have only two checks, 1 and 2. With the `4check` value, these checks are renumbered 2 and 4 respectively.

For more information, see the `testbench_output_checks` and `testbench_style` application variable man pages, and the `set_constraint` command man page.

Output Display Radix

The ESP shell supports the octal and hexadecimal display radix. The default display radix of the ESP-created testbenches is binary. To change the display radix, use the `-radix` option of the `set_testbench_pin_attributes` command.

The supported radix values are `binary`, `bin`, `octal`, `oct`, `hexadecimal`, or `hex`. These values are case insensitive.

To display the signal AIN in hexadecimal and the signal DIN in octal respectively, use the following commands:

```
esp_shell> set_testbench_pin_attributes AIN -radix hex
esp_shell> set_testbench_pin_attributes DIN -radix oct
```

Note:

For octal and hexadecimal displays, if some of the bits (but not all of the bits) are symbolic, that digit will be S instead of s. Therefore, in the example, if the signal AIN is only three bits wide (AIN[2:0]) and even if all three bits are symbolic, it will be displayed as S because as a hexadecimal digit, the MSB is 0.

Configuring Port Groups

You can define and use port groups within the ESP shell. Port groups associate a set of ports with a common clock. Use port groups to define multiport memories and multiclock domains.

The port group commands are as follows:

- `set_portgroup`
- `remove_portgroup`
- `get_portgroups`
- `set_portgroup_constraint`

In addition, the `-portgroup` argument of the `set_testbench_pin_attributes` command performs port association.

For more information, see the `set_testbench_pin_attributes` command man page.

You can associate a pin with one or more port groups.

Defining Clock Objects

Unless your design is purely combinational, you must specify at least one clock for a testbench.

To define a clock object in the ESP shell, specify the following:

```
create_clock
         clock_pin_name
         -period time_units
         -setup setup_time
         -initial 0 | 1
         -delay delay_time
         -phase first_phase_width
```

Setting a Simulation Timescale

You can set a timescale value that the ESP shell inserts into the generated testbench files and SPICE Verilog model files. To generate a timescale value, use the `set_simulation_timescale` command.

To set the timescale value, specify the following:

```
set_simulation_timescale
time_unit time_measure precision time_measure
```

For *time_unit* or *precision*, specify a value of 1, 10, or 100. For *time_measure*, specify a value of s, ms, us, ns, ps, or fs.

For example,

```
set_simulation_timescale 1 ns 1 ps
```

The ESP shell supports delay values smaller than picoseconds for RC analysis in the SPICE simulators. To specify the RC delay unit, set the `verify_rcdelay_unit` application variable to one of the supported values: `ps` and `p` for picosecond, `fs` and `f` for femtosecond. The default is `ps`.

For example, the following command changes the RC delay unit to femtosecond:

```
esp_shell> set_app_var verify_rcdelay_unit fs
```

The delay rounding variable, `verify_delay_round_multiple`, specifies the rounding value as a multiple of the RC delay unit. The default is 10. If you specify the RC delay unit as femtosecond, the `verify_delay_round_multiple` variable rounds off the delay value to the nearest 10 femtoseconds.

For example, the following command sets the partial transition threshold to 70351 femtoseconds:

```
esp_shell> set_app_var verify_partial_transition_threshold 70.351
```

For more information, see the man pages for the `verify_partial_transition_threshold, verify_delay_round_multiple, and verify_rcdelay_unit` application variables.

Setting Port Group Pins

You can group testbench pins together functionally to associate ports with the appropriate clock for multiple-clock and multiple-port memories. Using this functionality, the ESP tool sets the time at which to assert and check a pin.

To specify and group testbench pins together functionally, use the following example:

```
set_portgroup {A B}
set_testbench_pin_attributes CLKA -function Clock –portgroup {A}
set_testbench_pin_attributes WE -function Write -value High –portgroup
{A}
```

Use the `-clock` option to define the clock name of the group followed by the pins in the clock group.

Setting a Port Group Constraint

When multiple port groups and multiple clocks are involved, you must specify the constraints required for overlapping clocks. The types of contention constraints are `readwrite`, `writewrite`, and `none`.

To specify constraining pins, use the following command in the ESP shell:

`set_portgroup_constraint` ***`contentiontype`***

Generating Automatic Testbenches

You can create specific testbenches that compare the implementation design against the reference design to determine whether they are functionally equivalent to each other. The testbench generator, accessed using the `write_testbench` command, takes the port information for both the Verilog reference model and the SPICE implementation and generates a testbench file.

The `write_testbench` command supports several testbench styles. Use this command to generate a testbench that drives symbolic simulation. Automatic testbenches support input and output constraints.

After completing port matching, you can write a set of testbenches. Set variables to control the type of automatic testbenches before generating them.

To generate a set of testbenches in the ESP shell, specify the following command:

`write_testbench` *`file_path`*

Set specific options when using the `write_testbench` command so that the resulting testbench contains the required information.

For example,

`write_testbench` *`file`*`.tb`

To control the type of testbenches you can generate, use the following variables:

`testbench_binary_cycles`	`testbench_constraint_file`
`testbench_dump_symbolic_waveform`	`testbench_flush_cycles`
`testbench_initialization_file`	`testbench_symbolic_cycles`
`testbench_implementation_instance`	`testbench_module_name`
`testbench_output_checks`	`testbench_reference_instance`

`testbench_style`

The `testbench_style` variable controls the actual style of testbench that is written. The following testbench types are allowed: `cammatch, clkenum, clkwave, dataint, dualphs, holdchk, macro, portcov, protocol, rom, romhold, romproto, sram, and symbolic`.

For more information, see the `write_testbench` command man page.

Customizing Your Testbench

The ESP tool generates testbenches during the normal verification flow. For many designs, these automatic testbenches require only minor modifications for a successful verification.

The ESP tool structures these testbenches so that the verification specialist can replace individual parts of the testbench without manually editing the generated testbench. The tool accomplishes this by providing the inclusion of four files in the generated testbench, as listed in the following:

- Setup file

  Use this file to specify all the testbench overrides. Use the `testbench_setup_file` variable to specify the name of this file.

- Declaration file

  Use this file to define registers, wires, functions, tasks, and any other module-level Verilog code to be added to the testbench.

- Initialization file

  Use this file to define the device under test (DUT) initialization sequence.

- Global constraints file

  Use this file to define complex global constraints on inputs that the `set_constraint` command cannot specify.

Each of these files has a specific purpose. The `ifdef...`else...`endif Verilog compiler directives wrap the individually replaceable sections of the testbench. The `ifdef` statement of each section references a macro. This macro is defined according to the following naming convention:

`REMOVE_DEFAULT_`*`name of section`*

The replaceable sections of the testbench follow the template shown in Example 7-2.

Example 7-2 Testbench Replaceable Section Format

```
`ifdef REMOVE_DEFAULT_section name
 `else
 // Verilog code for section name
 `endif
```

To replace the default conditions of the testbench with your design-specific conditions, you must define the appropriate macro in your setup file and add the replacement code to your declaration file.

For example, to replace the initialization (INIT) section of the testbench, you must define the REMOVE_DEFAULT_INIT macro by adding the following code to your setup file:

```
`define REMOVE_DEFAULT_INIT
```

Next, add your new initialization code, such as the code in Example 7-3, into the declaration file.

Example 7-3 Design-Specific Initialization Code

```
task INIT;
 begin
      // It is good practice to set inno_cycle
      inno_cycle = "INIT";
      // put your Verilog code here
 end
endtask
```

You can customize a testbench to accomplish the following:

- Define statements controlling clock cycles before a module statement
- Declare reg and wire signals for primary I/O signals
- Instantiate a reference model
- Instantiate an implementation model
- Control the error vector file name used in binary simulation
- Control the VCD output statements
- Control the symbolic coverage output statements
- Control the applied test vectors
- Control symbol masking within the apply_global_constraints tasks
- Control `inno` library file functions and tasks by placing `ifdef` statements around functions and tasks defined in the common library file auxx/esp/primitives/inno

The next sections describe how you can customize testbench tasks and check for ignored outputs.

For more information about modifying automatic testbenches, see the “Testbench Customization” section in the ESP Help.

This section includes the following topics:

- Controlling Testbench Tasks
- Creating Output Checker Specifications
- Reporting Always Ignored Outputs

Controlling Testbench Tasks

The ESP tool allows you to modify an automatically generated testbench by replacing an existing task with a new task while keeping the same task name.

The tool qualifies every task within the testbench it generates with an `ifdef` statement so that the tasks you define replace the tasks at compile time. For a given task named *`taskname`*, if the identifier `REMOVE_DEFAULT_`*`taskname`* is defined, the task generated by the ESP tool is not used. If you do not define the `REMOVE_DEFAULT_`*`taskname`* identifier, the ESP tool uses the task that it generates.

For example, the output checker task for a signal named DOUT would be named check_DOUT and the code in Example 7-4 would be produced.

Example 7-4 Code Produced When the Output Checker Task Names a Signal as check_DOUT

```
`ifdef REMOVE_DEFAULT_check_DOUT
`else
task check_DOUT;
  begin
    $display("Checking DOUT                                          ", $time);
    $display("     ref  = %b\n     xtor = %b\n", DOUT_ref, DOUT_xtor);
    `ifdef _ESP_
      $esp_context("DOUT");
    `endif
    inno.compare(64, DOUT_ref, DOUT_xtor);
  end
endtask // check_DOUT
`endif
```

If you want to replace the task definition with your own, you can define the `REMOVE_DEFAULT_check_DOUT` identifier and provide your own definition in a declaration file that has the contents shown in Example 7-5:

Example 7-5 Using Task Identifiers

```
`define REMOVE_DEFAULT_check_DOUT
  task check_DOUT;
```

```
    begin
      $display("Not Checking DOUT  ", $time);
    end
  endtask // check_DOUT
```

If this file is named `myDeclarationFile`, the `set_app_var` *`testbench_declaration_file`* `myDeclarationFile` command inserts the file into the testbench just after the symbol declarations, and its task definition is used because the tool has not yet compiled the default definition.

You include the code for this in the declaration file whose name is the value of the `testbench_declaration_file` variable. This causes the ESP tool to insert the declaration file into the testbench just after you declare the symbols and just before you define the output checker tasks.

Creating Output Checker Specifications

The ESP tool generates counter examples when simulation results do not match the expected data. The `$esp_error` Verilog system task allows the tool to report errors or counter examples. Include the `$esp_error` system task in your Verilog code to allow the ESP tool to report errors or counter examples when a signal from the design does not match the expected value.

In simulate mode, the tool calls the `$esp_error` system task only when you use it to generate a counter example. If you do not use the `$esp_error` system task, the tool cannot report a test failure. You must use the `$esp_error` system task so that the tool can provide a binary error trace.

The format of a checker is shown in the following example.

Note:

The `$esp_context` system task requires a valid port name. Do not include the bit number associated with the port name.

Example 7-6 System Tasks Required to Record a Simulation

```
task check_output_1
  begin
    `ifdef _ESP_
    $esp_context("output_1");
    `endif
    if (output_1_ref != output_1_xtor)
      `ifdef _ESP_
      $esp_error("ERROR on output_1 signal",0);
      `else
      $display($time," : ERROR in output_1 signal");
      `endif
  end
endtask // check_output_1
```

The example shows that you can conditionally include the functions required for the ESP tool within a generic testbench so that the same testbench can be used within a simulator other than ESP, assuming that you are only using Verilog constructs.

Reporting Always Ignored Outputs

To help you find problems such as having a reference pin stuck at X throughout the entire simulation, the automated testbenches check for ignored outputs if you define the outputs using the `comparex` or `comparexz` functions. The default output compare function is `comparex`.

- The `comparex` function ignores the implementation output value if the reference output is X.
- The `comparexz` function ignores the implementation output value if the reference output is X or Z.

To ensure that the checkers are working correctly, the automated testbenches have Verilog code in the output checkers to ensure that the tool does not ignore the implementation output value for all the times that it is checked. For the `comparex` function, the tool checks at least one time when the reference output is not X; for the `comparexz` function, the tool checks at least one time when the reference output is not X or Z. Otherwise, the tool issues an error message similar to the following, and the equivalence check fails:

```
ERROR: Check bits of DOUT that are always ignored:00001000
```

This example indicates that the tool always ignored DOUT[3] in the implementation design because DOUT[3] in the reference design is always X (if the default `comparex` function is used) or always either X or Z if the `comparexz` output checking function is used.

This feature does not affect other compare functions that do not conditionally ignore the output check.

Important:

A side effect of this feature is that if you accidentally skip checking an output by conditionally calling the output checker task, the tool issues an error message at the end of simulation indicating that the output checking has been ignored (or in this case, it was not checked).

The tool implements the check by defining a register in the `alwaysIgnored_`*`signal_name`* testbench for each of the output and inout signals at the top-level of the reference design. These registers start with a value of 1 for each bit of the output signal. As the simulation progresses, if a value other than X can appear for that bit of the output during one of the compare operations, the corresponding bit of the `alwaysIgnored_`*`signal_name`* vector is set to 0. Thus, bits within this vector that are still 1 at the end of the simulation indicate that those bits of the corresponding reference output were always X at the compare times and that the implementation values were always ignored. To override this check, set the `alwaysIgnored_`*`signal_name`* bits to 0.

Consider the following error:

```
ERROR: Check bits of DOUT that are always ignored:00001000
```

To override the error, specify the following constraint, which sets bit 3 to 0:

```
set_constraint -set 0 {alwaysIgnored_DOUT[3]}
```

If you want to set all DOUT bits to 0, use the following command:

```
set_constraint -set 0 {alwaysIgnored_DOUT}
```

Example 7-7 shows the testbench code changes that enable the tool to check if the signal DOUT is always ignored.

Example 7-7 Testbench Code to Check for Ignored Outputs

```
// Declare all Input/Output signals
  reg [7:0] alwaysIgnored_DOUT = 8'b11111111;
  task check_DOUT;
    integer i;
    begin
      $display("Checking DOUT                                          ", $time);
      $display("     ref  = %b\n     xtor = %b\n", DOUT_ref, DOUT_xtor);
      `ifdef _ESP_
      $esp_context("DOUT");
      `endif

      for (i = 7; i >= 0; i=i-1) begin
        `ifdef _ESP_
        if ($esp_const_one(DOUT_ref[i] === 1'bx));
        else
        `else
        if (DOUT_ref[i] !== 1'bx)
        `endif
          alwaysIgnored_DOUT[i] = 0;
      end

      inno.comparex(8, DOUT_ref, DOUT_xtor);
    end
  endtask // check_DOUT

  initial begin
    INIT;
      .
      .
      .
    #1;

    if (alwaysIgnored_DOUT !== 0) begin
      $display("ERROR: Check bits of DOUT that are always
ignored:%b",alwaysIgnored_DOUT);
      `ifdef _ESP_
      $esp_error("ERROR: Some bits of DOUT always Ignored");
```

```
      `endif
    end
    $finish(2);
  end
```

You can disable this feature by using a different compare function, disabling compare on the output altogether, or modifying the signal for each output port that tracks whether the port is checked. Because the `alwaysIgnored_`*`output_name`* register is only updated and checked for ports that use the conditional output checks, the use of this constraint technique depends on the testbench pin output checker value.

To disable reporting on a specific bit that the ESP tool always ignores at an output port, set that bit to 0 within the apply_global_constraints task. For example, to disable checking the bits 0 and 4 of an 8-bit signal called siga, add the code shown in Example 7-8 or Example 7-9 to your global constraints.

Example 7-8 Disable Bit Checking Using Two Lines of Code

```
alwaysIgnored_siga[0] = 0;
alwaysIgnored_siga[4] = 4;
```

Example 7-9 Disable Bit Checking Using One Line of Code

```
alwaysIgnored_siga[0:7] = alwaysIgnored_siga[0:7] & 8'b01110111;
```

8

Verifying Your Design

This chapter describes verification for the compare flow, the default ESP flow. For more information about the power intent verification flow and the simulation-only flow, read Chapter 11, “Advanced ESP Flows”.

After preparing the verification environment and setting up the design, you are ready to verify your design. The following sections describe how to verify one design against another and how to generate reports:

- Verifying Functional Equivalence
- Running Testbenches in Parallel
- Enabling Automated Effort Selection
- Including Modified Testbenches in Your Verification
- Detecting Multiple Verification Failures
- Advanced Redundancy Checking Capabilities
- Reporting and Interpreting Results

Verifying Functional Equivalence

To verify the functional equivalence of the reference and implementation designs in the ESP shell, specify the following command:

```
verify
```

Running Testbenches in Parallel

The ESP shell permits multiple testbenches to run in parallel. You can run the testbenches on the current machine or across multiple machines, depending upon the settings of three ESP shell variables.

The following variables affect the operation of the distributed testbench feature:

- `verify_max_active_testbenches`

  To trigger parallel testbench execution, set this variable to a value greater than 1.

- `verify_use_distributed_processing`

  To use the host file that the `verify_distributed_processing_hosts_file` variable specifies, set this variable to `on`. If you disable the `verify_use_distributed_processing` variable and the `verify_max_active_testbenches` variable is greater than 1, the ESP tool runs testbenches in parallel on the current machine using multiple cores, if available.

- `verify_distributed_processing_hosts_file`

  To specify the name of the host file, use this variable. The host file configures the specified hosts and execution methods that the tool uses for distributed execution of testbenches.

The ESP tool supports distributed execution using the Remote Shell (RSH), the Secure Shell (SSH), the Platform Computing Load Sharing Facility (LSF), and the Open Grid Scheduler or Grid Engine (OGS or GE).

For more information, see the `add_distributed_processors` command and `distributed_testbench` flow man pages.

Enabling Automated Effort Selection

You can enable the ESP tool to run an equivalence test that starts with the lowest effort level and automatically progresses to higher levels as needed.

For example:

```
...
 set_app_var verify_auto_effort_selection true
 verify
...
```

When you enable the `verify_auto_effort_selection` variable, the tool automatically selects the lowest effort level possible to show equivalence between the Verilog and SPICE designs, resulting in faster verification. If the lower effort levels result in error vectors, the effort level is increased. Because the selection process has some overhead, designs that normally take longer than 30 minutes see the largest benefit. In some cases, the ESP tool determines that only the highest effort level is sufficient and the runtime might increase by about 10 percent. Binary debug simulation (`debug_design`) always runs in the high effort mode because symbolic runs that generate counter examples always terminate at the highest effort level.

Including Modified Testbenches in Your Verification

If you have customized your testbench, you must point the tool to the appropriate testbenches to include them in the verification run.

To add custom testbenches to the verification run in the ESP shell, specify the following command:

`set_testbench` *`files`*

where `files` is the path of the file that contains the modified testbenches that you want to verify.

To remove a testbench, use the `remove_testbench` command.

Detecting Multiple Verification Failures

Ideally, when you run verification, you do not have any failures, but often you need to debug multiple failures before the design passes verification. If you want the ESP tool to detect multiple failures per verification run, set the `verify_max_number_error_vectors` variable to the number of error vectors you want to detect in a single run. By default, the tool detects one failure per verification run. You need to set this variable before you run verification.

Note:

When you use the `verify_max_number_error_vectors` variable, you might see a runtime and memory penalty because of the additional processing overhead.

This feature detects multiple *independent* failures. For example, the tool cannot detect a failure that an earlier failure conceals. To see all the error vectors that the tool detects during the verification run, use the `report_error_vectors` command. For details, see “Reporting Multiple Errors” on page 8-7.

Advanced Redundancy Checking Capabilities

In redundancy verification, you intentionally add spare cells to your design to replace any defective cells, improving yield on arrayed blocks such as memories. The process of cell insertion is based on control values determined post silicon testing. These control values are programmed using fuses external to the memory block and look like control inputs to the block being verified. ESP verifies that this redundancy logic performs correctly by symbolically selecting a row, column, or bank to repair and shows that it is functionally equivalent to a nonrepaired memory. From the outside view, the repaired memory and the original memory are identical.

The advanced redundancy verification features allows you to check the implementation design to ensure that some allowed value of the redundancy controls compensate memory core faults, whether the Verilog reference model includes redundancy modeling or not. The tool determines if there is an assignment to the redundancy controls that makes the designs equivalent, even with symbolic injected errors.

Table 8-1 Advanced Redundancy Verification Commands

Commands
`set_stuck_fault_group`
`set_net_group`
`set_symbol_to_pass`
`report_net_groups`

Table 8-1 Advanced Redundancy Verification Commands(Continued)

Commands
`report_stuck_fault_groups`
`set_constraint` (`-symbolic_static` option)

The `set_stuck_fault_group` command specifies the number of symbolic faults that can occur in a group of nets. The `set_net_group` command defines a group of nets that are assigned symbolic faults. This group of nets form the bitlines of the design for a column redundancy scheme. The `set_symbol_to_pass` command defines symbols that search for values that allow the simulation to pass. The tool produces an error vector only if it does not find values that allow the simulation to pass. After bad rows or columns are determined, the column and row redundancy controls must not change. The `-symbolic_static` option of the `set_constraint` command allows the column and row redundancy controls to be symbolic, but the control values do not change during the simulation.

Note:

The `set_symbol_to_pass` command requires only one ESP-CV license. The `set_stuck_fault_group` command requires two ESP-CV licenses during an ESP session.

The Tcl script in Example 8-1 shows how to use the redundancy verification commands for an SRAM with column redundancy where REDUND is the redundancy enable signal and RCA is the redundant column address.

Example 8-1 Script Using Advanced Redundancy

```
read_verilog -r ram_redund.v
read_spice -i ram_redund.sp      match
set_testbench_pin_attributes -func clock CLK
set_constraint -symbolic_static REDUND
set_constraint -symbolic_static RCA
set_net_group -i -net {RBL*} read_bitlines
set_stuck_fault_group -i -number 1 -net_group read_bitlines
set_symbol_to_pass RCA
set_symbol_to_pass REDUND
verify
report_log
quit
```

Use the `report_symbols_to_pass` command to get at least one possible value of the `set_symbol_to_pass` command signals that lets your verification complete without error.

Syntax:

```
report_symbols_to_pass -tb tbid -lines N
```

The `-tb` option reports symbols for the specified testbench. This is a required option.

The `-lines` option reports only `N` faults (smaller report). The default is all faults.

For example, if you use the `report_symbols_to_pass -tb tb0` command in your script, you get an output similar to the following:

```
...
 inno_tb_top.REDUND_sym |inno_tb_top.REDUND_sym1
      ||inno_tb_top.REDUND_sym2
      |||inno_tb_top.REDUND_sym3
      ||||inno_tb_top.RCA_sym[0:2]
      |||||  inno_tb_top.RCA_sym1[0:2]
      |||||  |  inno_tb_top.RCA_sym2[0:2]
      |||||  |  |  inno_tb_top.RCA_sym3[0:2]
      |||||  |  |  |
      |||||  |  |  |
      |||||  |  |  |
      1---110---------        inno_tb_top.xtor.RBL_3_
      1---001---------        inno_tb_top.xtor.RBL_4_
      1---101---------        inno_tb_top.xtor.RBL_5_
      1---011---------        inno_tb_top.xtor.RBL_6_
      1---111---------        inno_tb_top.xtor.RBL_7_
      0---------------        inno_tb_top.xtor.RBL_8_
      1---000---------        inno_tb_top.xtor.RBL_0_
      1---100---------        inno_tb_top.xtor.RBL_1_
      1---010---------        inno_tb_top.xtor.RBL_2_
      1---010---------        inno_tb_top.xtor.RBL_2_
```

For more information, see the man pages.

