
Affirma Equivalence Checker Tutorial

Product Version 2.2 August 2000

© 1999-2000 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1

Introduction	4
<u>Defining Environment Variables</u>	7
<u>Performing a Comparison</u>	7

2

Comparing Two RTL Designs	9
<u>Compiling a VHDL Design</u>	9
<u>Compiling a Verilog Design</u>	10
<u>Comparing the Designs</u>	10
<u>More Information</u>	14

3

Comparing an RTL and a Gate-Level Design	15
<u>Preparing the Reference Library</u>	15
<u>Compiling the Implementation</u>	16
<u>Loading the Specification into the Compare Tab</u>	20
<u>Comparing the Designs</u>	21
<u>More Information</u>	24

4

Using Map Commands to Disable Scan Logic	26
<u>Compiling the Designs</u>	26
<u>Detecting a Mismatch</u>	27
<u>Creating a Command File</u>	27
<u>Comparing the Designs</u>	28
<u>More Information</u>	28

5

Affirma Equivalence Checker Tutorial

<u>Using Counterexamples</u>	30
<u>Compiling the Specification</u>	30
<u>Loading the Implementation into the Compare Tab</u>	32
<u>Comparing the Designs</u>	33
<u>Debugging the Designs</u>	34
<u>Locating the Appropriate Logic Cone</u>	35
<u>Examining Logic Cones</u>	35
<u>Defining Assumptions</u>	38
<u>More Information</u>	41
<u>Glossary</u>	42
<u>Index</u>	47

Introduction

Affirma™ equivalence checker determines whether two versions of a design — a *specification* and an *implementation* — are equivalent. Rather than performing what is often a time-consuming simulation, Affirma equivalence checker uses a set of mathematical proofs to determine whether the two designs are equivalent.

Affirma equivalence checker can perform the following comparisons:

- RTL-to-RTL comparison

After you have written a specification at the register-transfer level (RTL), Affirma equivalence checker can verify that changes to the design do not alter its intended functionality.

- RTL-to-gate comparison

Affirma equivalence checker can compare a gate-level netlist to the RTL design and determine whether the designs are equivalent.

- Gate-to-gate comparison

Affirma equivalence checker can compare two gate-level netlists and verify that updates, such as test insertion and clock tree insertion, do not alter the intended functionality of the design.

In this release, Affirma equivalence checker supports RTL models written in Verilog or VHDL. It supports gate-level netlists written in Verilog only.

Whenever Affirma equivalence checker determines that two designs are not equivalent, it generates *counterexamples* that show where and how they differ. You can then define *map commands* that show the conditions under which the designs should be equivalent.

When using Affirma equivalence checker, remember the following:

- Affirma equivalence checker performs functional verification. That is, it does not take timing into account when performing a verification.

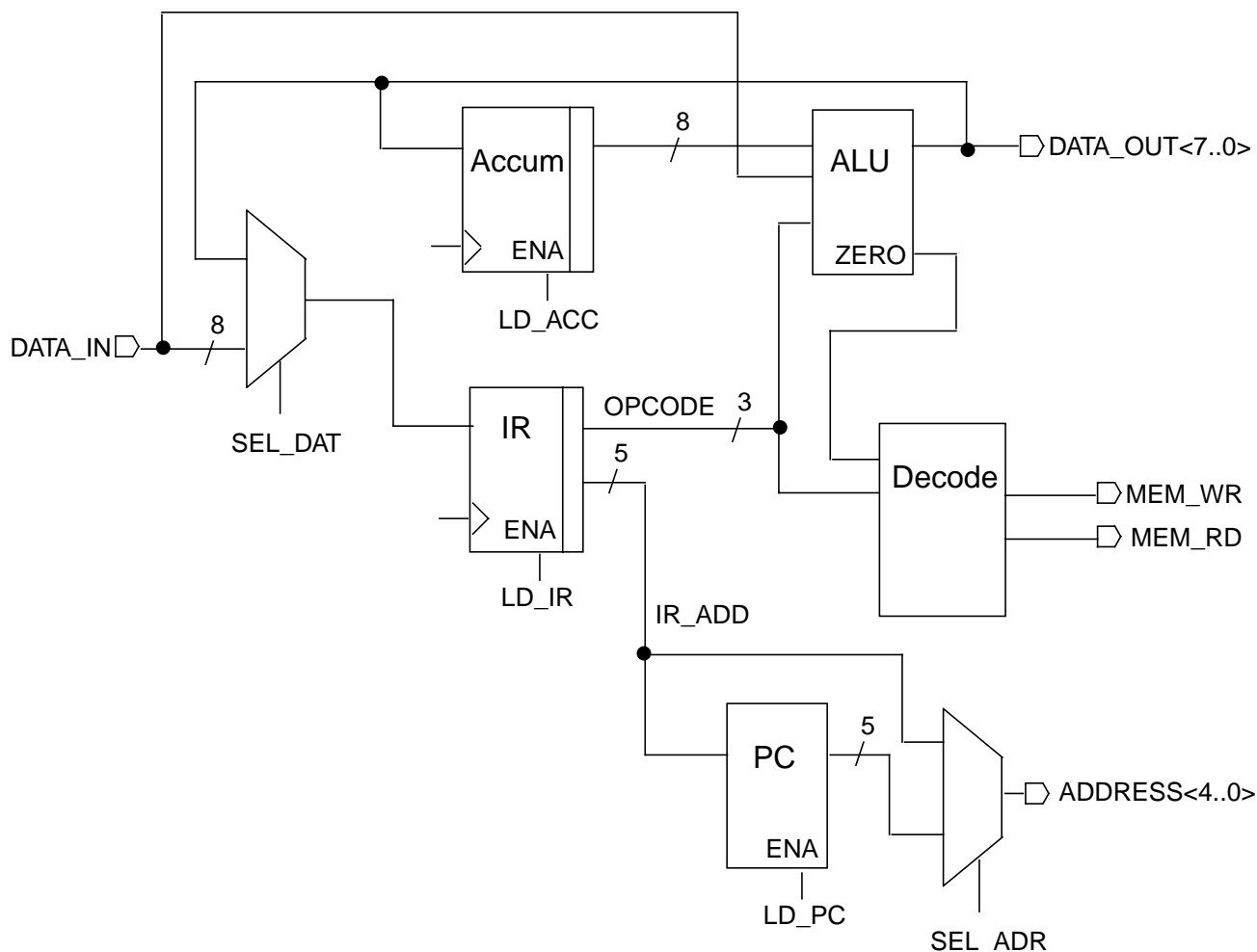
Affirma Equivalence Checker Tutorial

Introduction

- Affirma equivalence checker assumes that you have fully tested the specification. Equivalence checking can prove only that the specification and implementation are equivalent. It cannot prove that the specification is correct.