Reporting and Interpreting Results

The following sections describe various reports you can use to debug verification failures:

- Reporting Multiple Errors
- Viewing the Report Log
- Reporting Status
- Error Reports
- Reporting Verification Progress
- Reporting Coverage
- Reporting Symbolic Coverage

Reporting Multiple Errors

If you have enabled the tool to detect multiple error vectors per verification run, you can use the `report_error_vectors` command to see all the error vectors. This command works with the `verify_max_number_error_vectors` variable and reports the detected errors. Use this report to identify error vectors for debugging purposes. For details, see "Detecting Multiple Verification Failures," "Debugging Using Traditional Verilog Techniques," and the respective man pages.

Viewing the Report Log

You can generate a report log that provides information about the most recent simulation results of the current design. The report log shows the warning and error messages produced during the verification of the designs.

To generate a log report in the ESP shell, use the `report_log` command.

To save a report to a file, do the following:

1. Click the Save icon.
   The "Save Report to a File" dialog box appears.
2. In the "File Name" box, enter a name for the report and click Save.

Reporting Status

You can generate a report that summarizes your verification run. The report includes error conditions that occurred while checking the simulation log files. The report also includes checks done on oscillation, randomization, unexpected completion of simulation, and counter examples produced.

To generate a status report, use the `report_status` command.

If you use a generated testbench, the `report_status` command reports the status on all output signals. The `report_status` command provides information about the passing, failing, and terminated points for the testbenches used.

To save a report to a file, do the following:

1. Click the Save icon. The "Save Report to a File" dialog box appears.
2. In the "File Name" box, enter a name for the report and then click Save.

Error Reports

The `report_status` command reports the overall status of the simulation that the checkers determine within your custom testbench. The report also notes if a symbol is “randomized” or if an oscillation has occurred in the Verilog netlist. If these occur, the tool reports that the simulation results are inconclusive.

Note:

In simulate mode, this command does not report the number of passing or failing points; in compare mode, the command reports the number of passing or failing points.

The `report_status` command reports the following status values:

- INCONCLUSIVE (Abnormality) - A simulation compile or runtime error occurred. Runtime errors are randomizations of symbols, oscillations of nets, and failures of the simulation to complete properly.
- FAILED - The simulation failed with a simulation signal that does not match an expected value.
- SUCCEEDED - The testbench passed with no errors and no abnormal conditions.
- Not Run (matched) - All the matching has been successful, but the simulation has not been initiated yet.

Reporting Verification Progress

In the ESP shell environment, information is displayed to the standard output. This information is updated as the verification proceeds. From the transcript, you can see the design that is processed and observe the results of the verification.

During verification, the ESP shell assigns a status message for each compare point it identifies:

Pass

A compare point match passes verification. Passing verification means that the ESP shell determined that the functions that define the values of the two port design objects are functionally equivalent.

Fail

A compare point match does not pass verification. Verification fails when the ESP shell has determined that the two design objects that constitute the compare points are not functionally equivalent.

Abort

Verification stops. This occurs when the ESP shell reaches a user-defined failing limit. For example, the ESP tool normally stops verification after it finds 20 failing points. The unverified points are labeled aborted. In addition, an aborted point represents a compare point that the ESP shell did not assign as either passing or failing.

Aborted points occur when you interrupt the ESP shell during verification:

- When a wall clock time limit is exceeded
- When a compare point is part of a combinational loop that the ESP shell cannot break automatically
- When two design objects are too difficult to verify

Pend

Compare point match is pending verification.

Based on the preceding categories, the ESP shell classifies final verification results in one of the following ways:

Succeeded

The implementation design is functionally equivalent to the reference design. All compare points passed verification.

Failed

The implementation design is not functionally equivalent to the reference design. The ESP shell could not successfully match all design objects in the reference design with comparable objects in the implementation design, or at least one design object in the reference design was determined to be nonequivalent to its comparable object in the implementation design (a compare point failure).

If you interrupt verification by pressing Control-c or with a user-defined time-out (such as the CPU time limit), and the tool detects at least one failing point before the interruption, the tool still reports a verification failure.

Inconclusive

The ESP shell could not determine whether the reference and implementation designs were equivalent. This situation occurs in the following cases:

- A matched pair of compare points was too difficult to verify, causing an aborted compare point. The tool did not find any failing points elsewhere in the design.
- You interrupted verification either by pressing Control-c or meeting a user-defined time-out, and the tool did not detect any failing compare points before the interruption.

Incomplete

You stopped verification (for example, because of a time-out or a Control-c), but it ran long enough to produce some results.

Interrupted

You stopped verification before the ESP shell could report useful information. You cannot run a diagnosis or report on incomplete results.

If a verification is inconclusive because it you interrupted it, partial verification results might still be available. You can create reports on the partial verification results.

Reporting Coverage

When you enable coverage, the tool reports a summation of simulation results over the time of simulation. The report lists areas not covered by verification. Use the report to ensure that your testbenches do not lack significantly in their ability to verify the design.

To enable coverage so that you can report and examine the results of your verification run, specify the following command in the ESP shell:

```
set_app_var coverage true
```

To obtain coverage information in the ESP shell, use one of the following commands:

```
report_coverage -all
```

OR

```
report_coverage -testbench testbench_path
```

In the `report_coverage` command, the `-all` option merges the coverage information for all testbenches that were run and reports the merged coverage. However, if you want to report the coverage metrics for a specific testbench, use the `report_coverage` command with the `-testbench` *`testbench_path`* option. The `testbench_path` value might not be obvious, so use the `report_status` command to confirm the correct path.

To use a coverage specification file that you have created (containing filter commands), add the `-spec` *`specfile_path`* option to the `report_coverage` command as follows:

```
esp_shell (verify)> report_coverage -testbench \
ESP_WORK/ram1r1/esp.tb.sym -spec my_filters
```

For details on code coverage, see the “Code Coverage” sections in the ESP Help.

For more information about coverage filters, see the `coverage_filters` flow man page.

Reporting Symbolic Coverage

The `coverage` variable controls the automatic output of symbolic coverage data. If you disable this variable, the tool does not output any coverage data.

The default coverage output file is called esp.cov. It is used as the full path to the symbolic coverage file where coverage data is displayed. If you define the `coverage_db_file` environment variable, the tool uses the variable in place of esp.cov.

The coverage data is only saved if the testbench passes simulation. The `report_coverage` command reports symbolic coverage information gathered during the symbolic simulation of the design. By default, only coverage data for the tested design is recorded.

Example 8-2 Example of report_coverage Command Output

```
**************************************
Report : report_coverag
Design : ram1r1w
Version: K-2015.12-SP1
Date   : Thu May  5 03:12:02 2016
**************************************
 ESP Coverage Database Version 3.0 read on May 5, 2016 03:12:02
**********************************
inno_tb_top.ref (ram1r1w)
Symbolic Net Coverage: 48.28%
Non-Symbolic Nets: 15 / Total Nets 29
Propagated Symbol Coverage:     0%
Propagated Symbols: 0 propagated / 18 symbols
Dropped Symbols: 18 dropped / 18 symbols
Toggle Coverage: 55.17%
Untoggled Nets: 13 / Total Nets 29
**********************************
```

Example 8-3 shows the standard log output file generated in simulate-only mode.

Example 8-3 Standard Output

```
6 clsrvxx > esp_shell -f cmds | tee log

                              ESP (R)
                 Version Z-2006.12-SP1 -- Mar 21, 2007
               Copyright (c) 1998-2007 by Synopsys, Inc.
                          ALL RIGHTS RESERVED

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

Hostname: clsrvxx (linux)

set_verify_mode simulate
Information: Cleared container(s) ref and imp. (ESPUI-050)
```

```
1
read_verilog {tb_good.v test.v}
Information: Analyzing source file "tb_good.v" .... (ESPVER-001)
Information: Analyzing module ( inno_tb_top ). (ESPVER-004)
Information: Analyzing source file "test.v" .... (ESPVER-001)
Information: Analyzing module ( blk ). (ESPVER-004)
Information: 0 parse error(s) and  0 warning(s). (ESPVER-025)
Information: Config: Extracted  module inno_tb_top from
reference
design
(ESPCFG-092)
1
set_top_design blk
1
verify

Verify testbench tb0 ......
Information: Verification succeeded. (ESPUI-033)
1
report_status
**************************************
Report : report_status
Design : blk
Version: Z-2006.12-SP1
Date   : Wed Mar 21 10:58:19 2007
**************************************

Reference Design:      r:/WORK/blk

Verification:
  Status:        SUCCEEDED

  tb0: pend   *NOPATHSPECIFIED*

*NOPATHSPECIFIED*
quit
Maximum memory usage for this session: 32 MB
CPU usage for this session: 0.22 seconds

Thank you for using ESP (R)!
7 clsrvxx >
```

For details on symbolic code coverage, see the “Code Coverage” sections in the ESP Help.

9

Debugging Your Design

This chapter describes verification debugging techniques. For additional debugging information for the simulation-only flow, see Advanced ESP Flows.

Only those designs that fail verification are debugged. A verification failure is a difference between your reference and implementation designs. By examining the details of the simulation for both designs, you can pinpoint the problem areas.

The ESP shell enables you to do the following:

- Create a simulation output file in the following formats:
  - VCD - standard Verilog format
  - FSDB - Verdi binary format
- Invoke a waveform viewing tool.
- Start an interactive signal trace.

Using the ESP tool to generate a waveform from problem vectors can help you correct the behavioral model or SPICE netlist. As an alternative, you can refine your constraints to account for the design differences.

The following sections describe methods to debug your design:

- Debugging Verification Mismatches
- Debugging Using Traditional Verilog Techniques

- Debugging With Interactive Signal Tracing
- Debugging Passed Designs at Will
- Displaying Symbolic Equation Data
- Stopping Simulation if Oscillations Occur
- Stopping Randomization by Default

Debugging Verification Mismatches

The ESP tool debugs most verification failures using binary simulation mode. After locating failures by running verification, analyze them by using one of the following:

- Traditional Verilog debugging techniques

  Traditional Verilog techniques use a waveform viewer. See "Debugging Using Traditional Verilog Techniques."

- Interactive Signal Tracing (IST) mode

  This mode is unique to the ESP shell. It allows you to interactively explore the SPICE parts of a design. See "Debugging With Interactive Signal Tracing."

- Debug using HSPICE simulators

  This mode runs SPICE simulations on failing testbenches. See "Debugging Violations With HSPICE."

Debugging Using Traditional Verilog Techniques

If you use the `verify_max_number_error_vectors` command, you might have multiple error vectors to choose from. To find out a specific error vector name and number, use the output of the `report_error_vectors` command.

This section describes debugging techniques in the following sections:

- Debugging Violations With HSPICE
- Outputting Waveform Data
- Waveform Output Formats
- VCD Output File Support ($dumpclose)

For more information, see "Detecting Multiple Verification Failures and the `debug_design` and `verify_max_number_error_vectors` command man pages.

To debug using traditional Verilog techniques,

1. Create a waveform file by running the following command:

```
debug_design
-dumpscope file_name
-tbid testbench_id
-vector_num error_vector_num
-testbench testbench_path
```

2. Select a format for the waveform output file:

```
set waveform_format vcd | fsdb
```

3. Run the `debug_design -symbolic_binary` command if the output of the `debug_design` command does not result in any output mismatches (0 errors). This option helps you debug race conditions.

4. View a waveform by running a binary simulation and write the resulting waveform to a Verilog VCD file. Use a waveform viewer of your choice to view the resultant ASCII contents in the VCD file.

5. Specify the following command to set the waveform viewer path and options used to invoke the viewer:

```
set_app_var waveform_viewer string
```

   This sets the `waveform_viewer` variable to control the waveform viewers.

   Note:

   The default of the `waveform_viewer` variable is as follows:

```
vfast %v; verdi -ssy -ssv -ssz -ssf %v.fsdb %s
```

6. To start the preferred waveform debugging tool, use the `start_waveform_viewer` command.

Debugging Violations With HSPICE

The `debug_design` command debugs Verilog with binary vectors. Use the `start_explore` command to perform interactive signal tracing in the SPICE design. The detailed violation report uses the `report_inspector_results -id xxxx` command and lists the exact command to use for each of these debugging methods.

To debug a violation in HSPICE, use the `write_spice_debug_testbench` and `start_spice_simulator` commands. The `write_spice_debug_testbench` command creates a SPICE deck to run in HSPICE or other SPICE simulator. The `start_spice_simulator` command runs the configured SPICE simulator.

For example, consider the following debug command:

```
debug_design -tbid tb0 -vector_num 25 -type pwr_short
```

Use the following command to create a SPICE input file for verification with HSPICE:

```
write_spice_debug_testbench -tbid tb0 -vector_num 25 -type pwr_short
```

Follow with the `start_spice_simulator` command which starts the SPICE simulator. Specify the SPICE simulator with the `spice_simulator` variable. For more information, see the man pages.

Outputting Waveform Data

The `waveform_dump_control` variable controls automatic output of waveform data. In the compare mode, the default is `on` and the Tcl shell controls the output of waveform values. However, in the simulate mode, you usually control waveform output in your custom testbench, so the default is `off`. If you want automatic waveform data output, enable the `waveform_dump_control` variable and use the following guidelines:

- Start the output file name with the string `dump`. If you define the `waveform_dumpfile` variable, the string contained in that variable is used as the start of the output file name.
- Start waveform output at the top module name. The `testbench_module_name` variable contains the name of the top-level module. By default, the variable value is `inno_tb_top`.
- Use the `waveform_format` variable to control the waveform file output format. The options are `vcd` or `fsdb`.
- Waveform output only occurs within the `debug_design` command.

You can also control waveform output by using the `debug_design -dumpscope` command. The `-dumpscope` option specifies the file to control waveform data generation.

Waveform Output Formats

The ESP tool supports VCD and FSDB waveform output formats. For details on the FSDB tasks, see the “Frequently Asked Questions > Verilog” section in ESP Help. For an overview of ESP task support, see “System Tasks.”

VCD Output File Support ($dumpclose)

The ESP tool implements a VCD system task called `$dumpclose` to control the VCD output file sizes.

Usage: `$dumpclose ;`

The `$dumpclose` statement turns off VCD output for the rest of the simulation. This allows you to do selective output of signals. This statement is specific to ESP and is not in the *Verilog Language Reference Manual*.

Debugging With Interactive Signal Tracing

Interactive signal tracing is used to debug verification errors by back-tracing RC nets in a SPICE model and analyzing RC nodes in your SPICE model. You cannot use this tool to back-trace nets in your Verilog design.

This section describes interactive signal tracing in the following sections:

Features

Interactive Signal Tracing is an interactive debugging tool used in the SPICE environment that enables you to

- Query the cause of any transition in the implementation design
- Back-trace causal paths in the implementation design
- Back-trace sources of X-values in the implementation design

Uses

Interactive signal tracing enables you to debug the following problems:

- Mismatches between the implementation and reference designs
- SPICE outputs all Zs
- Incorrect X-value on a SPICE output

Commands

Interactive signal tracing supports the following commands:

- `start_explore`

  Enables interactive signal tracing. Use at the `esp_shell (verify)` prompt.

- `stop_explore`

  Ends your interactive signal tracing session.

- `is_net_explorable`

  Checks to see if the net is explorable.

- `print_net_backtrace`

  Shows the series of events that led up to a transition in your SPICE netlist.

- `print_net_trace`

  Shows the result of input net changes on your SPICE implementation.

- `get_net_transitions`

  Shows transitions on a SPICE net in a textual waveform output.

- `get_net_top_level_name`

  Enables you to create a collection containing the top-level hierarchical name of a net.

See “Determining Transition Dependency” for command details.

For more information about the commands, see the man pages.

Determining Transition Dependency

Interactive Signal Tracing (IST) capability allows you to determine if a node depends on another node for a transition at the time specified, after executing the `start_explore` command. To enable IST, use the following command:

`print_net_dependency` *`net_name check_net_name time`*

- `net_name`: A net that is making a transition at `time`.
- `check_net_name`: A separate node whose transition might depend on the *`net_name`*.
- `time`: Time of the *`net_name`* transition in resolution units. For the default timescale of “1 ns 1 ps” this is ps, so enter a simulation time of 230 as 230000.

For example, if the transition on DOUT[7] at 230 ns depends on XI6.XOUT7.NS, the command

```
print_net_dependency inno_tb_top.xtor.DOUT[7] \
                     inno_tb_top.xtor.XI6.XOUT7.NS 230000
```

produces the following output:

```
Checking whether inno_tb_top.xtor.DOUT[7] is dependent on
                      inno_tb_top.xtor.XI6.XOUT7.NS
MATCH! inno_tb_top.xtor.DOUT[7] is dependent on
           inno_tb_top.xtor.XI6.XOUT7.NS
The Trace is:
inno_tb_top.xtor.XI6.XOUT7.NS
inno_tb_top.xtor.DOUT[7]
```

If the transition does not depend on XI6.XOUT7.NS, the command

```
print_net_dependency inno_tb_top.xtor.DOUT[7] \
                         inno_tb_top.xtor.XI6.XOUT7.NS 230000
```

produces the following output:

```
Checking whether inno_tb_top.xtor.DOUT[7] is dependent on
 inno_tb_top.xtor.XI6.XOUT7.NS
inno_tb_top.xtor.DOUT[7] is not dependent on
 inno_tb_top.xtor.XI6.XOUT7.NS
```

The next section describes how to use interactive signal tracing commands in an interactive signal tracing debugging methodology. Following the methodology, a design debugging example is presented that uses these commands to debug a verification mismatch.

See “Interactive Signal Tracing Example 1 - ram1r1w SPICE or Verilog Mismatch” on page 9-9.

Methodology

Use the following steps to determine the cause of an incorrect SPICE output:

1. To enable interactive signal tracing, execute the `start_explore` command from the `esp_shell (verify)` prompt.
2. From your verification failure report log, determine the SPICE output net you need to debug.
3. Execute the `is_net_explorable` command to determine if you can use interactive signal tracing commands on the net. You can only use interactive signal tracing commands on `rcmode` nets. The default mode for SPICE nets is `rcmode`.
4. Execute the `print_net_backtrace` command and look for any unexpected transition. Trace back several levels until you find something questionable.
5. Execute the `print_net_trace` command on one of the nets of the causal chain that causes suspicious transitions. The generated report shows you how and why the net transitioned throughout the run. From this report, you can review the current value on the net, the triggering input change, and the path's resistance, capacitance, and load values.
6. If you still have not found the root cause of the difference in the SPICE output, do the following tasks:
   - Execute the `debug_design` command to create a VCD file for debugging purposes. This command reruns the last failing testbench in binary mode for use with a waveform viewer.
   - Execute the `start_waveform_viewer` command and look for any unexpected behavior.
   - Execute the `print_net_trace` command on questionable nets.
7. To end your interactive signal tracing session, execute the `stop_explore` command.

Interactive Signal Tracing Example 1 - ram1r1w SPICE or Verilog Mismatch

This example uses the verification mismatch in the ramr1w1 design to illustrate the interactive signal tracing (IST) debugging methodology.

Models and Schematics

For the ram1r1w design schematics, see "Interactive Signal Tracing Example 2 - ram1r1w Schematics" on page 9-14.

Debugging Strategy

When you run interactive signal tracing, you have already read in your reference and implementation designs, matched ports, configured your testbench, and executed the `verify` command. For the ram1r1w design, your report log, shown in Example 9-1, indicates an error at time 230. The report shows that your reference design correctly transitioned to 00000000 when CLR went high but your implementation design remained at 10000000, failing to match the reference design's transition.

Example 9-1 ram1r1w Design Report Log

```
 ============ Input Signal Values ==============                 200
       SYMCYCLE1
                WE = 0
               CLR = 1
                OE = 0
                RE = 0
                 A = 000
                DI = 00000000
================================================
Checking DOUT                                                     200
     ref  = 10000000
     xtor = 00000000

Checking DOUT                                                     220
     ref  = 10000000
     xtor = 10000000


Checking DOUT                                                     230
     ref  = 00000000
     xtor = 10000000

At time 230000 : ERROR in above signal (ref != x)
```

To use interactive signal tracing (IST) to debug this mismatch, you first enable IST by executing the `start_explore` command from the `esp_shell (verify)` prompt.

Note:

You can only use interactive signal tracing commands to do net traces and node analysis on your SPICE model `rcmode` nets; you cannot use interactive signal tracing to analyze your Verilog model.

In the ram1r1w example, the bit DOUT[7] does not transition correctly at time 230. Using the interactive signal tracing `print_net_backtrace` command, you can trace the DOUT[7] net back to the primary input to check for unexpected transitions.

Ensure that DOUT[7] is explorable, as follows:

```
esp_shell(explore)
esp_shell(verify)> start_explore
esp_shell(explore)> is_net_explorable inno_tb_top.xtor.DOUT[7]
1
esp_shell(explore)> print_net_backtrace inno_tb_top.xtor.DOUT[7]
```

The tool randomly selects one path for back-tracing. You can also choose a specific net to back-trace. For details, see the man page. Example 9-2 shows the DOUT[7] back-trace report.

Example 9-2 Report Generated by the print_net_backtrace Command

```
Time     Old Value->New Value  Net
184090     0 -> 1    inno_tb_top.xtor.DOUT[7]
184080     1 -> 0    inno_tb_top.xtor.XI6.XOUT7.NS
183950     0 -> 1    inno_tb_top.xtor.SEN
183500     1 -> 0    inno_tb_top.xtor.XI1.NET2C
183470     0 -> 1    inno_tb_top.xtor.RCK
183420     1 -> 0    inno_tb_top.xtor.XI1.X8A.N4
182390     0 -> 1    inno_tb_top.xtor.XI1.X8A.N3
181190     1 -> 0    inno_tb_top.xtor.XI1.X8A.N2
180290     0 -> 1    inno_tb_top.xtor.XI1.X8A.N1
180200     1 -> 0    inno_tb_top.xtor.XI1.NET2
180180     0 -> 1    inno_tb_top.xtor.XI1.RE_P
180140     1 -> 0    inno_tb_top.xtor.XI1.CLK_1
180060     0 -> 1    inno_tb_top.xtor.XI1.X4.NET1
180000     1 -> 0    inno_tb_top.CLK1
```

From this back-trace report, you can see that DOUT[7] successfully made the transition to 1 but failed to make the final transition to 0. At this point check if the XCLR signal is working correctly using the `print_net_backtrace` command on XCLR as follows:

```
esp_shell(explore)> print_net_backtrace inno_tb_top.xtor.XI6.XOUT7.XCLR
```

The tool reports the following:

```
Time Old Value New Value Net
=======================================================
220750 1 ->0 inno_tb_top.xtor.XI6.XOUT7.XCLR
220720 0 ->1 inno_tb_top.xtor.CLR_II
220690 1 ->0 inno_tb_top.xtor.XCLRIIFF.X2.XQ
220650 0 ->1 inno_tb_top.xtor.XCLRIIFF.X2.S
```

This report shows that XCLR is working correctly. The next step is to investigate the node NS and determine if XCLR transitions to 1. You need to identify the cause that prevents the XCLR from transitioning correctly. The node NS is in the output buffer, see “Interactive Signal Tracing Example 2 - ram1r1w Schematics” on page 9-14.