This document uses a simple CPU in its examples. (See [Figure 1-1](#) on page 5.) This design is made up of several modules — accumulator, arithmetic logic unit, instruction register, program counter, and decoder. Several versions of this system are provided for you to run the equivalence checking examples.

Figure 1-1 CPU Design



The source files for the designs, written at the RTL and the gate level, and the library that these designs reference are included in the Affirma equivalence checker product kit. They are stored in the `your_install_dir/tools/examples/ivf` directory, where `your_install_dir` represents the top of your Cadence installation hierarchy.

Affirma Equivalence Checker Tutorial

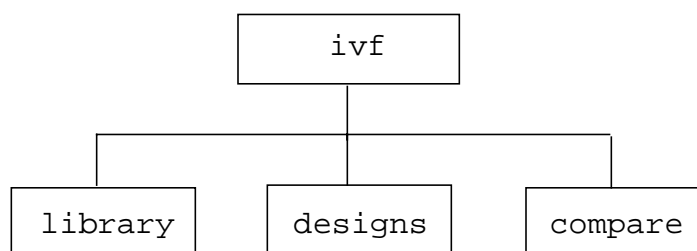
Introduction

If you want to run the examples that are described in this document, you must first change to a working directory, and then copy the example directories into your working directory, as follows:

```
cp -r your_install_dir/tools/examples/ivf .
```

The CPU design is organized in a directory structure that is similar to the one shown in [Figure 1-2](#) on page 6.

Figure 1-2 Equivalence Checking Directory Structure



Each subdirectory contains a particular set of files, as follows:

- The `library` subdirectory contains the source file or files for the reference library, if necessary.
- The `designs` subdirectory contains the source files for the various versions of the design.
- The `compare` subdirectory is the directory from which you compile your designs and perform the comparison. Affirma equivalence checker places the compiled design and the comparison results in subdirectories of this directory.

By running the examples in this document, you will see how you can use Affirma equivalence checker at many points in the design process. However, please note that this document gives you only a quick introduction to the tool. You can read more about Affirma equivalence checker in the [*Affirma Equivalence Checker User Guide*](#).

Defining Environment Variables

Before you use Affirma equivalence checker, you must define the following environment variables (where *your_install_dir* is the top-level directory in which Affirma equivalence checker is installed):

Variable	Description
CDS_LIC_FILE	Specifies the path to the Cadence license file on your system.
LD_LIBRARY_PATH (Solaris) or SHLIB_PATH (HPUX)	Specifies the path to the directory in which your Cadence shared libraries have been installed (usually <i>your_install_dir/tools/lib</i>).
PATH	Specifies the default search path for binary files. This variable must include paths to <i>your_install_dir/tools/bin</i> to access Affirma equivalence checker executable files.

Performing a Comparison

Regardless of the type of comparison that you are making — RTL-to-RTL, RTL-to-gate, or gate-to-gate — you perform the same steps:

1. Prepare the libraries, if necessary.

The `libprep` command analyzes and elaborates a library and creates a hierarchy of library cells, called a *reference library*, that you can use in your designs. The `libprep` script ensures that your library cells are ready for use in equivalence checking; that is, it ensures that all of the cells are synthesizable.

You need to prepare your libraries only once.

2. Compile the designs.

The `dp` command compiles and elaborates a Verilog or VHDL design into a complete design hierarchy, in which all of the instances in the design are linked together. The compiled design is stored in a *design directory*.

3. Compare the designs.

You can run Affirma equivalence checker in two ways:

Affirma Equivalence Checker Tutorial

Introduction

- ❑ With the `heck` command — You perform equivalence checking by issuing the `heck` command from the command prompt.
- ❑ With the graphical user interface (GUI) — You perform equivalence checking with the GUI by issuing the `heck -gui` command.

When it performs the comparison, Affirma equivalence checker writes information to a result directory, including a summary report file. The default result directory name is based on the names of the designs you are comparing: *design1-vs-design2*.

4. If the designs are not equivalent, locate and resolve the differences.

If the designs do not match, Affirma equivalence checker returns one or more *counterexamples*. You can use the counterexamples to locate the differences between the specification and the implementation.

You may need to modify your original source files, or you may need to supply Affirma equivalence checker with a set of assumptions, and then compare the designs again.

In this tutorial, you will run through these steps several times — to compare two RTL designs and to compare an RTL and a gate-level design. In addition, you will debug two cases in which Affirma equivalence checker detects differences between the designs.

More Information

You can read more about Affirma equivalence checker in the following documents:

- [Affirma™ Equivalence Checker User Guide](#)
- [Implementation Verification Methodology Guide](#)
- [Verilog Modeling Style Guide](#)
- [VHDL Modeling Style Guide](#)
- [Library Preparation Guide](#)

Comparing Two RTL Designs

The `designs` example directory contains two RTL models for the CPU. The *specification* is written in VHDL; the *implementation* is written in Verilog. You can run Affirma equivalence checker on these two designs to ensure that they are functionally equivalent. In this chapter, you perform the comparison with the `heck` command.