To investigate that node, use the interactive signal tracing `print_net_trace` command as follows:

```
esp_shell(explore)> print_net_trace inno_tb_top.xtor.XI6.XOUT7.NS \
                            160000 230000
```

Example 9-3 Output of Node NS Using the Interactive Signal Tracing Command

```
************************************
* -rcwhy inno_tb_top.xtor.XI2.RWL_0_
*  Time: 0
*  Current value: 0
*  Update triggered by change on:
*    inno_tb_top.xtor.XI2.RCK_ = 1
*    inno_tb_top.xtor.XI2.XF0.innovss = 0
*    inno_tb_top.xtor.XI2.W_0_ = 0
*  Pulldown xtors:
*    nmos (1.20/0.18, r=1.88, g=inno_tb_top.xtor.XI2.RCK_=1, i=MI4,
s=RNOR22, path=inno_tb_top.xtor.XI2.XF0)
*  Pulldown resistance: 1.88 kOhm
*  Loads:
*     0.026 pF inno_tb_top.xtor.RWL_0_
*    ---------
*     0.026 pF
*  Effective capacitance: 0.000 pF ( 0.000 fF )
*  New value: 0 (no change)
************************************
************************************
* -rcwhy inno_tb_top.xtor.XI2.RWL_0_
*  Time: 182480
*  Current value: 0
*  Update triggered by change on:
*    inno_tb_top.xtor.XI2.RCK_ = 0
*  Pullup xtors:
*    pmos (4.80/0.18, r=1.07, g=inno_tb_top.xtor.XI2.RCK_=0, i=MI2,
s=RNOR22, path=inno_tb_top.xtor.XI2.XF0)
*      pmos (4.80/0.18, r=1.07, g=inno_tb_top.xtor.XI2.W_0_=0, i=MI1,
s=RNOR22, path=inno_tb_top.xtor.XI2.XF0)
*  Pullup resistance: 2.14 kOhm
*  Loads:
*     0.012 pF inno_tb_top.xtor.XI2.XF0.NET1
*     0.026 pF inno_tb_top.xtor.RWL_0_
*    ---------
*     0.039 pF
*  Effective capacitance: 0.026 pF
*  New value: 1
*  Delay: 60 ps (@ 182540)
************************************
************************************
* -rcwhy inno_tb_top.xtor.XI2.RWL_0_
*  Time: 182540
*  New value: 1
************************************
```

From this output, you can see that the pullup resistance value is 6.43 and the pulldown resistance is 4.82. From this data, you can infer that DOUT[7] did not transition correctly because a resistance value of 6.43 is not adequate to pull the signal up. Instead of 6.43, the resistance value needs to be reduced by a factor of about 1.4 to function as a pull-up resistor. At this point, you have debugged your problem. Your remaining decision is to decide how to change the resistance value. To do this, use the ESP `set_instance_strength_multiplier` command or directly edit the resistance value in your SPICE netlist. Also consider using the delay handling capabilities of ESP, described in "Controlling Partial Transitions" on page 4-9.

In this example, you use the interactive signal tracing `print_net_backtrace` command to trace back the mismatched signal through the output buffer, sense amp, and clock generation blocks. The `print_net_backtrace` command enables you to pin down the problem area to the node NS. Then, you use the `print_net_trace` command to analyze the conditions at the node NS and resolve the problem.

Interactive Signal Tracing Example 2 - ram1r1w Schematics

This section contains the following:

- Figure 9-1 shows the top-level schematic of the ram1r1w design
- Figure 9-2 shows the clock generation circuit
- Figure 9-3 shows the sense amp and output buffer circuits

Figure 9-1 Top-Level ram1r1w1 Design

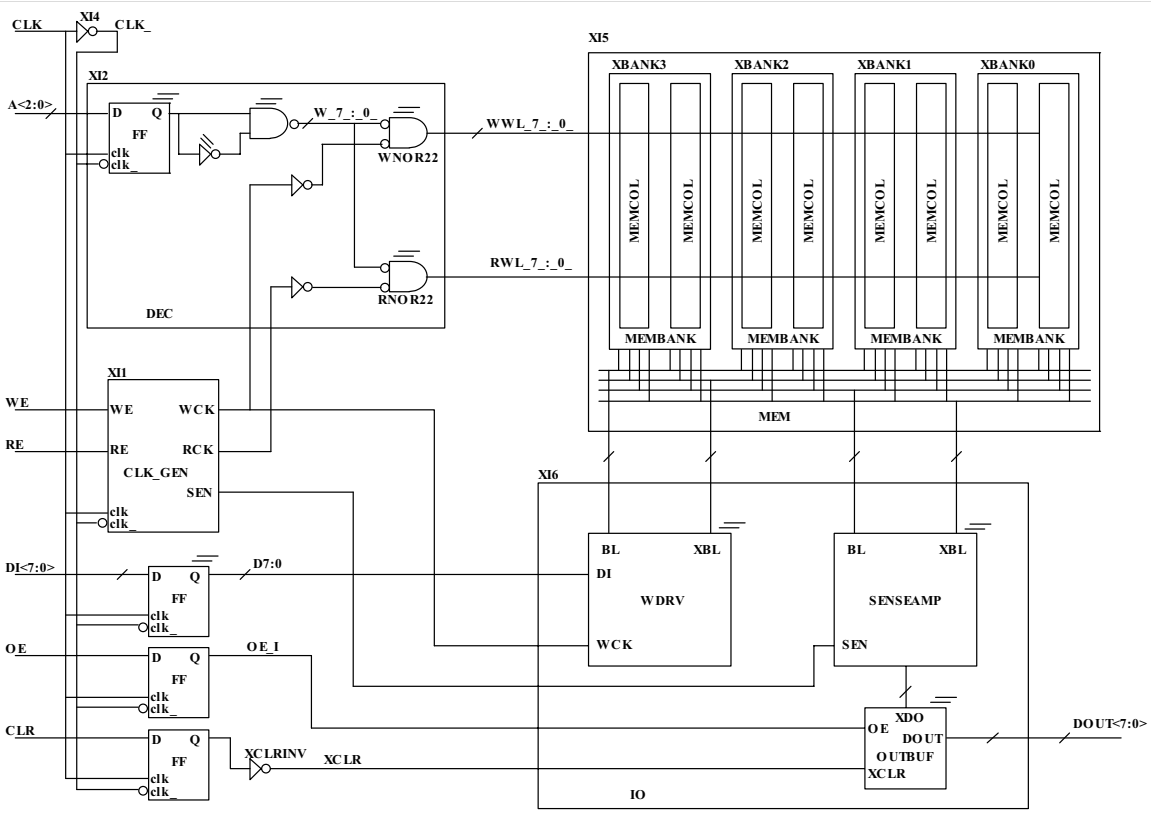


Figure 9-2 Clock Generation Circuit

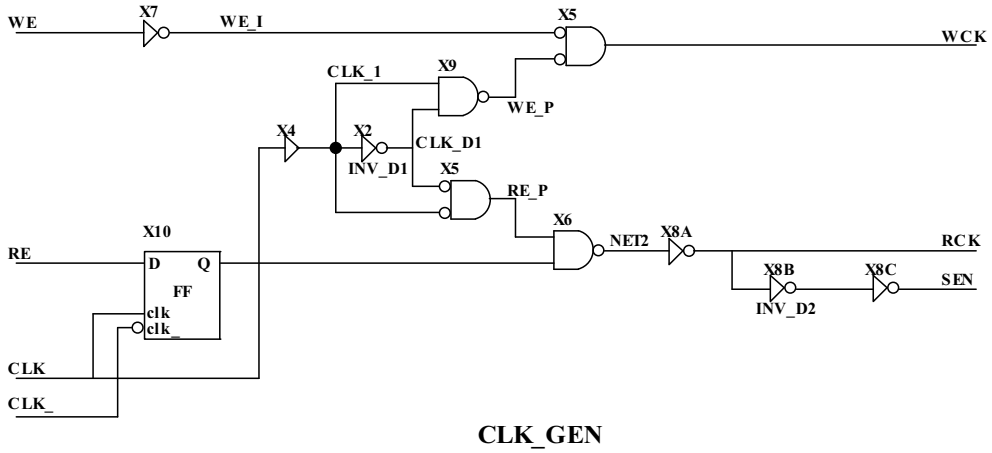
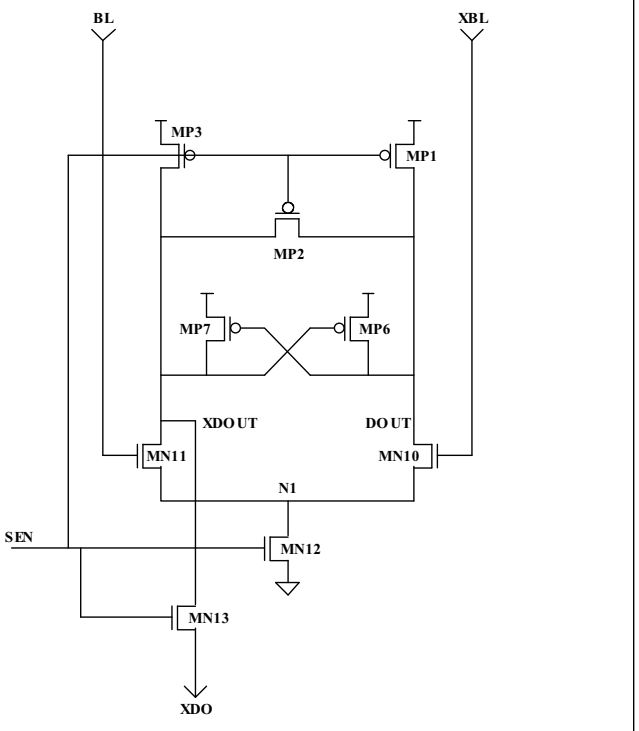
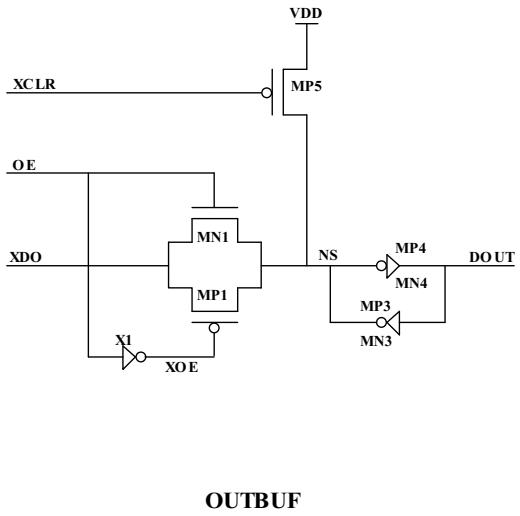


Figure 9-3 Sense Amp and Output Buffer Circuits





Verdi Debug Cockpit

The ESP tool allows Interactive Signal Tracing (IST) debugging capabilities through the Verdi® interface. When you enable IST, an ESP menu is added to the Verdi graphical user interface (GUI) pull-down menu. You can analyze ESP mismatch or power integrity violation (PIV) vectors within the IST environment and highlight SPICE design nodes in a schematic built from the native SPICE netlist.

To enable IST, use the `explore_with_viewer` command which has the following options:

- `-time` *`time`* : Start IST mode at the specified time
- `-tbid` *`testbench_id`* : Use an error vector from the testbench specified by the testbench ID
- `-vector_num` *`error_vec`* : Use an error vector from the specified testbench
- `-type` *`error|osc`* : Specify the vector type (error or oscillation)
- `-netlist_type` *`type`* : Specify the type of SPICE design netlist to pass to the viewer (Verilog or SPICE)
- `-trace_violation` *`number`* : Specify the PIV violation number to trace

For more information, see the `explore_with_viewer` command man page.

Debugging Passed Designs at Will

You can run a binary debug simulation in the ESP tool at any time, if you supply a vector file to be used for the run. Therefore, if you save an error vector file you can rerun that binary trace even if you resolve all the errors in the design. This allows you to verify whether a specific error sequence works and examine the internal waveforms that result from the error sequence.

The `debug_design` command has the following option:

`-vector_file <`*`file_name`*`>`

For example, if a verification run finds a mismatch, the tool produces an esp.TestVector file and you copy it to the esp.TestVector.save1 file. At a later time (even if the simulation passes), you can read in the designs, match them, issue the command, look at the log file, or examine the waveforms from that run.

```
debug_design -vector_file esp.TestVector.save1
```

Otherwise, you might have used constraints to set up 20 special binary cycles to exercise your design before your symbolic cycles and the simulation never fails. If you still want to

examine waveforms during the binary cycles, create a null file named dummy_file, and after reading the designs, and matching them, issue the following command:

```
debug_design -vector_file dummy_file
```

This file does not constrain any symbolic cycle values. Only waveforms from your binary cycles are useful in this case.

Displaying Symbolic Equation Data

To display symbolic equation data, use the `$esp_equation`, `$esp_equation_symbols`, and `$esp_equation_size` tasks. For example,

```
initial $monitor("Symbols used:%s\ nEquations:%s",
                 $esp_equation_symbols(A), $esp_equation(A));
```

For task details, see the respective man pages. To access the man pages for these tasks, do not use the “$” in the command. For example, to access the `$esp_equation` man page, use the `man esp_equation` command.

These tasks are ESP-specific tasks created to support symbolic equation data; they are not part of the 1995 Verilog Language Reference Manuals (IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language; IEEE Standard number: IEEE Std 1364-1995)

Stopping Simulation if Oscillations Occur

The ESP tool includes the following two variables that enable you to stop simulation if a specified number of oscillation conditions occur:

- `verify_stop_on_zero_delay_oscillation`

  Legal values are `true` and `false` (the default is `false`).

- `verify_stop_on_nonzero_delay_oscillation`

  Legal values are `true` and `false` (the default is `false`).

Setting the `verify_stop_on_zero_delay_oscillation` variable to `true` stops simulation if the number of zero-delay oscillations on a node at a given time exceeds the `verify_max_number_of_oscillations` value.

Setting the `verify_stop_on_nonzero_delay_oscillation` variable to `true` stops simulation if the number of nonzero-delay oscillations on a node at a given time exceeds 1024.

If you consider oscillating nodes to be errors or just want to find the first one and investigate it without spending any more CPU time on the simulation, then set one or both of these variables to `true`.

Stopping Randomization by Default

Randomization occurs when the tool reaches a threshold for the amount of memory used for holding symbolic equations. During randomization, the ESP tool randomly selects a symbolic variable and sets it to a binary value of 0 or 1. The tool displays a message stating the symbolic value that is randomized and to what value it is set. The tool creates a esp.Random file that keeps a list of all randomized symbols and their respective values.

It is not a good idea to let a verification randomize. Randomization represents a reduction in the functional coverage of a testbench. Divide-and-conquer techniques should be used to eliminate randomization.

By default, the `verify_stop_on_randomize` application variable is set to `true` so that the simulation stops and alerts you if randomization occurs. To allow the simulation to continue evenif randomization occurs, use the following Tcl command in your script before the `verify` command:

```
set_app_var verify_stop_on_randomize false
```

For more information, see the `verify_stop_on_randomize` and `verify_randomize_variables` man pages.

10

Cell Library Verification in the ESP Tool

This chapter describes cell library verification in the ESP tool in the following sections:

Library Verification Concepts

Library verification is used to verify custom or standard cell library components such as latches, multiplexers, domino logic gates, I/O drivers, level shifters and other devices that require validation. Library verification supports Interactive Signal Trace (IST), while maintaining the detailed testbench required for library verification. It also allows you to create library verification style testbenches for use outside library verification.

The library verification testbench includes the following features:

- Setting cycle-based input and output constraints
- Control over testbench task customization
- Always-X output checking

Library verification applies test vectors differently from any other ESP testbench styles. In this testbench, the tool applies the clocks first and then applies the data inputs after the clock changes in the hold time. The testbench efficiently tests latch-based devices, scan cells, and I/O pad cells. With this testbench, you can define output constraints based upon current input pin values.

Matching Designs

Library verification in ESP requires you to define a matched design object for every matched design. This design object stores the names of the top-level designs and all the information required for matching the top-level ports of the two designs and the information required to create a testbench to perform the verification. For library verification, the ESP tool can define thousands of matched design objects during setup. During the verify step, the tool validates all or a specified subset of these designs using matched design objects. When verification reports a discrepancy between two designs, you can debug the failing design.

Library verification uses different default settings for the verification variables than those used for design verification. To set the default for library verification or compare mode verification, use the `set_verification_defaults` command at the beginning of the shell session or the Tcl script. The library testbench style is added to the list of testbench styles available in ESP and becomes the default style when you set library verification default values using the `-library_verification` option.

The ESP Tcl commands use matched design objects to associate port matching of two designs. To set testbench pin and configuration attributes, use the `set_top_design -matched_designs` command.

The `match_designs` command performs name-based design matching between designs in the reference and implementation containers. To perform manual matching between designs that have different names, use the `set_matched_designs` command. To remove a

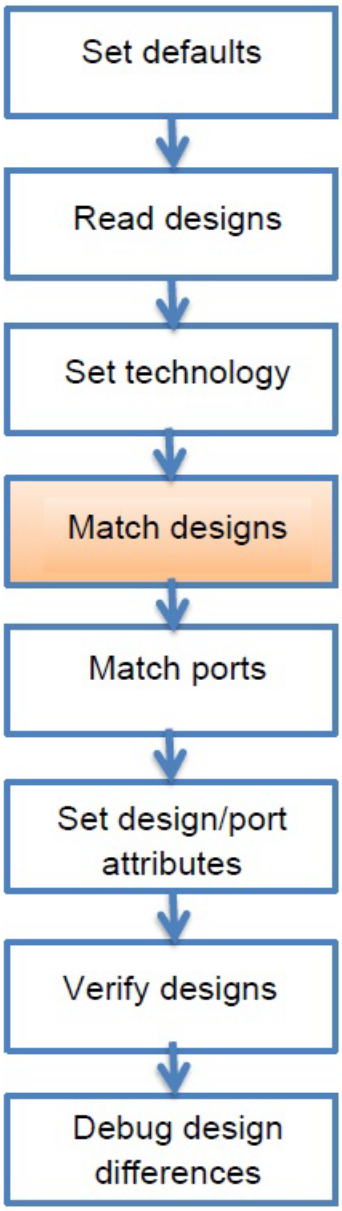
matched design along with the matched design object and previously set attributes, use the `remove_matched_designs` command.

To obtain a collection of the matched design objects, use the `get_matched_designs` command. To report the contents of one or more matched design objects, use the `report_matched_designs` command. To report the list of unmatched designs in one or more containers, use the `report_unmatched_designs` command.

Library Verification Flow

Figure 10-1 shows the steps used to perform verification in the ESP tool.

Figure 10-1 Library Verification Flow



The tool performs library verification by verifying multiple designs. The “Match designs” step identifies all the matching designs verified during the “Verify designs” step.

Creating a testbench for a matched design requires a list of matched ports and information about how to handle each port and how to apply and constrain symbolic test vectors. To simplify the creation of a testbench for each of the matched designs, you can set testbench properties for all or a subset of matched designs, or a specific design or testbench pin. To set the default settings for all matched designs or one or more designs, use the `set_matched_design_attributes` command. You only need to specify attributes that are different from the default.

When writing a testbench, the tool uses the value for the specific matched design or testbench pin. If you do not specify a value, the tool uses the default for the current matched design pair. If a default does not exist, the tool uses the global default for all matched designs.

Example 10-1 shows the library verification flow. The flow starts with setting library verification default values, which sets the testbench style for the library and sets the default output checker to be an exact comparison. The next command sets the default testbench clock cycle time to 40 ns for all matched designs.

Example 10-1 Library Verification Flow

```
# Set Defaults
set_verification_defaults -library_verification
create_clock -period 40 -setup 3 # Default testbench cycle time
set_matched_design_attributes -default –allow 01xz –checker eqzmx

# Read Design
read_verilog -r -vcs "myverilog.v +define+myoptions etc..."
read_spice   -i myspicenetlist

# Match Designs
set_matched_designs –r my_cell –i MYCELL
set_matched_designs –r my_cellto –i MYCELL

set mymat [get_matched_designs]

# Match Ports
set_matched_ports –matched_designs $mymat -i VSS
match_design_ports

# Set Design and Port attributes
set_matched_design_attributes –matched_designs my_cell –allow 01
set_testbench_pin_attributes –matched_designs $mymat VSS
      -function supply  value LOW
set_testbench_pin_attributes –matched_designs $mymat CLK
      -function clock –allow 01
```

```
# Verify Designs
verify –matched_designs $mymat

save_session firstverifyrun

# Debug Design Differences
set failed [get_matched_designs –filter “@status==failed”]
set dcell [list_attribute [lindex failed 0] name]
report_log -design $dcell

set_top_design –matched_designs $dcell
debug_design
quit
```

The flow consists of the following steps:

- Use the `set_matched_design_attributes` command to set default attributes for all the matched designs. In this case, all inputs have 0, 1, X, and Z inputs applied to them, and the default output checker is changed to allow an exact match when the reference output is Z and the implementation output is X. Default global design attributes can be reported and changed before or after reading the reference or implementation design.
- The tool then reads both the reference and implementation designs.
- After reading in both the designs, you identify the designs to be matched either by matching design names or specifying the designs manually. In this example, manual matching is done to match two reference designs to a single implementation design.
- After design matching is complete, you can match the ports of one or more designs. All ports must match before you set the testbench port and design attributes. The example shows that the port VSS only occurs in the implementation design and not in the reference.
- After port matching is complete, you can set the design and port attributes. The `set_matched_design_attributes` command shows that the default allowed input values for the ports of the matched design, my_cell, changed to 0 and 1 from the previous default input values used for all the other matched designs. The example shows that for all designs with a CLK port name, the function for that port is specified as a clock and the allowable inputs for all CLK ports is 0 and 1. This overrides any global defaults or design-specific defaults.
- When the entire setup is complete, you can perform verification on one or more of the matched designs. Note that distributed processing of designs is not supported with library verification. You can optionally save verification results after verification is complete. The save and restore feature allows you to examine and explore the

verification results in a future session. Saving and restoring also saves design and port attributes before running verification.

- After verification completes, you can report the passing or failing tests and then use the ESP Verdi® interface to view the waveforms.

Library verification expects the reference design to be a SystemVerilog or Synopsys .db file. The tool does not accept a SPICE reference design. The implementation design is expected to be SPICE for most features to work.

Managing Design Data

Reading Verilog or SPICE data into a container or clearing a container removes previously matched designs. Using the `read_db` command, library verification provides the ability to read Synopsys .db files as a reference model to compare against SPICE netlists.

You cannot change attributes in designs because it causes the SPICE data to become invalid for a particular design. However, you can change the SPICE netlist attributes before writing any ESP .db file. This occurs in the recommended flow by default. The ESP .db file is only written when you run the `check_design` or `verify` commands unless you specify that the file is written earlier.

The following commands change the container source data and invalidate all previously matched design data:

- `read_spice`
- `read_spice_behavioral_model`
- `read_verilog`
- `reset`

The matched design data is deleted for all previously matched designs.

Library Verification Commands

This following sections describe the ESP commands that support library verification:

- Commands to Manage Design Data
- Commands to Match Designs
- Commands to Match Ports
- Commands to Set Testbench Design Attributes

Commands to Manage Design Data

This section describes the commands used to manage design data.

The read_db Command

To specify a Synopsys .db file for a reference design, use the `read_db` command. Synopsys .db file data cannot be read into the implementation container. The .db file is transformed into an encrypted Verilog model for use during verification. The VCS parser does not support encryption. As a result, you cannot use the `read_db` command with the VCS parser.

The usage of the `read_db` command is as follows:

```
read_db    # Read DB formatted design file
     [-r]       (Read into reference container)
     File_names (List of files to read)
```

The write_esp_db Command

The ESP .db file only represents the top design that is in use. Therefore, you can use the `write_esp_db` command to generate an ESP .db file representing the SPICE data that is located in the implementation container for each design during library verification. To avoid using the wrong file, do not use this command during library verification. The `check_design` or `verify` command generates the ESP .db file in the correct location for each design during verification. The only change needed for library verification is to record the location of the ESP .db file in the matched design object.