Compiling a VHDL Design

To compile the CPU design that is written in VHDL:

1. Change to your `compare` directory.
2. Run the `dp` command, as follows:

```
dp -Design vhdl_cpulib -Top cpu ../designs/alu_rtl.vhd \  
  ../designs/count5_rtl.vhd ../designs/cpu_rtl.vhd \  
  ../designs/decode_rtl.vhd ../designs/reg8_rtl.vhd
```

The `-Design` option specifies the name that you want to give to the design directory. The `-Top` option gives the name of the top-level entity in the design. In this example, `vhdl_cpulib` is the name of the design directory; `cpu` is the name of the top-level entity. The `dp` command compiles the design and creates an SMV file in the design directory.

As it compiles the design, the `dp` command displays the following messages:

```
Compiling with ncvhdl...  
Elaborating with ncelab...  
Generating SMV with CFE...  
SMV generation successful!
```

Note: A Makefile in the `compare` directory defines a target for this design. If you prefer, you can use the following command to prepare this design:

```
make compile_vhdl
```

Compiling a Verilog Design

To compile the Verilog design, you run the `dp` command from your `compare` directory, as follows:

```
dp -Design verilog_cpulib -Top cpu ../designs/alu.v \  
../designs/count5.v ../designs/cpu.v ../designs/decode.v \  
../designs/reg8.v
```

The `-Design` option specifies the name that you want to give to the design directory. The `-Top` option specifies the name of the top-level module in the design.

The `dp` command displays the following messages as it compiles the design:

```
Compiling with ncvlog...  
Elaborating with ncelab...  
Generating SMV with CFE...  
SMV generation successful
```

These messages indicate that `dp` has successfully compiled and elaborated the design, thus creating the design directory named `verilog_cpulib`. The messages also indicate that `dp` has created the SMV file for the design.

Note: A `Makefile` in the `compare` directory defines a target for this design. If you prefer, you can use the following command to prepare this design:

```
make compile_verilog
```

Comparing the Designs

To compare the two designs with the `heck` command, enter the following command from the `compare` directory:

```
heck -Spec vhdl_cpulib:cpu:a -Impl verilog_cpulib:cpu
```

The specification is a VHDL design. Therefore, the `-Spec` option specifies the name of the design directory, the top-level entity, and the architecture for the specification. In this example, the entity name is `cpu` and the architecture name is `a`. If you do not specify a top-level entity and architecture, Affirma equivalence checker uses the top-level module specified in the `dp` command.

The implementation is a Verilog design. Therefore, the `-Impl` option specifies the name of the design directory and the top-level module in the implementation. In this example, the name of the top-level module is `cpu`. If you do not specify a top-level module, Affirma equivalence checker uses the top-level module specified in the `dp` command.

Affirma Equivalence Checker Tutorial

Comparing Two RTL Designs

When it compares the two designs, `heck` writes messages to `stdout`. These messages provide information about the version of Affirma equivalence checker that you are running and the command-line options that you have used to invoke it. For example:

```
v2.20-n (c) Copyright 1998 - 2000 Cadence Design Systems, Inc.
Arguments: -CmdFile your_install_dir/tools/heck/include/defmaps/defcmd.txt -Spec
vhdl_cpulib:cpu:a -Impl verilog_cpulib:cpu
Result Directory: vhdl_cpulib_vs_verilog_cpulib
```

As `heck` analyzes the designs, it displays its progress. For example:

```
Compiling specification and implementation designs...
Generating hierarchy and verification data...
Reading maps from your_install_dir/tools/heck/include/defmaps/defcmd.txt...
Mapping latches by name...
Not all latches could be mapped by name.
Refer to vhdl_cpulib_vs_verilog_cpulib_1/map/map.detail for mapping information.
Performing functional latch mapping and verification...

RESULT: Implementation satisfies specification.

Generating verification reports...
WARNINGS:      0
ERRORS:        0
```

When it completes the comparison, Affirma equivalence checker creates a report that summarizes the results. This file is called `heck.report`, and it is placed in the result directory, a subdirectory of the directory in which you performed the comparison.

For example, `heck` generates the following report file for the RTL-to-RTL comparison of the CPU:

```
-----
heck.report file: This is a summary of the comparison results.
For details, refer to the heck.log file.
-----
```

	Specification	Implementation
	-----	-----
Design	vhdl_cpulib	verilog_cpulib
Module	cpu:a	cpu

```
=====
==
== RESULT: Implementation satisfies specification. ==
==
=====
```

Type of Comparison Points	Match	Mismatch
	-----	-----
Primary outputs.....	15	0
Black box boundaries.....	0	0
Glass box boundaries.....	0	0
Latches.....	28	0

CONDITIONS

Affirma Equivalence Checker Tutorial

Comparing Two RTL Designs

No conditions were specified for this comparison run.

WARNING COUNT

Comparison: 0 warnings

Compilation:

Spec: 0 warnings
Impl: 0 warnings

ERROR COUNT

0 errors

COMPILE INFORMATION

----- Specification:

```
% dp -Design vhdl_cpulib -Top cpu ../designs/alu_rtl.vhd
../designs/count5_rtl.vhd ../designs/cpu_rtl.vhd ../designs/decode_rtl.vhd
../designs/reg8_rtl.vhd
Date: day mm dd hh:mm:ss EDT 2000
refer to vhdl_cpulib/dp.log for more information
```

Implementation:

```
% dp -Design verilog_cpulib -Top cpu ../designs/alu.v ../designs/count5.v
../designs/cpu.v ../designs/decode.v ../designs/reg8.v
Date: day mm dd hh:mm:ss EDT 2000
refer to verilog_cpulib/dp.log for more information
```

COMPARISON INFORMATION

----- Comparison run command:

```
% heck -CmdFile your_install_dir/tools/heck/include/defmaps/defcmd.txt -Spec
vhdl_cpulib:cpu:a -Impl verilog_cpulib:cpu
Date: day mm dd mm:ss:mm 2000
Memory used: 5 MBytes
Run Time: 0.97s
```

Software Versions

Affirma_EC: v2.20-n
cfe: v3.11.(n.n)
NC: v3.11.(n)
dp: v2.0

Affirma equivalence checker also creates a file called map/map.detail in the result directory. This file shows which latches were compared to one another, through mapping.