After the ESP .db file is written, changing any of the design attributes causes an error, preventing the command from executing. Writing an ESP .db file for any design prevents changes to the design attributes for other designs in the top design. The `–matched_designs` option allows you to write ESP files for one or more matched designs. The `write_esp_db` command does not allow a file name to be specified when you use the `-matched_designs` option because each ESP .db file is written to the correct working directory for every matched design.

The usage of the `write_esp_db` command is as follows:

```
write_esp_db    # Write SPICE Verilog model file
     [-matched_designs list] (Write ESP DB file for matched designs
specified in the list)
     [-i]                   (Write model from implementation container)
     [File]                 (SPICE Verilog model file name)
```

Commands to Match Designs

To get the list of matching design pairs found in the containers, use the `match_designs` command. The command gets the list based on identical design names in each container. Command line options can limit the matching to be case-sensitive or to only match designs that have identical port names and sizes. Modifying the contents of any container by reading new data or resetting the container automatically clears the list of matching design pairs and previously set design attributes.

To manually match designs, use the `set_matched_designs` command. To remove a matched design use the `remove_matched_designs` command. To return a collection of matched designs, use the `get_matched_designs` command.

The match_designs Command

To match design names between the reference and implementation containers, use the `match_designs` command. If the names match, the matched design names are added to the list of matched designs. Matching is case insensitive but an exact match is prioritized. If there are multiple case-insensitive matches, one design is chosen arbitrarily. If more than one match is found, the tool issues a warning. To perform case-sensitive matching, use the `-exact` option with the `match_designs` command. To perform a preliminary check of port names and sizes so that the list of matched design pairs only contain designs that match by design name and have identical ports, use the `-strictports` option.

If match information already exists for a reference design, matching is skipped for that design because it is already done. The tool issues a warning message showing that the design match was skipped.

The usage of the `match_designs` command is as follows:

```
match_designs    # Match designs in reference and implementation by
design name
     [-exact]        (Only perform a case-sensitive design name match)
     [-strictports]  (Matches designs only if they have matching port
names and sizes)
```

The set_matched_designs Command

To manually match the designs that do not match by name, use the `set_matched_designs` command and specify a reference design and an implementation design. If the reference design occurs more than once or if a match already exists for that reference design, the tool issues a warning message indicating that the specified match already exists and that the current match is being skipped. Design name matching is case-sensitive.

The usage of the `set_matched_designs` command is as follows:

```
set_matched_designs    # Specify designs which match between reference
and implementation container
     -r design              (Reference container design name)
     -i design              (Implementation container design name)
```

The remove_matched_designs Command

The `remove_matched_designs` command removes one or more matched design pairs from a list of matched designs. Removal of the matched design information deletes any existing working directory and all previously set design port match information for the matched design. To remove all matched designs, use the `-all` option. The arguments for this command are the list of matched designs to be removed from the list. The reference design name is used to identify the match to be removed.

The usage of the `remove_matched_designs` command is as follows:

```
remove_matched_designs    # Specify matched designs to be removed
     [-all]                 (Remove all matched designs)
     [arg_list]             (Optional list of designs to remove from the
match list
```

The report_matched_designs Command

The `report_matched_designs` command reports all the matched design pairs. The usage of the `report_matched_designs` command is as follows:

```
report_matched_designs  # Report matched designs
```

```
esp_shell> report_matched_designs
**************************************
Report : report_matched_designs
Options:
Version: J-2014.12
Date   : Wed Jun  4 00:57:26 2014
**************************************
ID            Reference                Implementation      Status
my_cell       my_cell                  MYCELL              UNSET
my_cellto     my_cellto                MYCELL_____________UNSET
Total matched designs found: 2
1
```

The report_unmatched_designs Command

The `report_unmatched_designs` command reports all the unmatched designs in both containers. The usage of the `report_unmatched_designs` command is as follows:

```
report_unmatched_designs  # Report unmatched designs
```

```
esp_shell> report_unmatched_designs
**************************************
Report : report_unmatched_designs
Options:
Version: J-2014.12
Date   : Wed Jun  4 00:57:26 2014
**************************************
Unmatched reference designs
 R:nand

Total unmatched reference designs found: 1

Unmatched implementation designs
 I:NANDX
 I:NORX

Total unmatched implementation designs found: 2
1
```

The get_matched_designs Command

The `get_matched_designs` command returns a collection of matched design pairs. To get a collection of matched designs that matches specific criteria, use the `get_matched_designs` command with the `get_attribute` and `filter_collection` commands. To filter the returned collection based on attributes of the matched designs, use the `-filter` option.

The usage of the `get_matched_designs` command is as follows:

```
get_matched_designs  # Return a collection of matched designs
     [-filter expression]   (Filter for list of collections)
```

Commands to Match Ports

With the possibility of many existing matched designs, the port matching commands have the `-matched_designs` option to simplify matching ports on each of the matched designs.

The match_design_ports Command

The `match_design_ports` command matches all the ports by name between the current top reference and implementation designs. Name matching is case insensitive. The `-exact` option enables case-sensitive matching so that only ports with the exact names are

matched. The `-matched_designs` option allows you to specify a list of matched designs to perform port matching on. The matching goes through all the specified matched designs and match ports for each matched design pair.

The usage of the `match_design_ports` command is as follows:

```
match_design_ports         # Match design ports in reference and
implementation by name
     [-exact]                    (Performs case-sensitive port name match)
     [-matched_designs list]     (Do port matching on all matched designs
specified in the list)
```

The set_matched_ports Command

The `set_matched_ports` command allows you to manually match ports on matched designs when the ports do not have the same name or bit size. By default, the testbench port name is the first port name in the reference port list. To change the port name used in the testbench, use the `-tbpin` option. If there are no designs specified, the tool uses the current top-level reference design.

The usage of the `set_matched_ports` command is as follows:

```
set_matched_ports    # Manually match top-level design ports
     [-matched_designs list] (Do port matching on matched designs
specified in the list)
     [-tbpin tbportname]   (Optional name of port to be used in the
testbench)
     [-r ref_ports]        (List of reference design ports to be
associated with testbench port)
     [-i imp_ports]        (List of implementation design ports to be
associated with testbench port)
```

The tool issues a warning message for each design that the command does not succeed in matching the specified ports. The design that fails the match is not changed due to the failure.

The remove_matched_ports Command

The `remove_matched_ports` command removes specified matched ports from the current design or a specified list of designs.

The usage of the `remove_matched_ports` command is as follows:

```
remove_matched_ports # Remove matched top-level ports
     [-matched_designs list] (Do port removal only on matched designs
specified in the list)
     [-r ref_port_name]  (List of reference design ports)
     [-i imp_port_name]  (List of implementation design ports)
     [-tbpin tbportname] (Name of port used in the testbench to be
removed)
     [-all]              (Remove all matched ports)
```

The report_matched_ports Command

The `report_matched_ports` command reports all the testbench pins or a specified matched port from the current design or list of designs.

The usage of the `report_matched_ports` command is as follows:

```
report_matched_ports # Report Matching Information about matching design
ports
     [-tbpin tbportname] (Name of a specific testbench port to report on)
     [-matched_designs list]                  (Report matched testbench
ports on all matched designs specified in the list)
```

The report_unmatched_ports Command

The `report_unmatched_ports` command returns all the unmatched pins for the current design or a specified list of designs.

The usage of the `report_unmatched_ports` command is as follows:

```
report_unmatched_ports # Report information about unmatched design ports
     [-matched_designs list]                  (Report unmatched ports on
all matched designs specified in the list)
```

The get_testbench_ports Command

The `get_testbench_ports` command returns a collection of testbench ports for the current matched design or a specified design. You can report only one matched design. If the matching is not complete, the command reports the list of currently matched testbench ports.

The usage of the `get_testbench_ports` command is as follows:

```
get_testbench_ports # Returns a collection of testbench ports for a
design.
     [-design design]     (Report testbench ports for the matched design
specified)
```

Commands to Set Testbench Design Attributes

Library verification provides the ESP tool the capability to set attributes on more than one top-level design during setup.

The commands that affect how a testbench is formatted have the `-matched_designs` option to specify the matched design to which the command is applied.

The usage of the `-matched_designs` option is as follows:

```
[-matched_designs list]    (Apply to matched designs specified in the
list)
```

The following commands include the `-matched_designs` option:

Table 10-1 Commands and Their Actions

Command	Action
`list_active_testbench`	List active testbenches
`remove_clock`	Remove clock attributes from testbench pins
`remove_constraint`	Remove a previously created input constraint, match the `set_constraint` command arguments or use the `-all` option
`remove_testbench`	Remove testbench files to be used in verification
`rename_testbench_pin`	Change the name of a top-level testbench pin
`report_clock`	Report clocks in a design
`report_constraints`	Report all input constraints
`reset_clock`	Reset clock parameters to the defaults
`reset_testbench_pin_attributes`	Reset testbench pin attributes to the defaults
`set_active_testbench`	Set testbench files to be used in verification
`set_constraint`	Create a new constraint
`set_initialization`	Set the simulation netlist initialization
`set_symbol_to_pass`	Configure the testbench pins to be `$esp_choose_var`
`set_testbench`	Set testbench files to be used in verification

The tool ignores the following commands when a library verification testbench is generated even though they support the `-matched_designs` option because they are not used in the library verification testbench. The testbench generator issues a warning when it finds that values are requested for these features.

```
set_input_delay      # Set an input delay on a testbench pin
set_portgroup_constraint # Setting the portgroup constraint
report_portgroup_constraint # Reporting the portgroup constraint
```

The set_verification_defaults Command

The `set_verification_defaults` command sets default testbench and simulation parameters that are used for library verification. To set the defaults for library verification, use the `–library_verification` option. To set the defaults for non-library verification that are the current compare mode defaults, use the `–design_verification` option. By default, the ESP tool uses design verification values. This command overrides the existing variable values with the new values specified.

Note:

The `set_verification_defaults` command affects many defaults in the ESP shell and should be the first command in the library verification flow sequence. You must also use the `set_verification_defaults` command after using any of the verification commands.

The usage of the `set_verification_defaults` command is as follows:

```
set_verification_defaults # Set verification global defaults
     [-library_verification]     (Set verification defaults for library
cell verification)
     [-design_verification]     (Set the verification defaults for
non-library cell verification. This is the default)
```

The create_clock Command

Library verification needs to differentiate between a primary clock, buffered clock, and secondary independent clock that can have a different waveform controlled by clock constraints. To specify the primary clock, secondary clock, or buffered clock in a testbench, use the `-clktype` option of the `create_clock` command. The primary clock is the only toggled clock in binary and flush cycles and is the first toggled input in the initialization or reset vector. The `-clktype` option is used exclusively for library testbenches.

If you specify the primary clock option for more than one clock, the tool exits with an error when you try to set the primary clock after one has been specified. You can change the primary clock by removing it and adding the clock again without specifying the primary clock type. The tool ignores the `-period` and `-setup` options of the `create_clock` command for all clocks except the primary clock.

To define an equation that determines the condition under which a secondary clock can change, use the `-constraint` option. You can also use the `–constraint` option to specify a copied or inverted buffered clock. The tool exits with an error if you specify the `-constraint` option with a primary clock. This option is also used exclusively for library testbenches.

The tool issues a warning and ignores the `-delay` and `-phase` options of the `create_clock` command when you create library testbenches. The warning shows up in the `write_testbench`, `check_design`, or `verify` stage depending on when the testbench is

created. If a matched design does not have a clock pin, the tool issues a warning message and ignores the clock setting for that design.

The usage of the `create_clock` command is as follows:

```
create_clock    # Configuring the clock waveform for an existing
testbench pin
     [-matched_designs list] (Set testbench clock for matched designs
specified in the list)
     [-period nanoseconds]  (Clock period)
     [-setup nanoseconds]   (Clock setup/hold time)
     [-initial 0|1]         (Clock initial value)
     [-delay nanoseconds]   (Clock phase delay offset to start of period)
     [-phase nanoseconds]   (Clock first phase length if not 50% duty
cycle)
     [-clktype string]     (One of primary, buffered, secondary)
     [-constraint equation] (Library testbench constraint for clock)
     [Clock pin name]       (List of Clocks to be added or default clock
values if no clock name specified)
```

The set_matched_design_attributes Command

To set the default or design-specific attributes for creating library verification testbenches, use the `set_matched_design_attributes` command. If you update the current reference top design attributes, the `-default` and `-matched_designs` options are not available. To change the attributes of a specific design other than the top design, you must specify an existing reference container matched design name with the `-matched_designs` option. The tool issues an error message if the design does not exist. Any change to either container removes all the matched design pair attributes. For example, to remove all design-specific attributes, use the `reset -i` command. After you write the testbench, you cannot change the attribute values.

To set the strength of all Verilog testbench drivers for a design, use the `-verilogdrive` option. The current Verilog drivers use a default strength of `strong`. You can select a stronger strength of `supply`, or a weaker strength of `pull`, or the weakest strength of `weak`.

To define the channel length, use the `-channellength` option. SPICE testbench drivers use transistor sizes to set the SPICE netlist port drive strength and output capacitive load. The drive input delay on these drivers is 0, except for transition to Z, to match the Verilog driver delays. The tool extracts the default channel length from the SPICE netlist transistors. If none are found, the tool uses a default of 0.18 microns.

To set the default driver strength for all input or inout drivers, use the `-indrive` option. The default is 10 microns, which corresponds to the N transistor width. The P device width is set to two times the N channel width. All output and inout ports are attached to an inverter gate capacitive load.

To specify the transistor N channel width, use the `-outload` option. The default is 10 microns.

To select a check mode, use the `-checkmode` option. The default output check mode is `discrete`. Library verification supports a one check per phase called `discrete`, a two check per phase called `discrete2` (similar to the testbench `styles 4check` feature), and a continuous output check.

The continuous output check mode uses a delay between the time an output changes and the time when the comparison of the two outputs is complete. To change the delay value, use the `-checkdelay` option. The default is 1 time unit.

To specify the initialization vector for each design or one of the keywords (`zero`, `onehot`, `functional`), use the `-initialization` option. If you do not specify the initialization vector, the tool uses the value of the `testbench_initialization_file` application variable that all the other testbench styles use. To change the default setting of the `testbench_initialization_file` variable, use the `–default` option.

To specify if one set of symbols is applied per phase (`functional`), each symbolic input is changed individually in a one-hot manner (`skewmode`) or multiple symbolic vectors are applied per phase (`derace`), use the `-vectormode` option. The default is `functional`.

The `set_testbench_pin_attributes` command specifies the defaults for all input, output and inout ports. If you do not specify any of the defaults for a specific port, use the following options:

- To set the permissible symbolic input values, use the `-allow` option.
- To set the output checker, use the `-checker` option. Library verification introduces new short-named checkers. Library verification supports these compare functions: `eq`, `eqx`, `eqz`, `eqxz`, `eqzmx`, `eqx_zmx`, `eqw`, and `equ`.
- To specify the library verification mode used to drive bidirectional ports, use the `-iomode` option.

To specify a different global constraint file for each design, use the `–constraint` option. To change the `testbench_constraint_file application` variable and set a default global constraint file for all testbenches, use the `–constraint` option with the `–default` option.

To set the resistance threshold and determine if a driver is weak or not, use the `–weak_threshold` option. The variable value is specified in k Ohms and the default is 50 k Ohms. The weak output checker uses this value.

To specify a declaration file to be included in a testbench, use the `–declaration` option. To change the `testbench_declaration_file` application variable and set a default declaration file for all testbenches, use the `–default` option with the `–declaration` option.

To specify a setup file to be included in a testbench, use the `–setup` option. To change the `testbench_setup_file` application variable and set a default global setup file for all testbenches, use the `–default` option with the `–constraint` option.

The usage of the `set_matched_design_attributes` command is as follows:

`set_matched_design_attributes`	Sets design attributes for testbench creation
`-default`	Updates default attribute values
`-matched_designs list`	Lists the matched designs to set attributes on
`-verilogdrive str`	Specifies a Verilog drive strength. One of Supply, Strong, Pull, Weak. Default is Strong
`-channellength num`	Specifies a channel length of testbench drivers. Default is minimum SPICE transistor length o r 0.18 microns
`-indrive num`	Specifies the input driver strength in microns to be used with all SPICE inputs
`-outload num`	Specifies the output load in microns to be applied to all SPICE outputs
`-checkdelay num`	Specifies the check delay to be used with continuous output check mode. Default is 1ns
`-checkmode mode`	Specifies the output check mode. One of discrete, discrete2, continuous. Default is discrete
`-initialization str`	Specifies a predefined initialization vector type or an external initialization file to reset the design at beginning of simulation
`-vectormode str`	Specifies the type of input vector to apply inputs. One of functional, skewmode, derace. Default is functional
`-checker str`	Specifies the default checker to be used for all outputs of design
`-allow str`	Specifies the default allowed input values during symbolic cycles
`-iomode str`	Specifies the handling of bidirectional port testbench driver
`-constraint file`	Specifies a global constraint file
`-weak_threshold flt`	Specifies the weak threshold for eqw output checker
`-declaration`	Specifies the declaration file
`-setup`	Specifies the setup file

The report_matched_design_attributes Command

The `report_matched_design_attributes` command reports the library verification design attributes for all matched designs. If you do not set a specific attribute, the command reports the default. The report shows the reference design and the implementation design that are matched with it. The command reports all the attributes set by the `set_matched_design_attributes` command and the number of binary and symbolic cycles used to create the testbench.

The usage of the `report_matched_design_attributes` command is as follows:

```
report_matched_design_attributes    # Reports testbench design attributes
for matched designs
[-matched_designs list]  {A list of matched designs to report attributes
on)
esp_shell (verify)> report_matched_design_attributes
**************************************
Report : report_matched_design_attributes
Version: J-2014.06
Date   : Wed Jun  4 18:16:01 2014
**************************************

Matched Design ID    Verilog  Chan  Input  Output Check Checker
Default  Bin   Sym   Input    IO     Init    Initialization

Drive   Length Drive  Load   Delay Type       Checker Cycle Cycle Values
Mode    Type    Vector

*Default*           Strong    0.18   10     10      1   Discrete   eq
2    5     01      _allowz onehot

my_cell             Weak      0.18   10     10      1   Continuous eqx
2    8     01x     _noz    functional
```

Commands to Set Testbench Matched Port Attributes

Library verification allows you to set attributes on more than one top-level design during setup. All commands that affect testbench pin attributes have an option to specify the matched designs to apply the command.

The set_testbench_pin_attributes Command

The `set_testbench_pin_attributes` command sets testbench pin attributes on a specific testbench pin. To identify the driver used for inout ports in library verification, use the `-iomode` option. The legal values for this option are `allowX`, `allowZ`, `noZ`, `inputonly` and `outputonly`. The default is `allowX`. To specify an equation of input ports that indicates when the tested design drives the bidirectional port, use the `-ioenable` option.

In library verification, the tool ignores testbench pin attribute function values except for function type `clock`, `supply` and `binary`. The tool also ignores the `-portgroup` and `-power_domain` option values.

The usage of the `set_testbench_pin_attributes` command is as follows:

```
set_testbench_pin_attributes    # Configuring the testbench pins
     [-matched_designs list] (Set testbench attribute for designs
specified in the list)
     [-function string]     (Set the Port Function)
     [-checker string]      (Set the Port Checker)
     [-value string]        (Set the active logic value of the port)
     [-radix string]        (Change testbench display radix in log file
from binary to octal or hexadecimal)
     [-portgroup list]      (List of portgroups)
     [-power_domain string] (Set the Port Power Domain)
     [-direction string]    (Set the tb port direction)
     [-allow list]          (Set the permissible symbolic input values)
     [-iomode string] (Set bidirectional port mode)
     [-ioenable equation] (Identify equation for when bidirectional port
is driven from the device)
     tbpin                  (Name of the port)
```

The report_testbench_pins Command

Library verification allows you to report testbench ports and their associated reference and implementation port connections and displays the port direction and testbench pin bus size in the `report_testbench_pins` command report. The `report_testbench_pins` command also reports the bidirectional port driver along with the specified enable condition, the specified clock type, and the clock constraint.

The usage of the `report_testbench_pins` command is as follows:

```
report_testbench_pins    # Report testbench pins in the design
     [-matched_designs list]                (Report testbench attributes
for matched designs specified in the list)
     -all                   (Report on all ports)
     tbpin                  (Name of a specific port)
```

The output format of the `report_testbench_pins` command is as follows:

```
**************************************
Report : report_testbench_pins
Design : dut
Version: J-2014.06
Date   : Thu Jun  5 01:58:50 2014
**************************************

Testbench Dir Size Active Input InOut  Output Func  Clk   Clk        Port
Groups Power Domain
PortName           Value  Value Mode   Checker      Type  Constraint

in_data   in   5    Low   01xz  n/a    n/a    Other  n/a  n/a
out_data  out  3    None  None  n/a    eq     Other  n/a  n/a
clk       in   1    Low   01    n/a    n/a    Clock  Pri  n/a
clkB      in   1    Low   01    n/a    n/a    Clock  Buf  ~clk
sclk      in   1    High  01    n/a    n/a    Clock  Sec  n/a
io_data   io   4    Low   01Z   _noz   eq     Other  n/a  enable
```

The write_testbench Command

To generate a testbench using the set of design and testbench pin attributes for the current top-level reference design, use the `write_testbench` command. To write a testbench for one or more matched designs, use the `-matched_designs` option. You need not use the `write_testbench` command during library verification, as the `check_design` or `verify` commands automatically generate testbenches in the correct working directory for each verified design.

Using the name of a testbench with the `write_testbench` command can cause a conflict when multiple designs attempt to use the same testbench file. The safer option is to use the command without specifying a file name if the tool is writing a testbench for more than one design. The command has to record the location of the testbench file in the matched design object.

The `write_testbench` command does not allow a `file_path` to be specified with the `-matched_designs` option so that a testbench file is written to the correct working directory for each matched design. After the testbench is written, changing any of the design attributes causes the tool to issue an error preventing the command from executing.

You cannot generate a library verification testbench in the inspector mode. The tool issues an error message and the command that initiated the creation of the testbench exits.

The usage of the `write_testbench` command is as follows:

```
write_testbench    # Write testbench template(s)
     [-matched_designs list]                (Write testbench for matched
designs specified in the list)
     [file_path]            (Testbench file name prefix)
```

Commands to Verify

To verify one or more designs during library verification, use the `check_design` and `verify` commands. Individual design directories store the verification data results for use during debugging.

The check_design Command

To perform a preliminary compile of a specified set of designs or the current top-level design, use the `check_design` command. The command reports any errors during compile. As the `check_design` command processes each design, it updates the status of the design. The `check_design` command then generates testbenches and writes out an ESP .db file if the tool has not previously generated testbenches for the design.

Note:

As soon as the ESP tool writes a testbench or generates a .db file for a design, the tool disables the commands that set the design attributes.

The usage of the `check_design` command is as follows:

```
check_design     # Check all testbenches for syntax errors with
constraints and initialization files
     [-matched_designs list]               (Check matched designs
specified in the list)
```

The verify Command

In library verification, the `verify` command verifies the current matched design or a specified list of matched designs. If you did not execute the `check_design` command, the `verify` command performs all the `check_design` command operations on each design in the list on which you did not execute the `check_design` command. After completing the `check_design` command tasks, the `verify` command verifies the list of matched designs one at a time.

The usage of the `verify` command is as follows:

```
verify     # Do verification
     [-matched_designs list]               (Verify matched designs
specified in the list)
```

Commands to Report Status

To report the status during library verification, use the `report_log`, `report_error_vectors`, `report_test_vectors`, `report_aborted_points`, `report_failing_points`, `report_passing_points`, `report_status`, and `report_coverage` commands.