Affirma Equivalence Checker Tutorial

Comparing Two RTL Designs

Affirma equivalence checker can map latches by name or by the function that they perform. If it cannot map all of the latches, Affirma equivalence checker writes the names of the unmapped latches to the `map/map.detail` file.

For example, Affirma equivalence checker is not able to map all of the latches in the RTL-to-RTL comparison by name. It mapped the remaining latches by function. Therefore, the contents of the `map/map.detail` file are as follows:

```
# map.detail file: This file contains the mapping information
# for the associated comparison run.
# -----

#                               Specification                               Implementation
#                               -----                               -----
# Design                        vhdl_cpulib                            verilog_cpulib
# Module                        cpu:a                                cpu

# Latches mapped by default name
map spec:alul.aluout[0] impl:alul.aluout[0]
map spec:alul.aluout[1] impl:alul.aluout[1]
map spec:alul.aluout[2] impl:alul.aluout[2]
map spec:alul.aluout[3] impl:alul.aluout[3]
map spec:alul.aluout[4] impl:alul.aluout[4]
map spec:alul.aluout[5] impl:alul.aluout[5]
map spec:alul.aluout[6] impl:alul.aluout[6]
map spec:alul.aluout[7] impl:alul.aluout[7]
map spec:decode1.state[0] impl:decode1.state[0]
map spec:decode1.state[1] impl:decode1.state[1]
map spec:decode1.state[2] impl:decode1.state[2]
map spec:iregl.dataOut[1] impl:iregl.dataOut[1]
map spec:iregl.dataOut[2] impl:iregl.dataOut[2]
map spec:iregl.dataOut[3] impl:iregl.dataOut[3]
map spec:iregl.dataOut[4] impl:iregl.dataOut[4]
map spec:iregl.dataOut[5] impl:iregl.dataOut[5]
map spec:iregl.dataOut[6] impl:iregl.dataOut[6]
map spec:iregl.dataOut[7] impl:iregl.dataOut[7]
map spec:iregl.dataOut[8] impl:iregl.dataOut[8]
map spec:accum1.dataOut[1] impl:accum1.dataOut[1]
map spec:accum1.dataOut[2] impl:accum1.dataOut[2]
map spec:accum1.dataOut[3] impl:accum1.dataOut[3]
map spec:accum1.dataOut[4] impl:accum1.dataOut[4]
map spec:accum1.dataOut[5] impl:accum1.dataOut[5]
map spec:accum1.dataOut[6] impl:accum1.dataOut[6]
map spec:accum1.dataOut[7] impl:accum1.dataOut[7]
map spec:accum1.dataOut[8] impl:accum1.dataOut[8]
map spec:alul.zero impl:alul.zero

# Cannot map spec latch pcount1.qout[1] by name;
# Cannot map spec latch pcount1.qout[2] by name;
# Cannot map spec latch pcount1.qout[3] by name;
# Cannot map spec latch pcount1.qout[4] by name;
# Cannot map spec latch pcount1.qout[5] by name;
# Cannot map impl latch pcount1.q[1] by name;
# Cannot map impl latch pcount1.q[2] by name;
# Cannot map impl latch pcount1.q[3] by name;
# Cannot map impl latch pcount1.q[4] by name;
```

Affirma Equivalence Checker Tutorial

Comparing Two RTL Designs

```
# Cannot map impl latch pcount1.q[5] by name;
# The following map commands were generated by Heck during latch mapping
# by function:
map spec:pcount1.qout[1] impl:pcount1.q[1]
map spec:pcount1.qout[2] impl:pcount1.q[2]
map spec:pcount1.qout[3] impl:pcount1.q[3]
map spec:pcount1.qout[5] impl:pcount1.q[5]
map spec:pcount1.qout[4] impl:pcount1.q[4]

#-----
# Affirma_EC:          v2.20-n
# cfe:                 v3.11.(n.n)
# NC:                 v3.11.(n)
# dp:                 v2.0
```

Note: A Makefile in the `compare` directory defines a target for this comparison. If you prefer, you can use the following command to compare these designs:

```
make compare_vhdl_verilog
```

More Information

This chapter has shown you the basic `heck` command-line options. You may also want to try the following `heck` options:

<code>-ResultDir</code>	Specifies the name of the directory to which <code>heck</code> writes output files. The default result directory name is based on the names of the designs in the comparison: <i>design1-vs-design2</i> .
<code>-Keep</code>	Prevents the equivalence checker from overwriting a result directory with the same name.
<code>-Version</code>	Returns the version of Affirma equivalence checker that is installed on your system and exits without further processing.
<code>-Help</code>	Displays a short description of the <code>heck</code> command-line options.

Comparing an RTL and a Gate-Level Design

Suppose that you have synthesized the CPU model to produce a gate-level netlist. You can use Affirma equivalence checker to compare the netlist to the RTL design and to determine whether the designs are functionally equivalent.

The `designs` example directory contains a netlist for the CPU, and the `library` directory contains a reference library, which the gate-level netlist uses. You must prepare this library before you compile the design. In this example, you will use the GUI to compare the designs.

Preparing the Reference Library

To prepare the reference library, enter the following `libprep` command from the `library` directory:

```
libprep -v demo_lib.v -work demoLib -import -include cells_list
```

The `-v` option specifies the name of the Verilog source file that contains the library cell definitions. The `-work` option specifies the name that you want to give to the [reference library](#).

The `-import` option specifies that you want `libprep` to translate nonsynthesizable cells into a synthesizable form, and the `-include` option specifies the name of the file that contains the cells that you want `libprep` to import.