The report_log Command

The `report_log` command shows the contents of the last log for the latest `check_design`, `verify`, `debug_design`, power intent verify, or IST run if no options are specified. It uses the `verify_log_file` variable to find the last log file. If options are used, the top design context is used to find the design to report on. To specify a matched design pair to report, use the `-design` option. If the design has no log files to report, the command issues a warning message. Using the `-design` option does not change the value of the current design or top design.

The usage of the `report_log` command is as follows:

```
report_log    # Report the current designs last simulation log
     [-testbench testbench_path]     (Only report log results of
specified testbench)
     [-tbid testbench_id]   (Only report log results of specified
testbench)
     [-supply]             (Report log for supply signals)
     [-design id]          (Report for the matched design specified)
```

The report_error_vectors and report_test_vectors Commands

The `report_error_vectors` and `report_test_vectors` commands report the same information. The report lists the error vector file, testbench path, and debug design switches that can be used to run design debugging on a test vector for a specific design. To identify the design to report, run the `set_top_design -matched_designs` command.

Commands That Report Design Points

The `report_aborted_points`, `report_failing_points`, and `report_passing_points` commands list the specific output ports that abort, fail, or pass verification. If you have not specified any options, the report shows the current matched design pair and testbench outputs checked and their status.

To report a list of verified designs, specify the `-matched_designs` option. To limit the report to a specific testbench within the current matched design, specify the `-testbench` option.

Note:

You cannot use the `-testbench` and `-matched_designs` options together with the same command.

The usage of the `report_passing_points` command is as follows:

```
report_passing_points    # Reports passing points
     [-testbench testbench_path]     (Report points for a given
testbench)
     [-substring substring] (Only report points which contain specified
substring)
     [-matched_designs list] {Report for all matched designs specified in
the list)
```

The usage of the `report_failing_points` command is as follows:

```
report_failing_points    # Reports failing points
     [-testbench testbench_path]     (Report points for a given
testbench)
     [-substring substring] (Only report points which contain specified
substring)
     [-matched_designs list]       {Report for all matched designs
specified in the list)
```

The usage of the `report_aborted_points` command is as follows:

```
report_aborted_points    # Reports aborted points
     [-testbench testbench_path]     (Report points for a given
testbench)
     [-substring substring] (Only report points which contain specified
substring)
     [-matched_designs list]       {Report for all matched designs
specified in the list)
```

The report_status Command

The `report_status` command reports the status of every testbench scheduled for verification or completed verification for the current matched design pair. To report the failures on a testbench basis for a list of matched designs, use the `–matched_designs` option.

The usage of the `report_status` command is as follows:

```
report_status    # Report the current verification status
[-matched_designs list]                (Report on matched designs
specified in the list)
[-pass]                (Report passing cells or testbenches)
[-fail]                (Report failing cells or testbenches)
[-abort]               (Report aborted cells or testbenches)
[-pend]                (Report pending testbenches)
[-testbench testbench_path]     (Only report results of specified
testbench)
[-tbid testbench_id]   (Only report results of specified testbench)
```

The report shows the status of the set of testbenches for each design:

```
esp_shell (verify)> report_status
**************************************
Report : report_status
Design : dut
Version: J-2014.06
Date   : Wed Jun  4 18:16:01 2014
**************************************

Reference Design:      r:/WORK/dut
Implementation Design: i:/WORK/dut

Verification:
     Status:        FAILED

     tb0: fail   ESP_WORK/dut/esp.tb.sym
     Failing Points:     15
     Passing Points:     0
     Aborted Points:     16

ESP_WORK/dut/esp.tb.sym
```

The `report_status` command returns a list of testbench paths that meet the criteria specified with the command line options.

Commands for Symbolic Coverage

The library verification testbench applies symbols to every clock phase. This means that the tool needs to drop some symbols whose signals are not expected to be sampled during that phase. This is a desired functional check but is not helpful for symbolic coverage.

Detailed coverage is only applicable to one design, so you must set the matched design pair context before attempting to get detailed symbolic coverage for a design. To obtain a summary of the results for all designs, use the `–matched_designs` command-line option. Attempting to get detailed coverage results using the `specifications`, `filterdetail`, and `-unusedfilters` options causes the tool to issue an error message.

Note:

You cannot use the `–matched_designs` and `–testbench` options at the same time.

The usage of the `report_coverage` command is as follows:

```
report_coverage     # Report coverage
     [-matched_designs list]                (Summary coverage for
matched designs specified in the list)
     [-spec file_name]      (Coverage Report Specification file)
     [-filterdetail]        (Enable report of filter details)
     [-unusedfilters]       (Enable report of unused filters)
     [-all]                 (Report coverage for all testbenches)

     [-testbench testbench_path] (Only report results of specified
testbench)
```

Commands to Debug

Library verification works on multiple designs, not just the current design. When you run a debug command, you must specify the matched design to be debugged. The `get_matched_designs` command provides a list of failing or aborted designs that can be used to set the matched design pair to be debugged.

The debug_design Command

To generate internal signal values in VCD or FSDB file format, use the `debug_design` command. To examine the waveforms, use the `start_waveform_viewer` command.

The set_top_design Command

To set the top design name in each container, use the `set_top_design` command. To specify the design to be debugged, use the `set_top_design -matched_designs` command to set the context of the design. This works with all the debug commands without having to specify a design for each command. None of the debug commands have an option to select the matched design to be debugged.

The `–matched_designs` option sets the top design from the containers of a previously-matched design pair. The `-matched_designs` option thus allows you to specify an existing matched design pair. You can specify only one matched design. The reference top design and the implementation top design name are set from the matched design pair's reference top design and implementation top design names respectively. You can specify only one of the options at a time.

The tool issues an error message if you change the top design with the `–r` or `–i` options after setting the top-level matched designs and does not change the requested top design.

The usage of the `set_top_design` command is as follows:

```
set_top_design    # Setting the top design within a container
     [-matched_designs]           (Set top designs from matched design)
     [-r]              (Set top design in reference container)
     [-i]              (Set top design in implementation container)
     design_name       (Design name / Matched design name)
```

Interactive Signal Trace (IST)

Library verification allows IST to be used to investigate test vector results by tracing events and their causes in the ESP SPICE engine. To set the debug context before starting IST, use the `set_top_design –matched_designs` command.

SPICE Testbench

Library verification allows you to run a SPICE simulator to investigate test vector results in SPICE. To set the debug context before generating the SPICE testbench, use the `set_top_design –matched_designs` command.

11

Advanced ESP Flows

The ESP tool has two advanced modes:

- Power Integrity Verification Flow

  The power integrity verification flow is used to verify power management techniques such as power domains and switching, retention modes, and isolation. The power integrity verification flow is also referred to as the power verification mode. To enable the power verification flow, specify the `set_verify_mode inspector` command. This flow expands the compare flow to include power integrity verification.

- Simulate-Only Flow

  The simulate-only flow can be used with your custom verification testbench to symbolically simulate your Verilog or SPICE design and verify its functionality. To enable this flow, use the `set_verify_mode simulate` command.

- Redundancy Verification Flow

  The redundancy verification flow verifies whether the redundancy logic is functionally correct and the logic meets the stated fault tolerance. The flow does not require the reference design to have any redundant logic.

Power Integrity Verification Flow

To verify power management techniques, the ESP tool supports power integrity verification with minor additions to the default compare flow.

This section includes:

Power Integrity Verification Flow Overview

Designers use various power management techniques such as power domains and switching, retention modes, and isolation to manage power within a design.

- Power domains and switching

  Low power applications use switchable power domains. Multiple power domains within a memory design separate the core memory from the peripheral control circuitry. These are coupled with retention techniques.

- Retention modes

  Memory is a critical part of a design and not all memory can be “powered down.” Some memory must retain its value. Various methods are used to retain the value and allow fast recovery when needed.

- Isolation

  When design elements are powered off, they no longer drive the nets to which they are attached. The receiving net has to handle floating net values. Special isolation cells are used to drive zeros and ones or to latch the last value.

To enable the power integrity verification flow, specify the `set_verify_mode` inspector command.

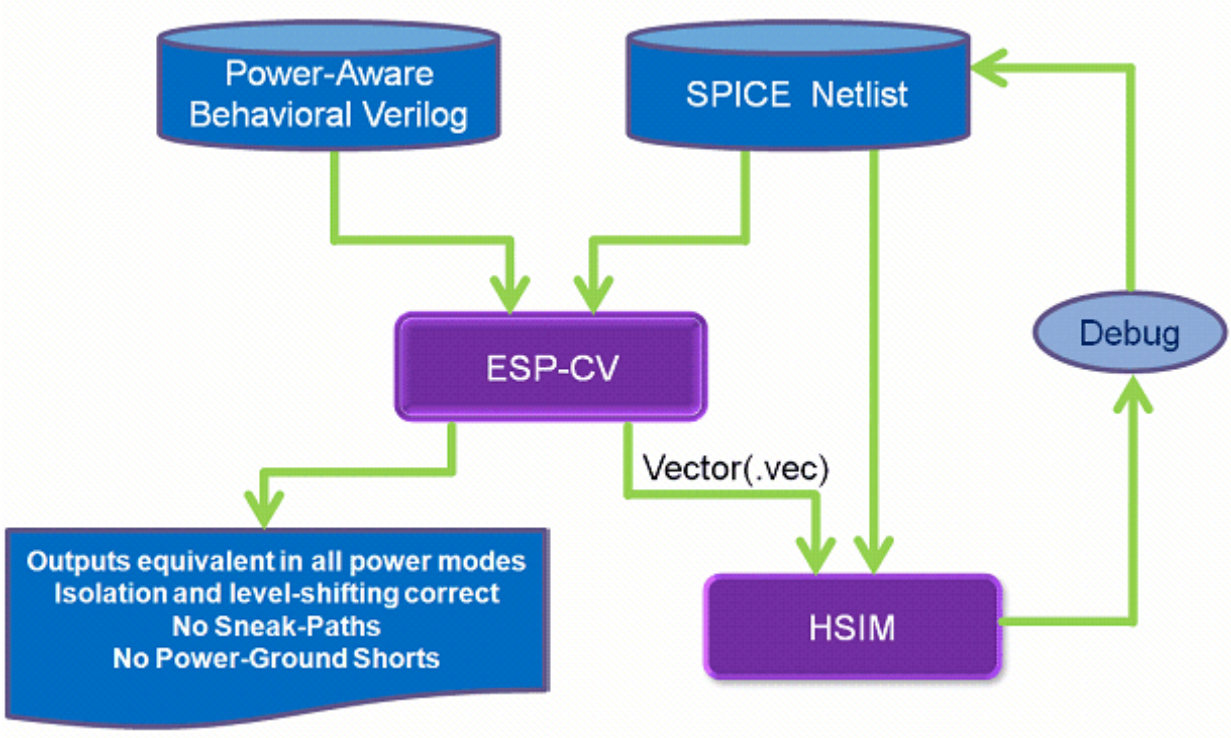
Power Integrity Verification Flow Methodology

Figure 11-1 shows the power integrity verification flow that is similar to the compare flow.

The primary differences are:

- Running the `set_verify_mode` command with the inspector value to enable power-integrity verification
- Using a low-power Verilog model (when available)
- Setting up the power-integrity checkers
- Reporting the power integrity violations

Figure 11-1 Power Integrity Verification Flow



A low-power Verilog model allows for complete verification. These models permit verification of retention modes. Level-shifter checks, short-circuit checks, and sneak-path checks can be performed without a low-power Verilog model.

To set up the ESP tool for power integrity verification, do the following:

1. Specify the `set_verify_mode inspector` command.

   This should be the first command in the command file because it resets the reference and implementation containers.

2. Specify all the power domains using the `set_supply_net_pattern` and `set_power_domain` commands.

3. Specify additional constraints using the `set_constraint` command.

4. Set up the power integrity checkers using the `set_inspector_rules` command.

5. Use the steps described in "Compare Flow Methodology" on page 3-3, as needed.

To run verification, execute the `verify` command. After verification is complete, view the results using the `report_inspector_results` command. Violation reports include the energy in pJ values and the power in nW values. The tool ranks the severity of the violations with these energy and power values.

To get a detailed report, use the `-id` option with the `report_inspector_results` command.

For example, to see the detailed report for error number 42, use the following command:

```
report_inspector_results -id 42
```

Power Integrity Verification Example Script

Example 11-1 shows the differences between a compare flow script and a power integrity verification script in different colors.

Example 11-1 Typical Power Integrity Verification Script

```
set_verify_mode inspector

# Read in the Reference Design
# Set the top of design
read_verilog -r ram.v
set_top_design -r ram1r1w

# Read in the implementation Design
# Set the top of the design
read_spice -i ram.sp
set_top_design -i RAM1R1W

# Match Reference and Implementation ports
match_design_ports
```

```
# Power domain information
set_supply_net_pattern -i -logic 1 -voltage 3.3 -type real VDD3
set_supply_net_pattern -i -logic 0 -voltage 0.0 -type real VSS3
set_supply_net_pattern -i -logic 1 -voltage 2.5 -type real VDD
set_supply_net_pattern -i -logic 0 -voltage 0.0 -type real VSS
set_constraint -set 1 VDD3
set_constraint -set 0 VSS3
set_constraint -set 1 VDD
set_constraint -set 0 VSS
set_power_domain -high_voltage 3.3 -low_voltage 0.0 -pins {DOUT VDD3 VSS3} \
outpad
set_power_domain -high_voltage 2.5 -low_voltage 0.0 -pins {A DI WE RE CLK \
VDD VSS} core

#  Defined Testbench settings: Be sure to set a clock
#  Generate a set of automatic testbenches
set_testbench_pin_attributes CLK -function Clock

#  Turn on code coverage
set_app_var coverage true

#  Run the verification
if ![verify] {
   #  If there is a failure look at the log files
    report_log
    debug_design
 } else {
   #  Generate a code coverage report
    report_coverage -all -filterdetail
 }
# Regardless of verification results
# report on the power intent checkers
# report on the power intent verification results
report_inspector_rules
report_inspector_results
```

For more information about how to download the design used in the examples, see the application note in SolvNet article 031109, “Power Intent Verification With ESP”.

Power Related Design Rule Violations

During verification, the ESP tool can detect several classes of design rule violations that directly affect power integrity.

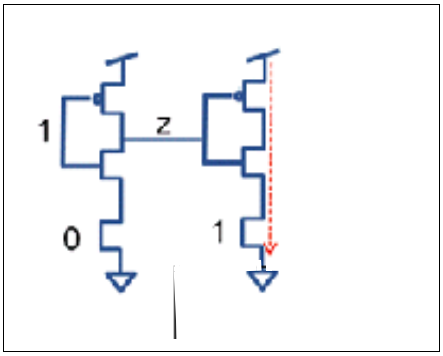
This section includes the following subsections:

- Incorrect Isolation
- Missing Level Shifter
- Power Ground Shorts
- Sneak Paths

Incorrect Isolation

Failure to provide isolation cells can result in excessive power drain because the path from power to ground might be enabled, as shown in Figure 11-2.

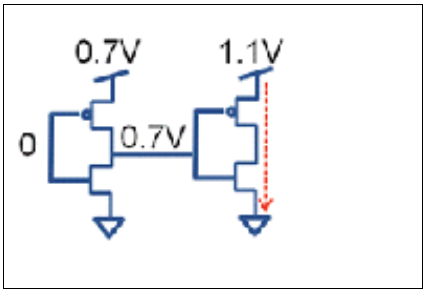
Figure 11-2 Incorrect Isolation



Missing Level Shifter

When a signal passes from a lower power domain to a higher power domain, a level-shifter cell is needed to avoid turning on both the PMOS and NMOS devices when there is a high value driven from the low power domain. This results in excess power dissipation, as shown in Figure 11-3.

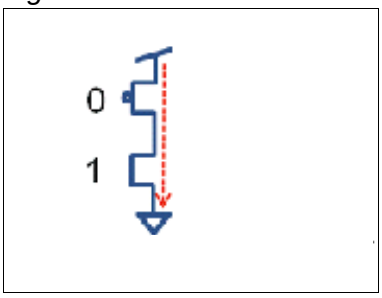
Figure 11-3 Missing Level Shifter



Power Ground Shorts

Improperly designed cells can result in power ground short circuits. Even when properly designed, there can be a momentary short circuit while signals transition to their stable values, as shown in Figure 11-4.

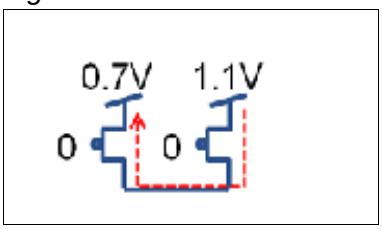
Figure 11-4 Power Ground Shorts



Sneak Paths

There can be circuit states that result in improper isolation of the power rails. Rails can be unintentionally connected to each other, as shown in Figure 11-5.

Figure 11-5 Sneak Paths



Reporting Options

The `report_inspector_results` command offers a variety of reporting options that enable you to get general or more specific information about a design or violations.

Example 11-2 Violation Report

```
*************************************
Report : report_inspector_results
Design : ram1r1w
Version: K-2015.12
Date   : Tue Oct 27 06:55:40 2015
*************************************
ID   |  Energy(pJ) = Avg_Power(mW) * Duration(ns)  |    Checker     |    Module.Net
 0                          NA                        INPUT_UNBUF   inno_tb_top.A_in__xtor
 1                          NA                        INPUT_UNBUF   inno_tb_top.DI_in__xtor
 2                          NA                        INPUT_UNBUF   inno_tb_top.RE_in__xtor
 3                          NA                        INPUT_UNBUF   inno_tb_top.WE_in__xtor
 4        2.704  =       0.2618  *       10.33        LEVEL_SHIFT   LAT.XQ
 5         NA    =         NA    *       19.98     LEVEL_SHIFT_HL   LAT.S
 6         NA    =         NA    *       10.34     LEVEL_SHIFT_HL   LAT.S
 7         NA    =         NA    *        9.64     LEVEL_SHIFT_HL   LAT.S
 8       0.1744  =       0.3963  *        0.44             SHORT   SENSEAMP.BL
 9       0.1744  =       0.3963  *        0.44             SHORT   SENSEAMP.XBL
 10       0.1744  =       0.3963  *        0.44             SHORT   SENSEAMP.XDOUT
 11       0.1744  =       0.3963  *        0.44             SHORT   inno_tb_top.DOUT_xtor
 12       0.1625  =       0.3963  *        0.41             SHORT   SENSEAMP.BL
 13       0.1625  =       0.3963  *        0.41             SHORT   SENSEAMP.XBL
 14       0.1625  =       0.3963  *        0.41             SHORT   SENSEAMP.XDOUT
 15       0.1625  =       0.3963  *        0.41             SHORT   inno_tb_top.DOUT_xtor
 16      0.05501  =       0.2037  *        0.27             SHORT   MEMCELL.S
 17      0.05501  =       0.2037  *        0.27             SHORT   MEMCELL.XS
 18      0.05501  =       0.2037  *        0.27             SHORT   RAM1R1W.RBL
 19      0.05501  =       0.2037  *        0.27             SHORT   RAM1R1W.WBL
 20      0.05501  =       0.2037  *        0.27             SHORT   RAM1R1W.XRBL
 21      0.05501  =       0.2037  *        0.27             SHORT   RAM1R1W.XWBL
 22      0.04049  =         0.15  *        0.27             SHORT   WDRV.NET1
 23      0.03277  =       0.3277  *         0.1             SHORT   FF.Q1
 24      0.03277  =       0.3277  *         0.1             SHORT   LAT.S
 25      0.03194  =       0.4563  *        0.07             SHORT   LAT.S
 26      0.02376  =       0.3394  *        0.07             SHORT   FF.Q1
 27      0.02376  =       0.3394  *        0.07             SHORT   LAT.S
 28      0.01104  =       0.2209  *        0.05             SHORT   MEMCELL.S
 29      0.01104  =       0.2209  *        0.05             SHORT   MEMCELL.XS
 30      0.01104  =       0.2209  *        0.05             SHORT   RAM1R1W.WBL
 31      0.01104  =       0.2209  *        0.05             SHORT   RAM1R1W.XWBL
 32      0.01023  =       0.1279  *        0.08             SHORT   MEMCELL.S
 33      0.01023  =       0.1279  *        0.08             SHORT   MEMCELL.XS
 34      0.01023  =       0.1279  *        0.08             SHORT   RAM1R1W.WBL
 35      0.01023  =       0.1279  *        0.08             SHORT   RAM1R1W.XWBL
 36      0.01023  =       0.1279  *        0.08             SHORT   WDRV.NET1
 37     0.009068  =       0.2267  *        0.04             SHORT   LAT.S
 38        1.013  =      0.05068  *       19.98             SNEAK   LAT.S
 39       0.5219  =      0.05067  *        10.3             SNEAK   LAT.S
 40       0.4886  =      0.05069  *        9.64             SNEAK   LAT.S
  Checker                          Violations
 SHORT                               30
 LEVEL_SHIFT                          1
 SNEAK                                3
 INPUT_UNBUF                          4
 LEVEL_SHIFT_HL                       3
  Total Unwaived Power Violations :     41
1
```

Create a detailed report for any violation shown in the report using the `-id` option to specify the violation. The detailed report includes the commands needed to perform further analysis using Interactive Signal Tracing.

Example 11-3 Detailed Violation Report

```
**************************************
Report : report_inspector_results
Design : ram1r1w
Version: K-2015.12
Date   : Tue Oct 27 06:55:40 2015
**************************************

Type            :                                    SHORT
Violation Range :                  <220540ps>...<220980ps>
Net Name        :            inno_tb_top.xtor.XI6.XR0.BL_1
Module Name     :                                SENSEAMP
Instance Name   :                inno_tb_top.xtor.XI6.XR0
Local Net Name  :                                   BL_1
Bus Net Name    :                                     BL
Bus Range       :                                    <1>
Local Inst Name :                                    XR0
Repetition      :                                      4
Energy          :                           1.743620e-01pJ
Average Power   :                           3.962773e-01mW
Duration        :                                   440ps
Peak Power      :                           3.962780e-01mW
Vector File     :    ./ESP_WORK/ram1r1w/tb0/pwr_short.1.tv
Debug Command   : debug_design -tbid tb0 -vector_num 1 -type pwr_short

IST Command     : start_explore -tbid tb0 -vector_num 1 -type pwr_short -time 220980;
print_net_trace {inno_tb_top.xtor.XI6.XR0.BL_1} 220540 220980; stop_explore

Viewer Command   : explore_with_viewer -tbid tb0 -vector_num 1 -type pwr_short -time
220980 -trace_violation {220540 220980 inno_tb_top.xtor.XI6.XR0.BL_1}

Spice TB Command  : write_spice_debug_testbench -tbid tb0 -vector_num 1 -type
pwr_short

Description    : At time 220980, checker 'SHORT' has been violated with power value
0.174362 pJ on net 'inno_tb_top.xtor.XI6.XR0.BL_1'(within module 'SENSEAMP' and
instance 'inno_tb_top.xtor.XI6.XR0')
1
```

Power integrity verification (the `set_verify_mode inspector` command) in the ESP shell has the following capabilities:

- Integration with the Verdi Debug Cockpit: The Verdi Debug Cockpit schematic view highlights power integrity verification (PIV) violating paths.
- Enhanced violation reporting: Enhanced violation reporting enables you to view the subsets of the violations that are of interest. For example, a SHORT violation detailed report includes the transistors that you turned on to provide a conduction path from power to ground.