As it compiles the library, `libprep` displays the following messages:

```
LibPrep: v03.11-p001: (c) Copyright 1996 - 1999 Cadence Design Systems, Inc.  
Build on Wed May 10 13:17:54 IST 2000 bulbul.Cadence.COM vcalds  
Started at: Mon Jul 10 10:15:42 EDT 2000  
ncvlog -work demoLib -nocopyr -pragma -append_log -linedebug -view heck  
/hm/claudia/examples/library/.timescale.v <files>
```

```
Verilog compilation (ncvlog) successful  
LibImport: v03.11-p001: (c) Copyright 1996 - 1999 Cadence Design Systems, Inc.  
Build on Wed May 10 13:17:49 IST 2000 synserv1 vcalds  
Started at: Mon Jul 10 10:15:48 EDT 2000
```

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

```
Generating Verilog Code for module DFFR
Generating Verilog Code for module SDFFR
LibImport completed: Mon Jul 10 10:16:26 EDT 2000
ncvlog -work demoLib -nocopyr -append_log -pragma -linedebug -view heck
lib.output.demoLib
Verilog compilation (ncvlog) for imported library successful

Library compilation for demoLib completed successfully
Libprep completed: Mon Jul 10 10:16:26 EDT 2000
```

Note: A Makefile in the `library` directory defines a target for this reference library. If you prefer, you can use the following command to prepare the library.

```
make demolib
```

Compiling the Implementation

You can compile your designs with the GUI, if you have not already done so with the `dp` command.

In this example, you use the `verilog_cpulib` design that you compiled in [Chapter 2, “Comparing Two RTL Designs”](#). This design is the specification. You must compile the gate-level netlist, which is the *implementation* in this comparison, as follows:

1. Invoke the GUI by entering the following command from the `compare` directory:

```
heck -gui
```

The Affirma equivalence checker main window appears. (See [Figure 3-1](#) on page 17).

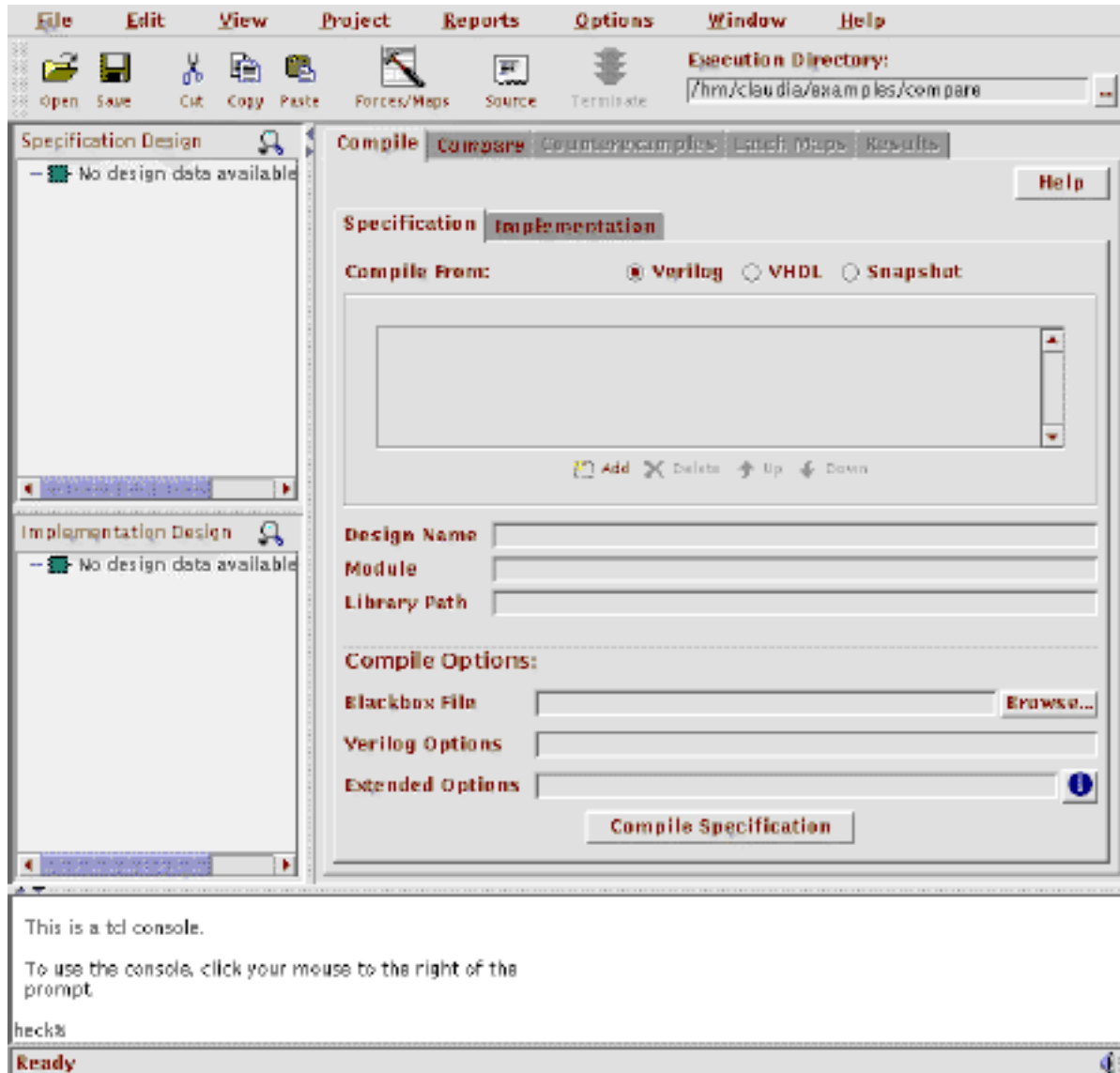
The Main window is divided into a number of tabs. These tabs let you compile your designs, compare your designs, and examine counterexamples, latch mapping tables, and the results of a comparison. Affirma equivalence checker moves between tabs as you work with your designs, and you can move between the tabs at any time by clicking on them.