You can accomplish this by adding the following new option to the `report_inspector_results` command:

`-voltage <voltage>`: Report the violations for the specified voltage

This option only applies to the output_voltage, pu_voltage and pd_voltage checkers and is usually combined with other options as follows:

```
report_inspector_results -i -voltage -greater_than 1
```

For more information and examples, see the `report_inspector_results` command man page.

See “Reporting and Interpreting Results” in Chapter 8 for other reporting options.

Simulate-Only Flow

In the simulate-only flow, the tool uses your Verilog or SPICE design along with your custom verification testbench to symbolically simulate your design and verify its functionality.

This section includes:

Simulate-Only Flow Overview

In the simulate-only flow, your testbench provides the expected results instead of another model. Your testbench must incorporate ESP-specific system functions to generate symbols and check the expected symbolic equations at the output.

The simulate-only flow supports the following features:

- Single `read_verilog` command to read all Verilog source files at the same time. The source files can contain any Verilog model, for example, RTL, switch level, or behavioral models. The `read_verilog` command must include the top-level testbench.
- Coverage data collection during symbolic simulation and coverage results reporting.
- Test failure counter examples for custom testbenches that follow ESP rules to report test errors. These are managed on a design by design basis.

To exit the simulate-only flow, execute the `set_verify_mode compare` command.

Enabling the Simulate-Only Flow

To enable the simulate-only mode, specify the `set_verify_mode simulate` command. The `set_verify_mode` command sets the internal state of the Tcl shell so that only one container is used and indicates that equivalence checking does not take place during the flow. In this flow, some Tcl commands are disabled because they are relevant only to the compare flow. For a list of supported commands, see “Supported Commands and Variables.”

When you execute the `set_verify_mode simulate` command, the value for the `waveform_dump_control` variable is disabled. When you switch from the simulate-only flow to the compare flow, the value of this variable is enabled.

Simulate-Only Flow Methodology

To run the simulate-only flow, use the following steps and see the example flow script in “Simulate-Only Flow Example Script” on page 11-13:

1. Exit the ESP GUI (if applicable).
2. Execute the `set_verify_mode simulate` command. See “Enabling the Simulate-Only Flow” on page 11-11.
3. Ensure that your testbench provides the expected verification results and incorporates ESP-specific system functions to generate symbols and check the expected symbolic equations. See “Creating Output Checker Specifications” on page 7-16.

4. Use the `read_verilog` command to read your design files and testbench file.

   In this flow, you specify all the Verilog source files to be simulated using the `read_verilog` command. You can specify every file used for simulation in the Verilog command option file `vfile`, or you can specify a list of files on the `read_verilog` command. Include the testbench file in the list of files.

5. Specify your top design.

   Use the `set_top_design` command to identify the top module of the design under test. If you do not set your top design, the tool uses the first non-referenced module name as the top-level design name. To obtain coverage data, you must set the top-level design correctly. This should be the top design the testbench instantiates and not the testbench top module name.

   This design name is used in the report headers and the work directory to store results.

   You can only set the top design to one container, the reference container.

6. To enable coverage checking and reporting, enable the `coverage` variable.

   Note: The `coverage_db_file` variable defines the name of a coverage output file to be used when outputting symbolic coverage data. The default is `esp.cov`.

7. Set your testbench module name.

8. Set your testbench instance name using the `testbench_design_instance` variable.

   If you do not specify a testbench name, the tool assigns a default testbench id of tb0 and identifies all internal result files with the testbench id of tb0. The testbench instance name can be a hierarchical path from the top-level testbench to a lower-level instance. For example, you could set the variable to the value x1.x15.ref.

   The `testbench_design_instance` variable is defined as an empty string. If the variable has a non-empty string value, this is interpreted as an instance name. The specified instance name is used to gather symbolic coverage data. If the instance name has no value, the tool searches the Verilog source code to find the design instance name specified by the `set_top_design` command.

9. Verify the design.

   The `verify` command runs symbolic simulation using the reference container. This command assumes that everything is in one container.

   When you request coverage, you must know the path to the design being tested. If the tool does not find any values for the top-level design instance name and the `testbench_design_instance` variable, it issues an error message.

10. Review the `report_status` report.

    For report details, see “Error Reports” on page 8-8. The `report_status` command must be able to report errors generated by custom testbenches. The errors that the

`report_status` command can detect and report depend on how you implement error checking in the Verilog source code. See “Creating Output Checker Specifications” on page 7-16.

11. Review the `report_log` report.
12. Review the `report_coverage` report. For an example report, see “Reporting Symbolic Coverage” on page 8-11.

    The Tcl shell adds symbolic coverage during the `verify` command when the `coverage` variable is enabled. To allow the Tcl shell to automatically gather symbolic coverage data, do the following:

    a. Specify the top-level testbench name using the `testbench_module_name` variable
    b. Specify the name of the coverage output file using the `coverage_db_file` variable or use the default output file name esp.cov

    For other reporting options, see “Reporting and Interpreting Results” on page 8-6.
13. Run the `debug_design` command, if needed.

    In simulate mode, the `debug_design` command uses only one container when generating the VCD files. The command uses variables to determine the top-level testbench name for output purposes. For waveform output details, see “Outputting Waveform Data” on page 9-5.

    For additional debugging information, see Chapter 9, “Debugging Your Design.”

Simulate-Only Flow Example Script

The ESP tool uses various the following Tcl commands to run the simulate-only flow.

Example 11-4 Simulate-Only Flow Methodology Script

```
set_verify_mode simulate
read_verilog -f vfile
set_top_design designname
set_app_var coverage on
set_app_var testbench_module_name mytbname
set_app_var testbench_design_instance dutname
verify
report_status
report_log
report_coverage
quit
```

Transistor-Only Simulation

In the simulate mode, you can simulate transistor-only designs without a Verilog reference design.

Transistor-only simulation consists of the following steps:

1. Enter the simulate mode:

   ```
   set_verify_mode simulate
   ```

2. Read the SPICE design into the implementation container.
3. Set the port directions:

   ```
   set_testbench_pin_attributes -direction
   ```

4. Set the custom testbench.
5. Simulate the design.

Example 11-5 Transistor-Only Simulation Script

```
set_verify_mode simulate
read_spice -i ram.sp
set_top_design -i RAM1R1W
set_testbench_pin_attributes CLK  -func Clock   -direction input
set_testbench_pin_attributes WE   -func Write   -direction input
set_testbench_pin_attributes RE   -func Read    -direction input
set_testbench_pin_attributes A    -func Address -direction input
set_testbench_pin_attributes DI   -func Data    -direction input
set_testbench_pin_attributes DOUT               -direction output
write_esp_db -i ram.gv
set_app_var coverage true
set_app_var testbench_design_instance xtor
set_testbench "VT.sym.xtor"
verify
report_log
report_coverage -all -spec cov.spec -filterdetail
quit
```

The transistor-only simulation capability uses binary input vectors; but the ESP tool also allows you to use symbolic input vectors.

For more information, see the `transistor_only` and the `set_testbench_pin_attributes` man pages.

Supported Commands and Variables

The simulate-only flow supports the following commands:

- `current_container`
- `current_design`
- `debug_design`
- `get_designs`
- `get_inouts`
- `get_inputs`
- `get_outputs`
- `get_pins`
- `read_verilog`
- `report_coverage`
- `report_log`
- `report_status`
- `report_top_design`
- `reset`
- `set_top_design`
- `set_verify_mode`
- `start_waveform_viewer`
- `verify`

The simulate-only flow supports the `testbench_design_instance` and `waveform_dump_control` variables.

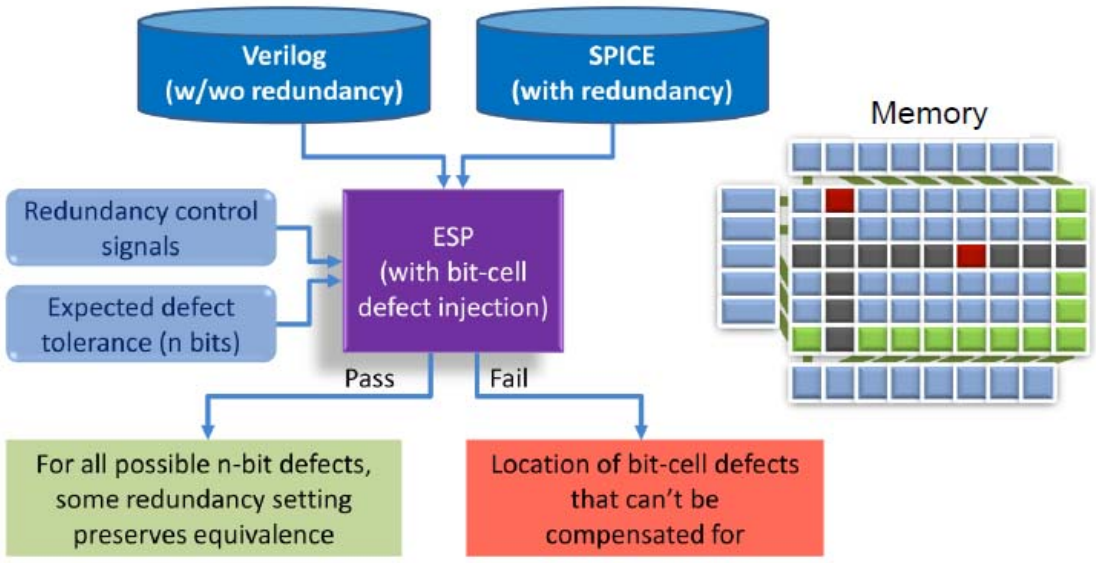
For details, see the man pages.

See “Reporting and Interpreting Results” in Chapter 8 for additional reporting options.

Redundancy Verification Flow

Large memory designs use redundancy logic to improve yield. The redundant logic consists of replaceable elements and elements to control the replacement. The ESP tool verifies redundant logic efficiently using the redundancy verification flow.

Figure 11-6 Redundancy Verification Flow



The redundancy verification flow verifies whether the redundancy logic is functionally correct and the logic meets the stated fault tolerance. The flow does not require the reference design to have any redundancy logic.

The redundancy verification flow applies faults in each replaceable element and then tries to find a set of values for the control ports that mask that fault at the design outputs. If the tool cannot find a set of redundancy control values that let the tested implementation mask a fault, it generates an error vector.

The redundancy logic flow extends the normal verification flow with the following additional capabilities:

- Specifying Fault Tolerance
- Identifying Replaceable Logic
- Identifying Redundancy Control Ports
- Reporting Options

Specifying Fault Tolerance

The number of replaceable elements is the fault tolerance of the design. The replaceable elements are often entire rows or columns. For very large designs, the replaceable element could be an entire subarray.

The `set_stuck_fault_group` command defines the fault tolerance for a specified set of faults. The `–number` option sets the number of blocks available for replacement. For example, if the design has one redundant column the following command defines the fault tolerance as one column:

```
set_stuck_fault_group –i –number 1 –net_group columns
```

Identifying Replaceable Logic

You help the ESP tool find replaceable logic elements by identifying a net within the replaceable element. The ESP tool symbolically applies faults to this net during verification.

The `set_net_group` command identifies a set of nets that the redundant logic replaces.

The `–net` option specifies the name of the net for the net group. The –net option is required. The net name is any net in the design. The net name can include hierarchical segments separated by the period character (`“.”`).

The asterisk character (`“*”`) is a wildcard character that represents zero or more characters. The wildcard character does not span hierarchical levels. You use the wildcard character in any part of the net name. The net name *RBL* matches nets in the current design level. The net name *.RBL* matches nets one level below the current design level.

The `–design` option specifies the name of the subcircuit to which the net name is relative. The design is the top-level subcircuit when the `–design` option is absent.

For improved performance, identify only one net within any replaceable element. A good candidate net is the output net of the element or a unique input net.

The read bitline is a good choice for column redundancy. This net typically spans the entire replaceable column. If the sense-amp is not part of the replaced column, you should also select the complementary bitline to provide verification that is more complete.

The read wordline is a good choice for row redundancy. This net typically spans the entire replaceable row.

The read bitlines of a memory can define the columns in a redundant memory. The following command shows how to specify this information:

```
set_net_group –i –net {RBL*} columns
```

The net expression “{RBL*}” uses the wildcard character * to indicate that any net in the current design level, starting with the characters RBL, is considered part of the “columns” net group.

When more than one net is needed to fully specify the net group, the –add option can be used with multiple set_net_group commands.

The following example shows how to use the –add and –design options to specify the replaceable columns in a memory with sub arrays leftArray and rightArray.

```
set_net_group -i –design leftArray –net {RBL*} columns
set_net_group –i –add –design rightArray –net {RBL*} columns
```

Identifying Redundancy Control Ports

The redundancy control ports are those ports that define and control redundancy operation. You use static symbols on these ports because redundancy is typically a static operation. During verification, ESP finds legal binary values for these ports that mask any faults defined by the `set_stuck_fault_group` command.

Two commands define a port as a redundancy control port:

- `set_constraint –symbolic_static` *`portname`*

  The `set_constraint` command with the `–symbolic_static` option defines a symbol that is applied at the start of simulation and is not changed. Do not use the `set_constraint –symbolic_static` command if redundancy controls change dynamically in normal operation.

- `set_symbol_to_pass` *`portname`*

  The `set_symbol_to_pass` command tells ESP which input symbols to change to find a set of values for the control ports that will mask a fault specified by the `set_stuck_fault_group` command.

  `portname` is the name of any port in the design.

If redundancy controls are not ports then you must use manual methods to specify the controls in ESP. The `$esp_choose_var()` system task defines a set of registers as symbols that ESP changes to make the design pass. The `$esp_choose_var()` system task is what the `set_symbol_to_pass` command adds to the testbench.

In the examples section, the Internal Control Latches example shows manual definition of redundancy controls when they are internal registers and not actual ports.

Reporting Options

The following commands show how redundancy verification is set up:

- `report_net_groups`

  The `report_net_groups` command shows the net groups defined by the `set_net_group` command. The following example shows the net groups report for the column redundancy example.

```
esp_shell (verify)> report_net_groups -all
      …
 read_bitlines : Add/Remove : Design : Net
                 Add                  RBL*
1
esp_shell (verify)>
```

- `report_stuck_fault_groups`

  The `report_stuck_fault_groups` command shows the fault groups defined by the `set_stuck_fault_group` command. The `report_stuck_fault_groups` command requires the `-i` option to specify reporting of faults in the implementation container. The following example shows the stuck fault groups report for the column redundancy example.

```
esp_shell (verify)> report_stuck_fault_groups -i
      …
Fault_Group Net_Group      Stuck_Faults
0           read_bitlines  1
1
esp_shell (verify)>
```

- `report_constraints`

  The `report_constraints` command reports the constraints from the `set_constraint` command. The following example shows the constraints report for the column redundancy example.

```
esp_shell (verify)> report_constraints
      …
 Make {REDUND} symbolic static.
 Make {RCA} symbolic static.
1
esp_shell (verify)>
```

Examples

This section includes examples of reports for the various redundancy verification commands:

- Single Column
- Multiple Column
- Column and Row Redundancy
- Internal Control Latches

Single Column

The SRAM supports replacement of a single column. The RCA input defines which column will be replaced. The REDUND input enables the replacement. The SRAM read bitlines define the column structure of the memory, so a fault on one of these lines is sufficient to indicate a bad column. The following example script marks the enhanced flow additions in purple. The example script, the Verilog file, and the SPICE file are available for download on SolvNet with this application note as the tar file redundancy.tgz.

```
# run_pass.tcl
read_verilog -r ram_redund.v
read_spice -i ram_redund.sp
match_design_ports
set_testbench_pin_attributes -func clock CLK
# # Identify redundancy control inputs set_constraint -symbolic_static
REDUND set_constraint -symbolic_static RCA set_symbol_to_pass REDUND
set_symbol_to_pass RCA
#
# Identify nets that are replaceable (read bitlines define a column)
set_net_group -i -net {RBL*} read_bitlines
#
# Define allowed fault tolerance (1 column can be replaced)
set_stuck_fault_group -i -number 1 -net_group read_bitlines
write_testbench tb.1
verify
report_log
quit
```

In the preceding example, the MEMCOL subcircuit has the net RBL. To avoid possible conflicts in other parts of the design, rewrite the `set_net_group` command as:

```
set_net_group –i –design MEMCOL –net RBL read_bitline
```

Multiple Column

The SRAM supports replacement of three columns. The RCA1, RCA2, and RCA3 inputs define which columns will be replaced. The REDUND input enables the replacement. The SRAM read bitlines define the column structure of the memory, so a fault on one of these lines is sufficient to indicate a bad column. The following example script marks the enhanced flow additions in purple.

```
read_verilog -r ram_redund.v
read_spice -i ram_redund.sp
match_design_ports
set_testbench_pin_attributes -func clock CLK
#
# Identify redundancy control inputs
set_constraint -symbolic_static REDUND
set_constraint -symbolic_static {RCA1 RCA2 RCA3}
set_symbol_to_pass REDUND
set_symbol_to_pass {RCA1 RCA2 RCA3}
#
# Identify nets that are replaceable (read bitlines define a column)
set_net_group -i -net {RBL*} read_bitlines
#
# Define allowed fault tolerance (3 columns can be replaced)
set_stuck_fault_group -i -number 3 -net_group read_bitlines
write_testbench tb.1
verify
report_log
quit
```

Column and Row Redundancy

The SRAM supports replacement of a single column and a single row. The RCA input defines which column will be replaced. The RRA input defines which row will be replaced. The REDUND input enables the replacement. The SRAM read bitlines define the column structure of the memory, so a fault on one of these lines is sufficient to indicate a bad column. The SRAM read wordlines define the row structure of the memory, so a fault on one of these lines is sufficient to indicate a bad row.

The following example script marks the enhanced flow additions in purple.

```
read_verilog -r ram_redund.v
read_spice -i ram_redund.sp
match_design_ports
set_testbench_pin_attributes -func clock CLK
#
# Identify redundancy control inputs
set_constraint -symbolic_static REDUND
set_constraint -symbolic_static {RCA RRA}
set_symbol_to_pass REDUND
set_symbol_to_pass {RCA RRA}
#
# Identify nets that are replaceable (read bitlines define a column)
set_net_group -i -net {RBL*} read_bitlines
# word bitlines define a row Set_net_group –I –net {RWL*} read_wordlines
#
# Define allowed fault tolerance (1 column can be replaced)
set_stuck_fault_group -i -number 1 -net_group read_bitlines
# Define allowed fault tolerance (1 row can be replaced)
set_stuck_fault_group –i –number 1 –net_group read_wordlines
write_testbench tb.1
verify
report_log
quit
```

Internal Control Latches

The SRAM supports replacement of a single column. The column address is serially loaded through the test interface TIN input by enabling redundancy loading with RSEN set to high and clocking in the four-bit address. The REDUND input enables the replacement. The SRAM read bitlines define the column structure of the memory, so a fault on one of these lines is sufficient to indicate a bad column.

This test requires manual intervention. The normal automated testbench cannot perform the control latch loading. The key technique is to define a custom load sequence during initialization and explicitly tell ESP about the redundancy control symbols.

The following declaration file defines the static symbols as redundancy controls.

```
// testbench_declaration_file: redTdf.v
// Declare static symbols for redundancy
reg [0:3] redundantColumnSym;

// Initialize symbols and define as redundancy controls
// This is what set_symbol_to_pass would define
initial $esp_choose_var(redundantColumnSym);
```

The following initialization file defines the loading sequence:

```
// testbench_initialization_file: redTif.v
// Enable scan input and load of redundancy control
SEN = 1’b1;
RSEN = 1’b1;
integer i;
for (i=0; i<4; i=i+1) begin
 TIN = redundantColumnSym[i];
 assign_buffer_value_CLK;
 #(`CLOCKPHASE);
 CLK = 1'b0;
 #(`CLOCKPHASE);
 CLK = 1'b1;
end
```

The following example script marks the enhanced flow additions in purple:

```
read_verilog -r ram_redund.v
read_spice -i ram_redund.sp
match_design_ports
set_testbench_pin_attributes -func clock CLK
#
# Setup up redundancy controls
set_app_var testbench_declaration_file redTdf.v
# Use custom load sequence
set_app_var testbench_initialization_file redTif.v
# Additional control input for redundancy
set_constraint –symbolic_static REDUND
set_symbol_to_pass REDUND
#
# Identify nets that are replaceable (read bitlines define a column)
set_net_group -i -net {RBL*} read_bitlines
#
# Define allowed fault tolerance (1 column can be replaced)
set_stuck_fault_group -i -number 1 -net_group read_bitlines
write_testbench tb.1
verify
report_log
quit
```

A

SystemVerilog Support

The ESP tool support for SystemVerilog is based upon IEEE Standard 1800-2005. ESP has limited support for SystemVerilog Boolean assertions and some SystemVerilog data types. The *FAQ SystemVerilog* man page lists the details.

The following sections describe the use of SystemVerilog language parser, data types, and assertions supported in ESP:

Accessing SystemVerilog Language Parser

To use SystemVerilog with the ESP tool, you must use a different language parser. This parser is the same parser that the Synopsys VCS simulator uses. You can access this parser using a command line option or a UNIX environment variable.

To change the way you invoke the ESP tool, use the `-sverilog` option of the `read_verilog` command.

Command Line

To invoke the ESP shell using the VCS-based language parser, use the following command:

```
% esp_shell -parser_vcs ...
```

For example,

```
% esp_shell -parser_vcs -f run.tcl
```

Environment Variable

Alternatively, to access the VCS-based language parser, set the UNIX environment variable ESP_PARSER to VCS before starting the ESP shell. You can set this variable to ESP or VCS.

For example, you can set the ESP_PARSER variable as follows:

```
% setenv ESP_PARSER VCS
% esp_shell ...
```

Supported SystemVerilog Data Types

The ESP tool supports the following SystemVerilog data types:

- Two-state SystemVerilog types: `shortint`, `longint`, `byte`, and `bit`
- Four-state SystemVerilog types: `logic`, `reg`, `integer`, and `time`. This data type can have X and Z values as well as 0 and 1.

The tool supports `shortreal` as a `real` data type.

SystemVerilog Assertions

The ESP tool supports the following SystemVerilog assertions:

- Boolean assertions using `assert` and `assume` statements
- Treats `assert` statements at the top-level of a design as constraints
- Treats all `assume` statements as constraints
- Single clock assertions
- The system functions `$onehot`, `$onehot0`, `$isunknown`, and `$countones`
- Typed formal arguments in property declarations
- The overlapping implication operator `|->`
- Concurrent assertions in procedures and modules

The ESP tool does not support sequential expressions for `assert` or `assume` statements. The tool does not support `sequences` and `expect` statements.

SystemVerilog Interpretation of the ** Operator

The ESP tool interprets the ** (power) operator according to the SystemVerilog definition. In some cases, the SystemVerilog definition produces different results compared to the Verilog 2001 definition.