When it starts up, the GUI displays the Compile tab so that you can prepare your designs for equivalence checking. The Compile tab has two subtabs, which let you switch between the specification and the implementation.

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

Figure 3-1 Main Window with No Project Information

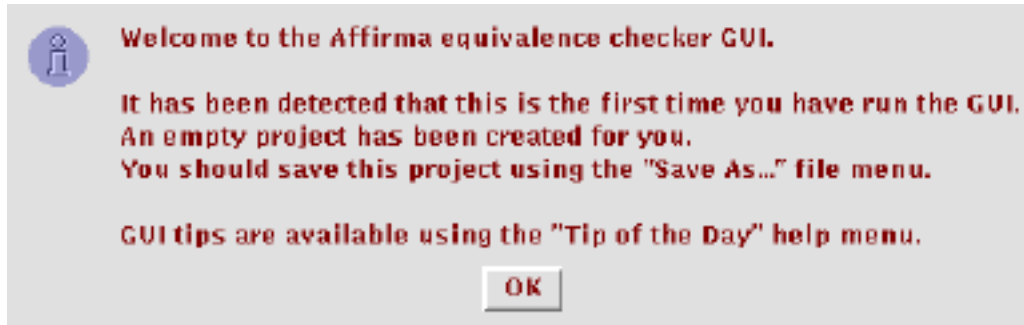


If this is the first time that you have run Affirma equivalence checker, the GUI also displays a message box that indicates that Affirma equivalence checker has created an empty project for you. (See [Figure 3-2](#) on page 18.) You create a *project* as you work with Affirma equivalence checker, and you can save the project when you exit from the GUI.

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

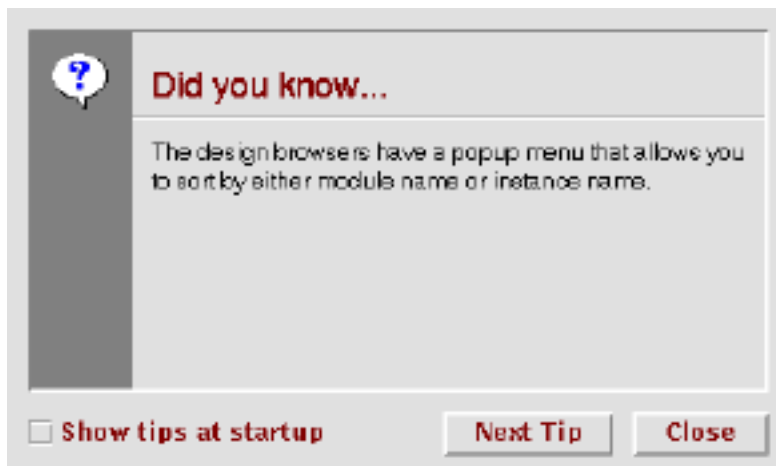
Figure 3-2 First-Time User Message Box



Click *OK* to dismiss the message box.

The GUI also displays a Tip of the Day message box. (See [Figure 3-3](#) on page 18.) This message box pops up whenever you start up the equivalence checker.

Figure 3-3 Tip of the Day Message Box



You can disable this message box by deselecting the *Show tips at startup* button, or browse through other tips by clicking *Next Tip*.

Close this message box by clicking *Close*.

2. Open the Implementation subtab and click *Add* below the list area. Affirma equivalence checker creates an empty text field. You can enter the name of the Verilog files that make up your design directly into the text field. For this example, enter `../designs/gates.v` and press *Return*.

If you do not know the name or location of a file, you can click *Browse* to navigate the directory structure and select the file that you want to compile.

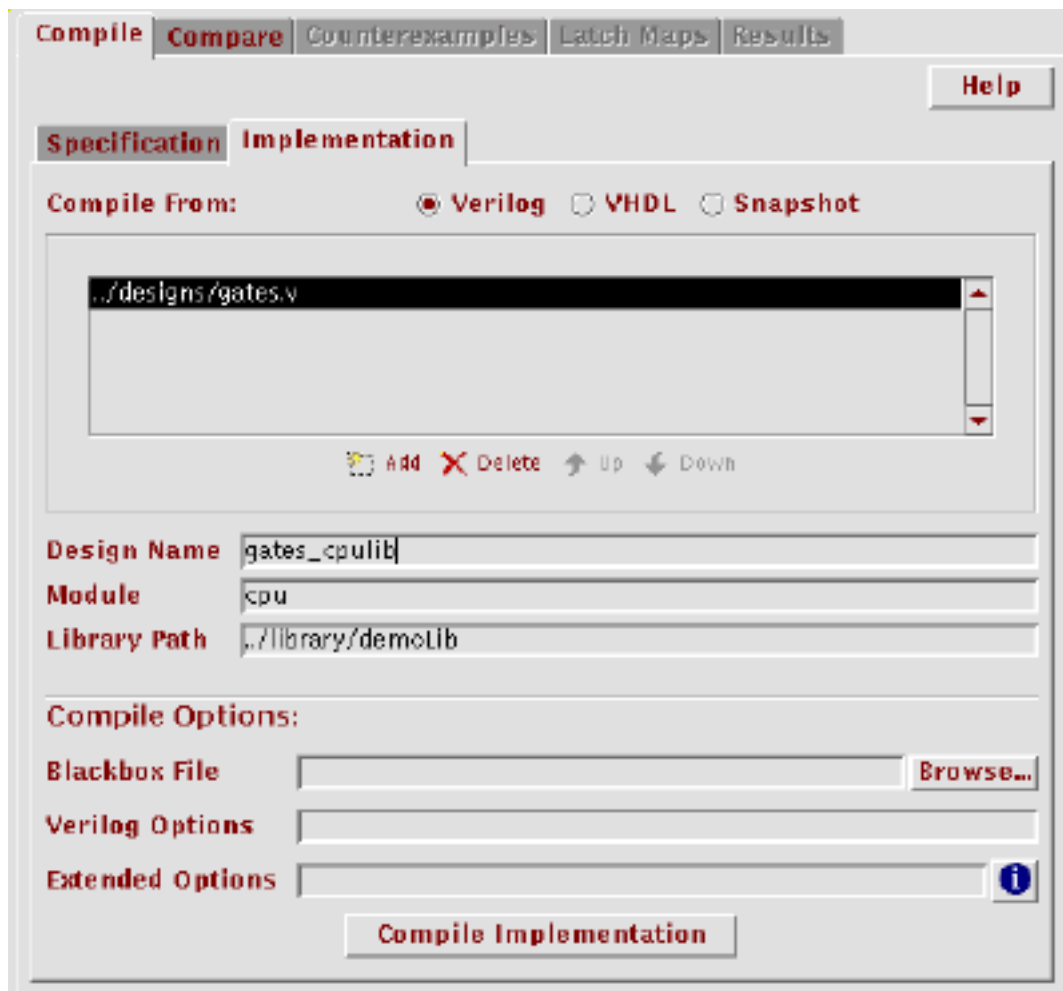
Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

3. In the *Design Name* field, enter the name that you want to give to the design directory and the name of the top-level module. In this example, enter `gates_cpulib` in the *Design Name* field, and enter `cpu` in the *Module* field.
4. Enter the path to your reference library in the *Library Path* field. For this example, enter `../library/demoLib`. This is the path to the reference library that you compiled in [Preparing the Reference Library](#) on page 15.

[Figure 3-4](#) on page 19 shows the Compile tab for the implementation.

Figure 3-4 Compiling the Implementation



5. Click *Compile Implementation*.

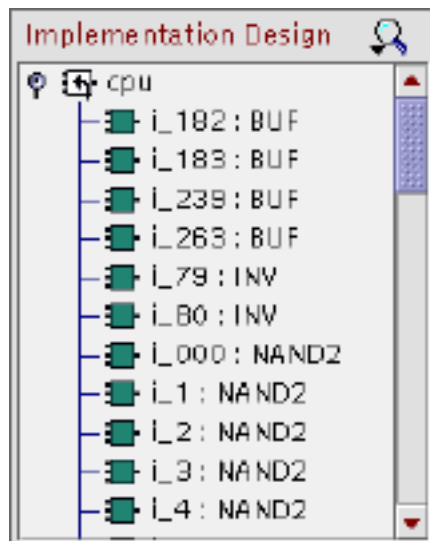
Affirma equivalence checker displays messages in the Command area at the bottom of the Main window. These messages show you that Affirma equivalence checker has

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

successfully compiled the design. Affirma equivalence checker then displays the implementation design hierarchy, as shown in [Figure 3-5](#) on page 20.

Figure 3-5 Implementation Design Hierarchy



Loading the Specification into the Compare Tab

After it compiles the implementation, Affirma equivalence checker opens the Compare tab. (See [Figure 3-7](#) on page 21.) In this tab, Affirma equivalence checker displays information about the design that you compiled — `gates_cpulib`. If you have already compiled the Verilog RTL model (`verilog_cpulib`), as described in [Compiling a Verilog Design](#) on page 10, you can load that design directly into the Compare tab, as follows:

- Click on the *Browse* button to the right of the *Specification Design* field and select `verilog_cpulib` from the list of designs in the Select Design form, as shown in [Figure 3-6](#) on page 20. Click *OK* to load the design.

Figure 3-6 Loading the Specification into the Compare Tab



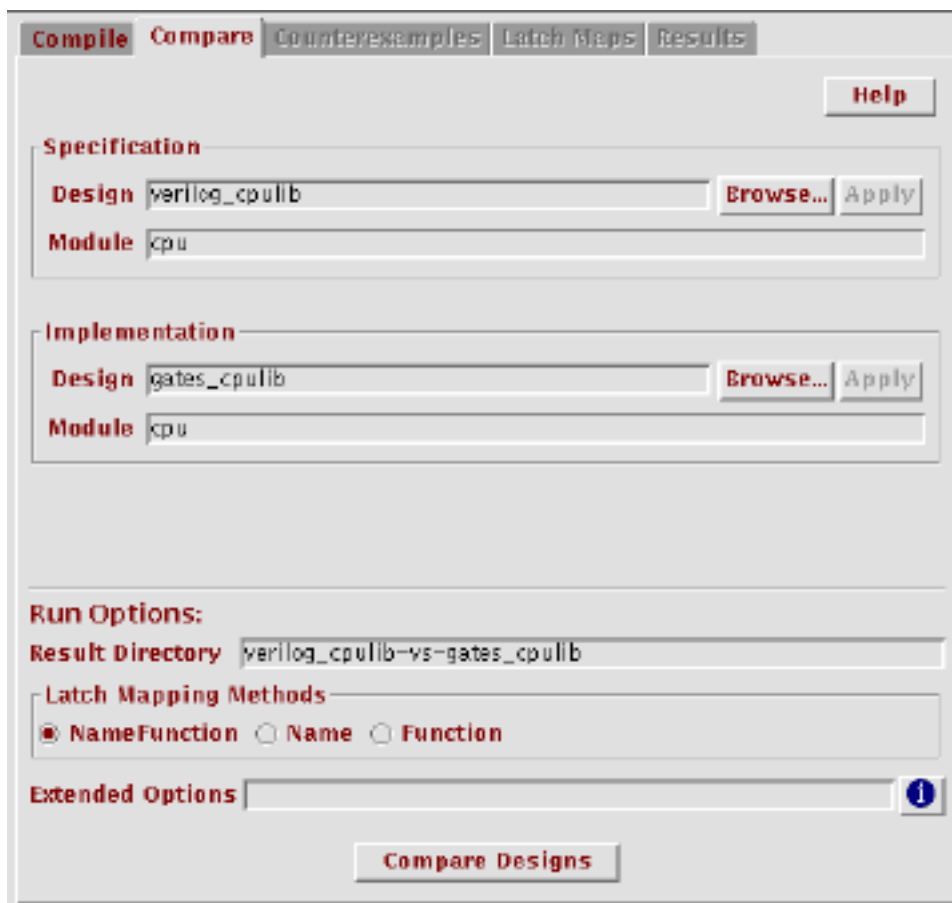
Comparing the Designs

Below the design information, the Compare tab contains the options that Affirma equivalence checker uses when it performs the comparison. Initially, the Compare tab displays the default settings for the name of the result directory and the latch mapping method (name and function mapping). The *Extended Options* field lets you enter any command-line options that the *heck* command accepts. For a list of these options, click the button to the right of the text field.