For example, consider the following code:

```
module test ;
real ff_1 = (10**12);
real ff_2 = (10**3.1);
initial begin
$display ("ff_1 is %f", ff_1);
$display ("ff_2 is %f", ff_2);
end
endmodule
```

Verilog-2001-compliant implementations produce the following output:

```
ff_1 is 1000000000000.000000
ff_2 is 1258.925412
```

SystemVerilog-compliant implementations produce the following output:

```
ff_1 is -727379968.000000
ff_2 is 1258.925412
```

SystemVerilog Design Construct Support

The ESP shell supports the following SystemVerilog features:

- SystemVerilog C-style increment/decrement, prefix/suffix syntax
  - `i++`
  - `++i`
  - `i--`
  - `--i`
- SystemVerilog assignment operators
  - `+=`
  - `-=`
  - `*=`
  - `/=`
  - `%=`
  - `^=`
  - `&=`
  - `<<=`
  - `>>=`
  - `<<<=`
  - `>>>=`

- SystemVerilog array slice assignments as shown in the following example:

```
module test;
reg [6:0] m[3:0][3:0];
reg [6:0] d;
reg [6:0] e[3:0];
integer r,c, b;
reg t;

 initial begin
      t=0;
      for (r=0; r<4; r=r+1) begin
           t=!t;
           for (c=0; c<4; c=c+1) begin
                for (b=0; b<7; b=b+1)
                begin
                     m[r][c][b] = t;
                     t=!t;
                end
           end
      end
      #10; $display($time()," Start");
      $monitor("%b [%b][%b][%b][%b]",d,e[0],e[1],e[2],e[3]);
      // assignments to e only legal in SystemVerilog
      // illegal in Verilog 2005 and earlier
      #10 d = m[0][0]; e=m[0];
      #10 d = m[1][1]; e=m[1];
      #10 d = m[1][0]; e=m[2];
      #10 d = m[0][1]; e=m[3];
 end
endmodule
```

Usage of break and continue Statements in Loops

The ESP tool makes use of the SystemVerilog `break` and `continue` statements in loops.

To use the break and continue statements in loops, you must do the following:

- Use the ESP VCS parser:

  ```
  esp_shell -parser_vcs
  ```

- Specify SystemVerilog parsing as part of the `read_verilog` command using the `-vcsoption` to specify the `-sverilog` argument:

  ```
  read_verilog -vcs { -sverilog }
  ```

Support for the assert #0 and assert final Statements

The ESP shell supports the SystemVerilog `assert #0` and `assert final` statements:

```
module top;
reg a;
wire not_a;
initial begin
a = 1'b1;
#10 $finish;
end
assign not_a = !a;
always_comb begin : b1
a1: assert final (not_a != a) $display("a1 passed at time %t", $time);
else $display("a1 failed at time %t", $time);
end
endmodule // top
```

Unsupported SystemVerilog Constructs

For a detailed list of supported and unsupported constructs, see the man page *FAQ SystemVerilog*.

B

Using the Open Verification Library With ESP

This appendix describes how to use the Accellera Open Verification Library (OVL) with the ESP tool in the following sections:

Open Verification Library Overview

OVL consists of a group of assertion checkers that Accellera has developed to verify design properties. Assertion checkers are instances of modules that you insert in your design. Verification tools use these checkers to guarantee that your design functions as intended.

Design, integration, and verification engineers use OVL assertion checkers to check design behavior during simulation, emulation, and formal verification. Accellera has implemented the OVL standard in Verilog, VHDL, SystemVerilog, and PSL (Verilog flavor) HDL languages. The ESP tool supports only the Verilog implementation of OVL.

Use the latest OVL version. Download the latest OVL version from the Accellera website at the following location:

http://www.accellera.org/activities/committees/ovl

Recommended OVL Flow

To enable the ESP tool to use OVL assertion checker libraries, do the following:

1. Download the OVL library files from the Accellera website at the following location:

   http://www.accellera.org/activities/committees/ovl

   These are public files and are free to use.

2. In your ESP setup file, specify the OVL directory and the path to all the OVL source files.

3. Within the OVL std_ovl_task.h header file, include a call to the `$esp_error()`task.

   The code for the call to the `$esp_error()`task is shown in red in Example B-1 on page B-3.

   Insert this code after the `OVL_MAX_REPORT_ERROR` conditional statement, which is near line 35 in the 2.4 release of the OVL. This modification enables the ESP tool to generate counter example vectors when a failure is detected. Without this change the assertions still work but counter example vectors are not created. Counter example vectors help you debug failures by providing data on the conditions that caused the failure.

   Note that you can also enable the tool to generate counter example vectors from the command line. For details, see “Generating Counter Example Vectors From the Command Line” on page B-3.

4. Include a reference to the assertion checker from within your Verilog design test code. For details, see “Reporting Assertion Errors.”

Example B-1 Modifying OVL_MAX_REPORT_ERROR to Create Counter Example Vectors

```
`ifdef OVL_MAX_REPORT_ERROR
     if (error_count < `OVL_MAX_REPORT_ERROR)
   `endif
   `ifdef ESP_OVL_FAIL_VECTOR
     begin
       $esp_error("ERROR: ovl_error_t");
   `endif
       case (property_type)
         … <code not changed> …
       endcase
   `ifdef ESP_OVL_FAIL_VECTOR
     end
   `endif
```

Generating Counter Example Vectors From the Command Line

To enable the tool to generate counter example vectors from the command line, modify the std_ovl_task.h file (as shown Example B-1), place it in your current working directory, and execute the command shown in Example B-2.

Example B-2 Generate Counter Example Vectors From the Command Line

```
espcv > test.v +define+ESP_OVL_FAIL_VECTOR  -l log +incdir+my_OVLPATH/std_ovl   \
                                           -y my_OVLPATH/std_ovl +libext+.v
```

Note:

The command in Example B-2 also works if you have replaced the original OVL std_ovl_task.h file with the modified version in Example B-1.

Using Assertion Checkers

To use OVL assertion checkers, include a reference to the checker from within your Verilog design test code, as shown in “Reporting Assertion Errors.” For more information, see the OVL documentation from Accellera.

Differences Between OVL Library Versions V2 and V1

The Accellera standard Open Verification Library is an evolving language standard that is likely to be added to the IEEE standards. Version V2 is a superset of V1 and is backward compatible with V1.

The interface name formats of the OVL V2 assertion checkers differ from those of OVL V1. The older V1 interface uses the `assert_`**`checkerName`** name format. The newer V2 interface uses the `ovl_`**`checkerName`** name format.

The V2 version adds parameters and interface ports and is the recommended format.

To help understand the version differences, consider the V1 `assert_always` task and the V2 `ovl_always` task. Both tasks perform the same check but have slightly different interface requirements. The `ovl_always` task adds an assertion fire output port, specified as `fire`, and a clock enable input port, specified as `enable`. These differences are shown in Example B-3.

Example B-3 OVL Library Differences

```
assert_always ichk (clock, reset, test_expr); // V1 version
ovl_always ichk (clock, reset, enable, test_expr, fire); // V2 version
```

Reporting Assertion Errors

Assertion failures are reported in the ESP run log with message formats that look similar to Example B-4.

Example B-4 Assertion Failure Report

```
OVL_ERROR : OVL_ALWAYS : VIOLATION : Test expression is FALSE : severity
1 : time 10 : test.ichk.ovl_error_t
```

Example B-5 shows the command line to test the `ovl_always` function. To use the ovl_always assertion checker in your design, replace *`MY_OVLPATH/`* with the path to your OVL libraries.

Example B-5 Command Line to Test the ovl_always Function in ESP

```
espcv  test.v +define+ESP_OVL_FAIL_VECTOR  -l log +incdir+MY_OVLPATH/std_ovl   \
                                           -y MY_OVLPATH/std_ovl +libext+.v
```

Use the source code in Example B-6 to test assertions in the `ovl_always` function in ESP. This example shows how symbols are used and how they can create failing test vectors in the ESP tool.

Example B-6 Source Code to Test Assertions in the ovl_always Function in ESP

```
module test();  // Example OVL Usage
`define OVL_ASSERT_ON
`define OVL_COVER_DEFAULT `OVL_COVER_NONE
`define OVL_XCHECK_OFF
`define OVL_IMPLICIT_XCHECK_OFF
`include "std_ovl_defines.h"
reg   risingclock, reset_n, enable, test_expr;
wire [`OVL_FIRE_WIDTH-1:0] fire;
reg sym0,sym1; // Symbols used in simulation

ovl_always #(`OVL_ERROR) ichk (risingclock, reset_n, enable, test_expr, \
                               fire);
initial  $esp_var(sym0,sym1);
initial begin
  $monitor($stime,
      " risingclock=%b, reset_n=%b, enable=%b, test_expr=%b, \
        fire=%b", risingclock, reset_n, enable, test_expr, fire);
  risingclock = 0; test_expr   = 0;
  reset_n = 1;  enable = 1;
  #10 risingclock = 1;
  #10 risingclock = 0;
  #5  test_expr = sym1;
  #5  risingclock = 1;
  #10 risingclock = 0;
  #5  test_expr = sym0;
  #5  risingclock = 1;
  #10 risingclock = 0;
  #5  test_expr = 1;
  #5;  #5  risingclock = 1;
  #10 risingclock = 0;
  #5  enable = 0;  // Disable clock
  #5;  #5  risingclock = 1;
  #10 risingclock = 0;
  #5  $finish(0);
end
endmodule
```

The code in Example B-6 uses assertions to check the `ovl_always` function in the ESP tool. Example B-7 shows the test results.

Example B-7 Output of Assertion Tests of the ovl_always Function in ESP

```
0 risingclock=0, reset_n=1, enable=1, test_expr=0, fire=00x
       OVL_ERROR : OVL_ALWAYS : VIOLATION : Test expression is FALSE : severity 1 :
time 10 : test.ichk.ovl_error_t
        10 risingclock=1, reset_n=1, enable=1, test_expr=0, fire=001
        20 risingclock=0, reset_n=1, enable=1, test_expr=0, fire=001
        25 risingclock=0, reset_n=1, enable=1, test_expr=s, fire=001
       OVL_ERROR : OVL_ALWAYS : VIOLATION : Test expression is FALSE : severity 1 :
time 30 : test.ichk.ovl_error_t
        30 risingclock=1, reset_n=1, enable=1, test_expr=s, fire=00s
        40 risingclock=0, reset_n=1, enable=1, test_expr=s, fire=00s
        45 risingclock=0, reset_n=1, enable=1, test_expr=s, fire=00s
       OVL_ERROR : OVL_ALWAYS : VIOLATION : Test expression is FALSE : severity 1 :
time 50 : test.ichk.ovl_error_t
        50 risingclock=1, reset_n=1, enable=1, test_expr=s, fire=00s
        60 risingclock=0, reset_n=1, enable=1, test_expr=s, fire=00s
        65 risingclock=0, reset_n=1, enable=1, test_expr=1, fire=00s
        75 risingclock=1, reset_n=1, enable=1, test_expr=1, fire=000
        85 risingclock=0, reset_n=1, enable=1, test_expr=1, fire=000
        90 risingclock=0, reset_n=1, enable=0, test_expr=1, fire=000
       100 risingclock=1, reset_n=1, enable=0, test_expr=1, fire=000
       110 risingclock=0, reset_n=1, enable=0, test_expr=1, fire=000

3 error(s) are found
```

C

Programming Language Interface (PLI)

Programming Language Interface (PLI) is a C language procedural interface that allows you to access and modify data in the Verilog HDL source descriptions and traverse the design hierarchy. PLI provides a library of C language functions that can directly access data within a design.

PLI is part of the IEEE standard. The standard is described in the 1995 Verilog Language Reference Manual (IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language; IEEE Standard number: IEEE Std 1364-1995).

This section describes the scope of PLI function support specific to the ESP tool. The ESP tool does not accept or return symbolic equations; it requires inputs to the PLI functions to be binary signals. The tool can use PLI functions to initialize programmable memory elements or to initialize binary values in other situations.

Note:

Support is available through SolvNet online customer support and by contacting the Synopsys Technical Support Center.

This section includes the following subsections:

PLI Routines

The 1995 Verilog IEEE standard describes the PLI routines in terms of three generations:

- Task or function (TF) routines
- Access (ACC) routines
- Verilog Procedural Interface (VPI) routines

Task or Function (TF) Routines

These routines are used primarily for operations involving user-defined system task or function arguments and for utility functions. The TF routines are also known as utility routines.

Access (ACC) Routines

These routines provide access into a Verilog HDL structural description. You use ACC routines to access and modify information such as delay values and logic values on a wide variety of objects in a Verilog description.

ESP ACC routine support extends to 2-D arrays and memories. This extension works in the same way as the VPI support for 2-D arrays and memories. The data value structures read and written to ACC functions are the same as the similar VPI functions. Note that the ESP ACC extension is not part of the 1995 Verilog IEEE standard; whereas, the VPI access to 2-D arrays and memories is part of the 1995 Verilog IEEE standard.

Verilog Procedural Interface (VPI) Routines

These routines provide an object-oriented access to both Verilog HDL structural and behavioral objects. The ESP tool does not support VPI routines.

What the ESP Tool Provides

The ESP tool provides the following components:

- Set of TF routines
- Set of ACC routines
- Source file named veriuser.c that contains the task registration array
- Two header files named veriuser.h and acc_user.h

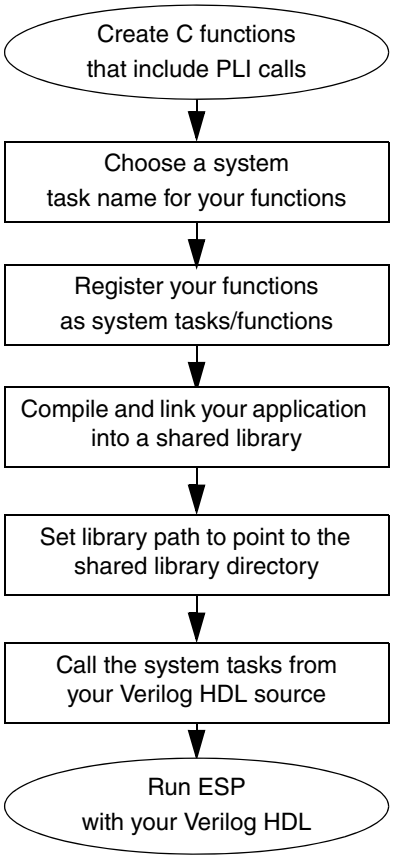
Note:
ACC routines related to `specify` blocks and VPI routines are not supported by ESP.

How to Use PLI Within the ESP Tool

To use PLI routines within the ESP tool, do the following:

1. Create C functions that contain the calls to the PLI routines. The ESP tool calls these functions when the associated user-defined system task is executed.
2. Create a user-defined system task or function name that is associated with each C function.
3. Record the user-defined system task or function with the corresponding C function in a table within the veriuser.c file.
4. Compile the C functions and veriuser.c file dynamically and create a libpli.so library file.
5. Set the library path to point to the shared library directory.
6. Call any of the C functions by calling the corresponding user-defined system task or function in the Verilog HDL design.

Figure C-1 Using PLI Functions With ESP Verilog Simulator Task Flow



To load the shared library into the ESP tool, you must set the LD_LIBRARY_PATH environment variable to include the directory with the shared library. You must also use the `-libpli1` command line option if either of the following is true:

- The shared library has a name other than libpli.so.
- The shared library requires the ESP tool to call a bootstrap function to define the PLI tasks or functions.

PLI Example

A PLI example, demo_1, is provided with the ESP tool. You can use the source files in the example as templates to create your own PLIs. The example is located in the INSTALL_DIR/doc/esp/demo/pli/demo_1 directory. To use it, run the tool on Linux OS.

The following files are located in the demo_1 directory:

- README
- test.v

  The Verilog design that references the user-defined system tasks and functions.
- test.c

  The code in this file uses PLI routines when called by the test.v file to print out the submodules of the top-level modules. The veriuser.h and acc_user.h files are included in this code to define all the required PLI variables and PLI routines. These include files in the release at INSTALL_DIR/auxx/esp/pli/include.
- veriuser.c

  This file contains `veriusertfs[]`, a data structure that specifies the user-defined PLI routines used in this example. A template source for this file is located in the INSTALL_DIR/auxx/esp/pli/src/veriuser.c file.
- base.log

  This file contains the expected output of the Verilog simulation based on the test.v file.
- makefile

  This file contains the makefile for this example. A template source for this file is located in the INSTALL_DIR/auxx/esp/pli/src/makefile directory.

To run the demo, do the following:

1. Create the libpli.so shared library file for the user-defined system tasks.

   ```
   gmake -f Makefile clean
   gmake -f Makefile INNOHOME=INSTALL_DIR
   ```

2. Simulate with the user-defined tasks present in the libpli.so file. The dopli file sets the `LD_LIBRARY_PATH` environment variable to the current directory so that it locates the libpli.so file that you created.

   ```
   cp INSTALL_DIR/auxx/esp/pli/src/dopli .
   source dopli
   espcv test.v
   ```

After running the `espcv` command, an output log file, esp_run.log, is created in your directory. The contents of the log file should be similar to the base.log expected output file.

Using Dynamically Linked PLI Libraries

The following is an example of a dynamically linked PLI library:

`-libpli1="`*`library`*"

-libpli1="*`library:bootstrap_function`*"

The `-libpli1` command-line option dynamically loads a specified PLI 1.0 library. The *`library`* name specifies a library that contains a PLI 1.0 application. The *`bootstrap_function`* name corresponds to a function that returns a pointer to a `veriusertfs` array. When a *`bootstrap_function`* name is not given, it is assumed that there is a `veriusertfs` array in the library.

The following example shows how to use the `-libpli1` option:

```
esp test.v -libpli1="../commonlib_dir/mylibpli.so:myfunc"
```

The ESP tool dynamically opens the library file mylibpli in the ../commonlib_dir/ directory. It executes the myfunc() function in the library, and treats the return value of `myfunc()` as if it is from the `veriusertfs` variable. The *`bootstrap_function`* must be used if more than one dynamic `library` is required.

D

History of Features and Enhancements

This section provides a list of features and enhancements added in the following ESP releases:

Features and Enhancements in Version K-2015.12

The K-2015.12 release comprised the following features and enhancements:

Verdi Debug Cockpit

The ESP tool allows Interactive Signal Tracing (IST) debugging capabilities through the Verdi® interface. When you enable IST, an ESP menu is added to the Verdi graphical user interface (GUI) pull-down menu. You can analyze ESP mismatch or power integrity violation (PIV) vectors within the IST environment and highlight SPICE design nodes in a schematic built from the native SPICE netlist.

For more information, see “Verdi Debug Cockpit” in Chapter 9.

Power Integrity Verification Enhancements

The power integrity verification (the `set_verify_mode inspector` command) in the ESP shell has the following new capabilities:

- Integration with the Verdi Debug Cockpit: The Verdi Debug Cockpit schematic view highlights power integrity verification (PIV) violating paths.
- Enhanced violation reporting: Enhanced violation reporting enables you to view the subsets of the violations that are of interest. For example, a SHORT violation detailed report includes the transistors that you turned on to provide a conduction path from power to ground.

  For more information and examples, see “Reporting Options” in Chapter 11.

Debugging Passed Designs at Will

Starting with the K-2015.12 release, you can run a binary debug simulation in the ESP tool at any time, if you supply a vector file to be used for the run. Therefore, if you save an error vector file you can rerun that binary trace even if you resolve all the errors in the design. This allows you to verify whether a specific error sequence works and examine the internal waveforms that result from the error sequence.

For more information and examples, see “Debugging Passed Designs at Will” in Chapter 9.

Additional Device Information for the print_net_trace Command

The `print_net_trace` command is enhanced to show the transistor instance name, subcircuit name, and path for each pull-up and pull-down device.

For example, the `print_net_trace` command output previously displayed the following:

```
pmos (1.50/0.18, r=3.42, g=inno_tb_top.xtor.XI6.XR7.XDOUT=0)
```

The `print_net_trace` command output now displays the following:

```
pmos (1.50/0.18, r=3.42, g=inno_tb_top.xtor.XI6.XR7.XDOUT=0,
      i=MP5, s=SENSEAMP, path=inno_tb_top.xtor.XI6.XR7)
```

Support for the -parameters Command Line Option

The ESP tool is enhanced to support the `-parameters` option when using the VCS® parser. Use the `read_verilog -parameters` option to specify a parameter file that provides settings for Verilog source parameters. This option is similar to the `-pvalue` option.

For more information and examples, see “Reading in Reference Designs” in Chapter 5.

Stop on Randomize Now True by Default

The default of the `verify_stop_on_randomize` application variable is changed to `true` so that the simulation stops if randomization occurs and you are explicitly aware of the randomization.

For more information and examples, see “Stopping Randomization by Default” in Chapter 9.

Obsolescence of ESP ModelGen and Direct SPICE Read Methodologies

The ESP tool no longer supports the `espmodelgen` and `set_process -f` *`models.inc`* commands.

You can use the device model simulation approach instead. For more information, see the `device_model_simulation` flow man page.

Obsolescence of Formality ESP

The ESP installation no longer includes executables for Formality® ESP.

Planned Obsolescence of the ESP Shell GUI

Starting with the L-2016.06 release, the GUI (the `esp_shell -gui` command, or the `start_gui` command within the ESP shell) is no longer supported and will not be available.

Features and Enhancements in Version K-2015.06

The K-2015.06 release comprised the following features and enhancements:

- Library Verification in the ESP Shell
- Power Integrity Verification Enhancements
- SPICE-to-SPICE Equivalence Checking
- Device Model Simulation Enhancements
- Merge and Report Coverage Results From Previous Sessions
- Verilog Specify Block Now On By Default
- Formality ESP Delay Rounding Value Change
- Single Key for Power Integrity Verification and Redundancy Validation
- Obsolescence of the SPARC Platform
- Planned Obsolescence of the Graphical User Interface

Library Verification in the ESP Shell

You can verify cell libraries as a large group of matched designs rather than as separate, individual cell equivalence checks in the ESP shell. This brings the capabilities of the Formality ESP tool into the ESP shell.

The library verification flow is as follows:

1. Set defaults
2. Set technology
3. Read designs
4. Match designs
5. Match ports
6. Set design port attributes
7. Set constraints
8. Verify

The following new commands are added to the tool to perform the tasks in the library verification flow:

- `set_verification_defaults`
- `read_db`
- `match_designs`
- `set_matched_designs`
- `set_matched_design_attributes`
- `report_matched_design_attributes`
- `remove_matched_designs`
- `report_matched_designs`
- `report_unmatched_designs`
- `get_matched_designs`

In addition, many commands now have a new `-matched_designs` option to support cell library verification.

For more information about these commands, see the man pages. For guidance on how, why, and when to use the commands, see the man page for the library verification flow and Chapter 10, “Cell Library Verification in the ESP Tool.”

Note:

If your existing Tcl scripts use shortened forms of the library verification commands, the abbreviated command might now be ambiguous and need to be changed. For example, if you use `match` instead of the `match_design_ports` command, the abbreviation becomes ambiguous between the `match_design_ports` command and the new `match_designs` command. You should use the complete name of the commands, options, or switches in any Tcl script.

For example, instead of `set_testbench_pin_attributes CLK -func Clock`, use the following:

```
set_testbench_pin_attributes CLK -function Clock
```

Power Integrity Verification Enhancements

Power integrity verification (`set_verify_mode inspector`) in the ESP shell has two new capabilities:

- Enable reporting of violation subsets

  Use the following new options of the `report_inspector_results` command to view the subsets of the violations:

Option	Description
`-checker <list>`	Violations for listed checkers
`-design <subcircuit>`	Violations in the subcircuit
`-net <net>`	Violations for the specified net in `-design <subcircuit>`
`-greater_than <number>`	Violations greater than the specified number
`-less_than <number>`	Violations less than the specified number
`-greater_or_equal <number>`	Violations greater than or equal to the specified number
`-less_or_equal <number>`	Violations less than or equal to the specified number
`-range <list>`	Violations between the range specified

Option	Description
`-power <power>`	Power (in mW)
`-energy <energy>`	Energy (in pJ)
`-duration <time>`	Lasting time (in ns)
`-time <time>`	Occurring time (in ns)
`-details`	Provide the detailed report for each violation
`-tbid <testbench_id>`	Report the specified testbench
`-nworst <number>`	Report the worst `<number>` of violations

You can specify multiple options with the `report_inspector_results` command.

The following sets of options are mutually exclusive:

- ```
  -greater_than
  -less_than
  -greater_or_equal
  -less_or_equal
  -range
  ```

- ```
  -power
  -energy
  -duration
  -time
  ```

- View and analyze violations in a subsequent ESP Shell session

  Use the `save_session` command before quitting the current session, and use the `restore_session` command as the first command for the subsequent session.

```
esp_shell
esp_shell> set_verify_mode inspector
esp_shell> read_verilog -r ...
...
esp_shell> verify
esp_shell> report_inspector_results
esp_shell> save_session my_design_piv
esp_shell> quit

esp_shell
esp_shell> restore_session my_design_piv
esp_shell> report_inspector_results -id 233
esp_shell> quit
```

SPICE-to-SPICE Equivalence Checking

The ESP shell supports direct SPICE-to-SPICE equivalence checking. Previously, this could only be accomplished by comparing the SPICE implementations to a common Verilog design in two separate verifications. With this release, a direct equivalence check is supported (without a Verilog design) by reading one of the designs into the reference container as follows:

```
set_process -r -technology_file process1.edm
read_spice  -r design1.spi
set_instance_strength_multiplier -r 1.5 -design OUTBUF MP3

set_process -i -technology_file process2.edm
read_spice  -i design2.spi
set_instance_strength_multiplier -i 0.4 -design INV23 MI5

match_design_ports
...
...
```

Device Model Simulation Enhancements

The `add_device_model` command has three new options: `-llist`, `-wlist`, and `-flist`. The options allow a Tcl list of length values, width values, and fin numbers to be specified for characterization. For example,

```
add_device_model -voltage 1 -ntype nf -ptype pf -llist 18n -flist {1 2 4}
```

The `-pn` option has been removed. If you are using the `-pn` option in an existing script, remove it and characterize the PFET and NFET devices separately.

The characterization algorithms are also improved. To produce .edm files that are characteristically closer to SPICE models, rerun previous device model simulation scripts with the K-2015.06 release.

Merge and Report Coverage Results From Previous Sessions

The ESP shell allows coverage merging and reporting on coverage information from simulations performed in previous esp_shell sessions. The new `-files` option of the `report_coverage` command takes a Tcl list of the coverage database files you are interested in, merges them, and produces a report.

For example, if you simulate three testbenches in separate subdirectories (VT.bin.dir, VT.ptl.dir, VT.dit.dir), you can use the following command in a subsequent esp_shell session

to merge and report coverage information, irrespective of using the `save_session` command:

```
report_coverage -spec cov.spec.mod2 -files {
      VT.bin.dir/ESP_WORK/ram1r1w/tb0/tb.cov \
      VT.ptl.dir/ESP_WORK/ram1r1w/tb0/tb.cov \
      VT.dit.dir/ESP_WORK/ram1r1w/tb0/tb.cov \
}
```

Verilog Specify Block Now On By Default

The ESP shell and the Formality ESP tools now read and use Verilog specify blocks by default during simulation and match the VCS tool's behavior. In previous versions, you could use Verilog specify blocks by setting the `verify_use_specify` application variable to `true`, but this was not the default.

If you do not want to use Verilog specify blocks, include the following in your Tcl script before the `verify` command:

```
set_app_var verify_use_specify false
```

Otherwise, use the `+nospecify` VCS argument with the `read_verilog` command:

```
read_verilog -r -vcs "ram.v +define+MODE1 +nospecify"
```

Formality ESP Delay Rounding Value Change

Starting with the K-2015.06 release, the Formality ESP RC delay rounding value is now 1 ps (compared to 10 ps in previous versions). For Formality ESP simulations, this value cannot be changed.

Single Key for Power Integrity Verification and Redundancy Validation

The ESP shell now uses a single ESP-CV license key for power integrity verification (with the `set_verify_mode inspector` command) and redundancy validation (with the `set_stuck_fault_group` command). Previously, these ESP shell capabilities required two separate ESP-CV license keys.

Obsolescence of the SPARC Platform

The ESP installation no longer includes executables for the SPARC platform.

Planned Obsolescence of the Graphical User Interface

Starting with the L-2016.06 release, the GUI (the `esp_shell -gui` command, or the `start_gui` command within the ESP shell) will no longer be supported and will not be available.

Features and Enhancements in Version J-2014.12

The J-2014.12 release comprised the following features and enhancements:

Obsolescence of the SPARC Platform

The ESP installation no longer includes executables for the SPARC platform.

The tool issues the following warning message if you invoke the ESP SPARC executable:

```
WARNING: As of the K-2015.06 version of ESP, the SPARC platforms will no
longer be supported
```

Single License Required for the set_symbol_to_pass Command

The `set_symbol_to_pass` command requires only one ESP-CV license. Previously, the `set_symbol_to_pass` command required two ESP-CV licenses to be available as part of the redundancy validation capability of ESP shell. The `set_stuck_fault_group` command still requires two ESP-CV licenses during an ESP shell session.

Support for break and continue in Loops

The ESP tool supports the use of the SystemVerilog `break` and `continue` statements in loops.

For more information, see “Usage of break and continue Statements in Loops” in Appendix A.

Support for the Command Line -timescale Option

You can use the -timescale argument with the `read_verilog -vcs` command to specify the timescale for Verilog source files before the first occurrence of the `timescale directive.

For more information, see “Specifying the Timescale for Verilog Source Files” in Chapter 3.

Support for SystemVerilog assert #0 and assert final Statements

The ESP shell supports the SystemVerilog `assert #0` and `assert final` statements.

For more information, see “Support for the assert #0 and assert final Statements” in Appendix A.

Support for the $clog2() System Function

The ESP tool supports the `$clog2()` system function that returns the ceiling function of the log base 2 of the argument (the log rounded up to an integer value). The argument can be an integer or an arbitrarily sized vector value. The argument is treated as an unsigned value, and an argument value of 0 produces a result of 0.

This system function can be used to compute the minimum address width necessary to address a memory of a given size or the minimum vector width necessary to represent a given number of states as follows:

```
integer result;
result = $clog2(n);
```

SystemVerilog Interpretation of the ** Operator

The ESP tool interprets the ** (power) operator according to the SystemVerilog definition. In some cases, the SystemVerilog definition produces different results compared to the Verilog 2001 definition.

For more information, see “SystemVerilog Interpretation of the ** Operator” in Appendix A.

Verilog-to-SPICE Optimization Enabled by Default

The `pi_mode` application variable controls an optimization for transistor RC delay analysis. The default for `pi_mode` is `on`.

If you do not want pi mode optimization during verification, set the value to `off` as follows:

```
set_app_var pi_mode off
```

Features and Enhancements in Version J-2014.06

The J-2014.06 release comprised the following features and enhancements:

- Formality ESP Device Model Simulation Support
- Infinite Decay Time Using the set_rcdecay_time Command
- Parameterized RTL Model Support
- Power-Up Reinitialization of SPICE Nodes
- SystemVerilog Design Construct Support
- Verilog-to-Verilog Strict Comparison

Formality ESP Device Model Simulation Support

The Formality ESP tool supports device model simulation.

For more information, see “Device Model Simulation” in Chapter 4.

Infinite Decay Time Using the set_rcdecay_time Command

The `set_rcdecay_time` command allows infinite decay time, essentially disabling the RC decay and treating all SPICE nets as Verilog trireg nodes. The `set_rcdecay_time` command is used to specify decay time for all SPICE nets when no driver is active.

For more information, see “Controlling SPICE RC Decay Time” in Chapter 4.

Parameterized RTL Model Support

The ESP shell supports parameters for top-level Verilog designs. The parameters for a top-level Verilog module can be specified using the VCS `-pvalue` option. Parameter overrides are supported for any Verilog design.

For more information, see “Parameterized RTL Model Support” in Chapter 3.

Power-Up Reinitialization of SPICE Nodes

The ESP shell allows SPICE design nodes to be explicitly reinitialized after a power-down then power-up sequence during simulation. The new `set_net_value` command allows you to set a value on a net to avoid any incorrect values after a power-down then power-up sequence.

For more information, see “Power-Up Reinitialization of SPICE Nodes” in Chapter 4.

SystemVerilog Design Construct Support

The ESP shell supports SystemVerilog features such as increment/decrement, prefix/suffix syntax, assignment operators, and array slice assignments.

For more information, see “SystemVerilog Design Construct Support” in Appendix A.

Verilog-to-Verilog Strict Comparison

The default output comparison for Verilog-to-Verilog equivalence checking is stricter. You can still use the `set_testbench_pin_attributes` command to change the `-checker` option to any of its available values.

For more information, see “Verilog-to-Verilog Equivalence Checking” in Chapter 3.

Features and Enhancements in Version I-2013.12

The I-2013.12 release comprised the following features and enhancements:

- Display of Input Delay in the Simulation Report
- Enhancements to Device Model Simulation Commands
- Performance Improvement for Verilog-to-SPICE Verification
- Support for multiphase input signals

- Support for Octal and Hexadecimal Display Values
- Synopsys Diagnostic Platform Variable
- Obsolescence of AIX Platform

Display of Input Delay in the Simulation Report

The report generated by the `report_log` command distinguishes inputs that are delayed using the `set_input_delay` command from inputs that change at the default times, such as the start of a testbench cycle. The report displays `@+<delay>` after the input value of the signal.

For more information, see "Specifying Testbench Input Delay" in Chapter 7.

Enhancements to Device Model Simulation Commands

The ESP Shell tool is enhanced to support FineSim as a simulator for device model simulation. The `create_model_library` command now supports the FINESIM value for the `-simulator` option.

For more information see "Device Model Simulation" in Chapter 4.

Performance Improvement for Verilog-to-SPICE Verification

The ESP Shell tool is enhanced to provide a 2x performance improvement for Verilog-to-SPICE verifications compared with the H-2013.06 release version.

Support for multiphase input signals

The ESP Shell tool provides additional symbolic testbench support for latch-based designs and designs that associate some signals with one phase of the clock and other signals with the other phase of the clock.

For more information see "Latch-Based Designs" in Chapter 7.

Support for Octal and Hexadecimal Display Values

The ESP Shell tool is enhanced to enable control of the display radix for testbench pins. The ESP Shell tool now supports the octal and hexadecimal display radix. The default display radix of the ESP created testbenches is binary. To change the display radix, use the new `-radix` option of the `set_testbench_pin_attributes` command.

For more information, see “Output Display Radix” in Chapter 7.

Synopsys Diagnostic Platform Variable

The Synopsys Diagnostic Platform (SDP) library can make it easier to determine what has happened when a Synopsys product is used. For example, the generated information might help explain why an ESP tool simulation stops and produces a stack trace. This library is integrated into ESP products.

Starting with the I-2013.12 release, you can enable the Synopsys Diagnostic Platform (SDP) library across all ESP executables when you set the `ESP_SDP_ENABLED` environment variable to 1. For example,

```
% setenv ESP_SDP_ENABLED 1
```

When you set the variable, the ESP tool execution generates its regular output along with an sdp.tgz file that can be sent to the Synopsys R&D team. Subsequent runs in the same directory generate sdp_1.tgz, sdp_2.tgz, and so on. You need not set the variable unless you encounter a problem.

Obsolescence of AIX Platform

The ESP installation does not include executables for the AIX platform.

Features and Enhancements in Version H-2013.06

The H-2013.06 release comprised the following features and enhancements:

- Changes to the Read/Write Mask Values
- Device Model Simulation
- Enhancement to Testbench Input Delay Specification
- Obsolescence of 32-bit Executables and the Solaris x86 Platform
- Support of Sub-Picosecond Timing Delay in RC Analysis

Changes to the Read/Write Mask Values

The ESP tool handles pins specified with the `writemask`, `readmask`, or `readwritemask` values in the `-function` option of the `set_testbench_attributes` command differently. This change affects only symbolic (*.sym) and dual-phase (*.2ph) testbenches. The pin functions are now symbolic during fully symbolic cycles. In previous releases, the fully symbolic cycles of these testbenches had a binary value for any pin specified with the `writemask`, `readmask`, or `readwritemask` value.

Device Model Simulation

The ESP Shell tool provides a more precise and convenient method to obtain transistor device information. This information is used to give better results when calculating RC delays model simulation.

For more information, see “Device Model Simulation” in Chapter 4.

Enhancement to Testbench Input Delay Specification

The ESP Shell tool is enhanced to allow design inputs to have their stimulus applied at different times versus other design inputs.

For more information, see “Specifying Testbench Input Delay” in Chapter 7.

Obsolescence of 32-bit Executables and the Solaris x86 Platform

Starting with the H-2013.06 release, the ESP installation does not include the following executables:

- Executables for the Solaris x86 platform
- 32-bit executables for all platforms

If you invoke any ESP 32-bit executable in the current release, the tool issues the following warning message:

```
As of the I-2013.06 version of the ESP tool, the 32-bit version of the
product is not delivered by default. If you require a 32-bit version for
any reason, contact Synopsys technical support.
```

Support of Sub-Picosecond Timing Delay in RC Analysis

The ESP Shell tool now allows sub-picosecond delay values for the SPICE design RC analysis using the new `verify_rcdelay_unit` application variable. In previous releases, the smallest delay precision available was 1 picosecond.

For more information, see “Setting a Simulation Timescale” in Chapter 7.

Features and Enhancements in Version H-2012.12

The H-2012.12 release comprised the following enhancements:

- Coverage Filter
- Delay Calculation Model
- ESP GUI

Coverage Filter

The ESP tool provides better handling of wildcards within Verilog escaped names and improves the filtering of the array module instances. When you use wildcards inside a Verilog escaped name (starts with a backslash, ends with a space), escape the wildcards with a backslash.

Consider the following nets:

```
inno_tb_top.xtor.abc.\xyz[0] .jk]
inno_tb_top.xtor.abc.\xyz[1] .jk]
inno_tb_top.xtor.abc.\xyz[2] .jk]
```

You can filter these nets using any of the following three lines of code:

```
filter NS inno_tb_top.xtor.abc.\xyz[\[0-2\]] .jkl
filter NS inno_tb_top.xtor.abc.\xyz[\*] .jkl
filter NS inno_tb_top.xtor.abc.\xyz[\?] .jkl
```

Escape the arrayed module instance brackets with the filter lines. For example, you can create a Verilog array of modules:

```
module tb;
...
testmod inst[1:0] ({arg1,arg1}, {arg2,arg2});
...
endmodule
module testmod (in1, out1);
...
endmodule
```

To differentiate the arrayed module instances from the wildcards, escape their brackets as follows:

```
filter NS tb.inst\[1\].in1
filter NS tb.inst\[0\].out1
```

Delay Calculation Model

The ESP shell supports the following enhancements to the delay calculation model:

Smallest RC Delay Value

In this release, ESP shell is enhanced to define the smallest RC-mode delay for the SPICE design be 1 picosecond (the “precision” value). This provides a more realistic timing analysis. In previous releases, the smallest RC-mode delay for the SPICE design is 0 ps (no delay at all).

Partial Discharge Threshold Default

The partial discharge threshold determines what are classified as slow transitions. In previous releases, this value has a default of 100 ps.

In this release, the default of the partial discharge threshold is now modified to auto, so that the value depends on the feature size, and improves the simulation results.

However, to use the previous default setting of the partial delay threshold, set the `verify_partial_transition_threshold` variable to 100:

```
set_app_var verify_partial_transition_threshold 100
```

Diffusion Parameter Calculation

During simulation, the drain and source diffusions of a MOSFET add capacitances that must be considered for delay calculations. These capacitances are modeled in the circuit netlist or in the SPICE library models using the parameters such as AD, AS, PD, and PS.

In previous releases, if ESP finds these parameters, it uses them. In this release, if the parameters are zero or absent, ESP uses zero as the value of these parameters.

Diffusion Area, AD = LD*W

Diffusion Perimeter, PD = PS = LD+(2*W)

where, LD is the lateral diffusion, W is the channel width.

Capacitance Calculations in the Direct SPICE Read Method

During Direct SPICE Read, ESP uses the RC values along the paths of the transistors that are turned ON for the delay calculations. In this release, capacitative modeling is enhanced to estimate the delay during the Direct SPICE Read method better.

Resistance Calculations in the Direct SPICE Read Method

ESP uses the sheet resistance model to estimate the resistance of the channel for a transistor that is turned ON. The sheet resistance values are obtained from the MOSFET's I-V curve. In previous releases, this sheet resistance model used a point on the I-V curve which is too far into the saturation, to be the best average resistance value for the transistor.

In this release, the sheet resistance model, used by the Direct SPICE Read method, is enhanced to calculate a more realistic average sheet resistance value.

The transistor-capacitance model includes the following elements:

- Channel overlap capacitance (COV)
- Diffusion sidewall capacitance (CSW)
- Diffusion sidewall gate capacitance (CSWG)

ESP GUI

The ESP GUI supports the following enhancements:

- A new DVE button is added to the Debug Design pane. Click the DV button to invoke the DVE waveform viewer.
- Use the symbolic binary option when executing a Debug Design run.
- The Format and Dumpscope File options are moved into the same row as that of the Binary Debug button on the Debug Design pane.
- The pin attribute function options sort in an alphabetical order.

Features and Enhancements in Version G-2012.06

The G-2012.06 release comprised the following features and enhancements:

- Automated Virtual Supply Recognition
- Modified Startup Banner Platform Keywords for Linux and AMD64

- Input-buffered Signals Split into Two New Independent Signals in Automatically Generated Testbenches
- Renamed Variable verify_suppress_nonzero_delay_osc
- Symbolic Values as Input Signals
- Allowed Verilog UDP as the Top-Level design for Verification

Automated Virtual Supply Recognition

The ESP shell supports automated virtual supply recognition. As in previous releases, you can use the `set_supply_net_pattern` command to specify custom virtual supplies. You can enable ESP shell to automatically determine the virtual supplies by setting the `netlist_virtual_supply_recognition` shell variable to true.

For more information, see “Recognizing Virtual Supplies” in Chapter 7.

Modified Startup Banner Platform Keywords for Linux and AMD64

The product banner displayed during tool startup now shows the Red Hat Enterprise Linux operating system platform keywords as RHEL32 or RHEL64, rather than Linux or AMD64.

This change indicates that these binaries run on the Red Hat Enterprise Linux operating system and not on other operating systems. Only the startup banner has changed. The directory paths and tool behavior are unchanged.

Input-buffered Signals Split into Two New Independent Signals in Automatically Generated Testbenches

Automatically generated testbenches provide two independent input signals to drive the reference and implementation designs. In previous releases, a single internal testbench input signal, <pin_name>_in__ was generated, and if the reference design drove an input port, the driven value propagated to the implementation design and masked the input value applied by the testbench. Similarly, the implementation design could drive the reference input, overpowering the input value applied by the testbench.

From the G-2012.06 release, the internal testbench signal is split into two independent signals <pin_name>_ref and <pin_name>_xtor.

For example, instead of creating net A_in__ for input A, to both the reference and implementation design, the testbench creates internal nets A_ref to the reference design and net A_xtor to the implementation design.

This change in the testbench might impact any verification that uses files from previous releases. Suppose your testbench has modifications in a declaration file that uses the <pin_name>_in__ net name (for example, RE_in__).

If you use this declaration file with the new testbench, the `check_design` command generates the following error message:

```
esp ERROR [PARSE-606]:
... Identifier ( RE_in__ ) not declared in current scope
```

If you have declaration files based on the testbenches generated by previous releases that directly refer to the testbench buffered signals with the _in__ suffix, then these need to be changed to refer to the two new signals, <pin>_ref and <pin>_xtor, if you use these declaration files with a testbench generated with the G-2012.06 release.

Renamed Variable verify_suppress_nonzero_delay_osc

The ESP shell variable `verify_suppress_nonzero_delay_osc` is renamed to `verify_suppress_nonzero_delay_oscillation` to be consistent with the naming conventions of all the variables.

When you modify the `verify_suppress_nonzero_delay_osc` variable setting, an `ESPCMD-052` information message is generated as follows:

```
ESPCMD-052(Info): ‘verify_suppress_nonzero_delay_osc’ is deprecated.
```

Symbolic Values as Input Signals

The ESP shell supports the addition of symbolic values X and Z along with the symbolic values 0 and 1 at the primary inputs of the design with the automatically generated testbenches. In previous releases, ESP shell allowed symbolic values X and Z only with custom testbenches.

For more information, see “Configuring Testbench Pins” in Chapter 7.

Allowed Verilog UDP as the Top-Level design for Verification

The ESP shell is enhanced to allow a Verilog UDP (user-defined primitive) to be used as the top-level design for verification. In previous releases, ESP supported UDP descriptions but required the UDP to be instantiated within a module.

Features and Enhancements in Version F-2011.12

The F-2011.12 release comprised the following features and enhancements:

- Distributing Testbench Execution
- Combinational SystemVerilog Assertions Support
- DRAM Capacitor Cells Support
- Transistor-Only Simulation Support
- Verilog-to-Verilog Equivalence Checking

Distributing Testbench Execution

The ESP Shell is enhanced to run multiple testbenches in parallel. The testbenches can be run on the current machine or across multiple machines, depending upon the settings of three new ESP shell variables.

For more information, see “Running Testbenches in Parallel” in Chapter 8.

Combinational SystemVerilog Assertions Support

The ESP Shell is enhanced to support simple combinational SystemVerilog assertions. To use these assertions, invoke ESP Shell with the new VCS parser by either adding the `-parser_vcs` option to the `esp_shell` command or by setting the UNIX ESP_PARSER environment variable to VCS before issuing the `esp_shell` command.

For more information, see “SystemVerilog Assertions” in Appendix A.

DRAM Capacitor Cells Support

ESP-CV is enhanced to support capacitively driven nets such as those found in DRAM core cells. ESP can now model the storage capacitor as an element that can drive other logic such as sense amplifiers.

Transistor-Only Simulation Support

The ESP Shell is enhanced to allow transistor-only designs to be simulated without requiring a Verilog reference design.

For more information, see “Transistor-Only Simulation” in Chapter 11.

Verilog-to-Verilog Equivalence Checking

The ESP Shell is enhanced to allow Verilog-to-Verilog equivalence checking. Previously, the compare mode (the default mode) required a Verilog design in the reference container and a SPICE design in the implementation container.

For more information, see “Advanced Redundancy Checking Capabilities” in Chapter 8.

Features and Enhancements in Version F-2011.06

The F-2011.06 release comprised the following features and enhancements:

Disable and Retrigger Behavior Modified to Match VCS

The ESP tool follows the VCS semantics to handle event disabling and retriggering within the same always block. To use the previous behavior, set the `verify_imitate_simulator` variable to the value `inno`.

Power Intent Checker Name Change from sneak_path to sneak

The inspector mode checker names and associated variable names are more consistent. The checker name sneak_path was changed to sneak to make it consistent with the `verify_max_number_pwr_sneak_vectors` variable. If you used the sneak_path checker name as part of a list of checkers for the -checker option of either the `set_inspector_rules` or `remove_inspector_rules` commands, you need to change the checker name to sneak in your Tcl scripts. The outputs of the `report_inspector_rules` and `report_inspector_results` commands also reflect this change.

New report_symbols_to_pass Command

You can use the new `report_symbols_to_pass` command to see at least one possible value of the `set_symbol_to_pass` command signals that lets your verification complete without error.

For more information, see “Advanced Redundancy Checking Capabilities” in Chapter 8.

Output Constraint Capability

You can use the `set_constraint` command with the new `-ignore` option to create output constraints.

For more information, see “Specifying Output Constraints” in Chapter 7.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W