To compare the designs:

1. Accept the default settings (shown in [Figure 3-7](#) on page 21) by clicking *Compare Designs*.

Figure 3-7 Comparison Settings



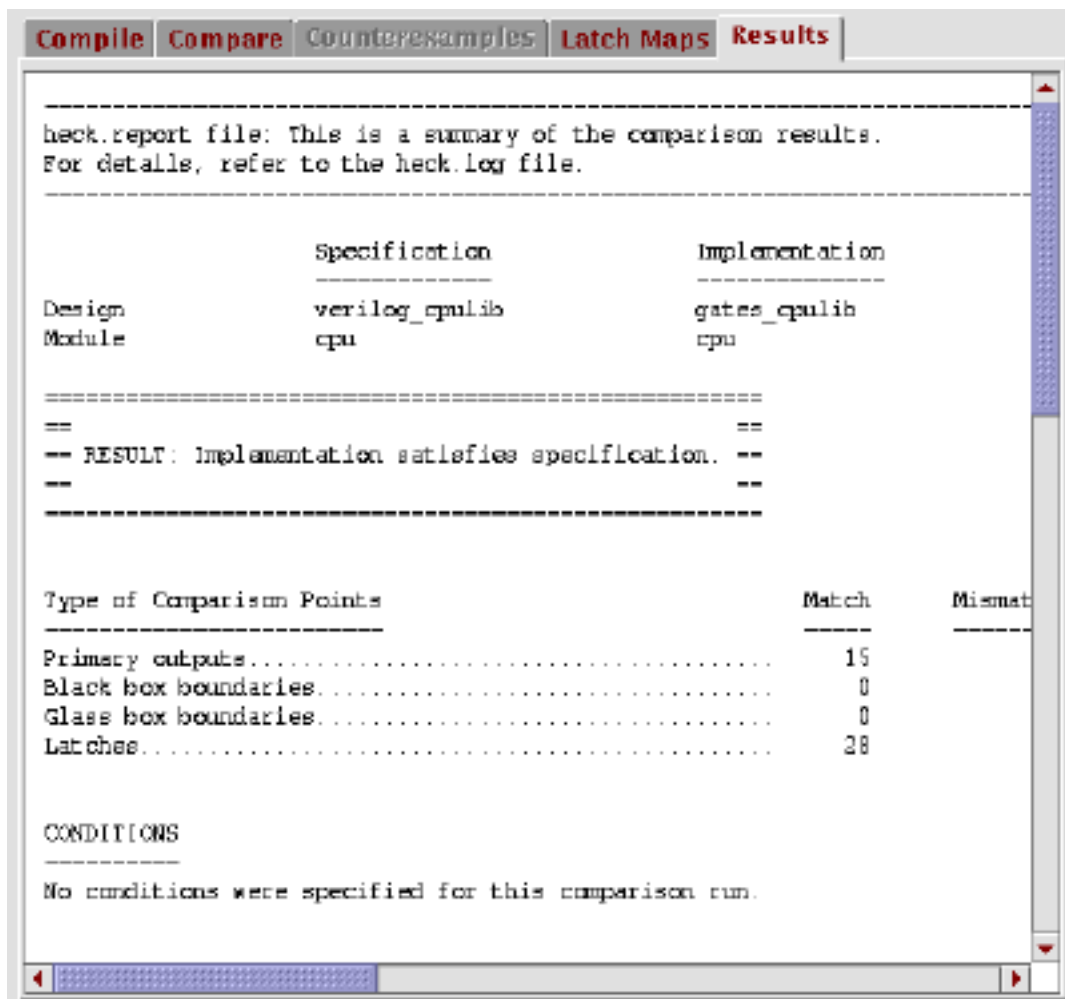
When it compares the two designs, Affirma equivalence checker writes messages to the Command area at the bottom of the window. These messages provide information about the tool's progress through the comparison.

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

When it has finished the comparison, Affirma equivalence checker displays the summary report file, `heck.report`, in the Results tab. (See [Figure 3-8](#) on page 22.) The Results tab shows the names of the designs that were compared and a statement that the implementation satisfies or does not satisfy the specification. Other information in the report includes a summary of the number and types of comparison points and any conditions, such as map commands, that were in effect during the comparison.

Figure 3-8 Results Tab



If you want to see how Affirma equivalence checker mapped latches in this comparison, you can open the Latch Maps tab. (See [Figure 3-9](#) on page 23.)

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

Figure 3-9 Latch Maps Tab

Specification Latches	Implementation Latches
pcount1.q[1]	pcount1_A.q_reg_1.Q
pcount1.q[1]	pcount1_A.q_reg_1.Q
pcount1.q[2]	pcount1_A.q_reg_2.Q
pcount1.q[2]	pcount1_A.q_reg_2.Q
pcount1.q[3]	pcount1_A.q_reg_3.Q
pcount1.q[3]	pcount1_A.q_reg_3.Q
pcount1.q[5]	pcount1_A.q_reg_5.Q
pcount1.q[5]	pcount1_A.q_reg_5.Q
pcount1.q[4]	pcount1_A.q_reg_4.Q
pcount1.q[4]	pcount1_A.q_reg_4.Q
alu1.zero	alu1.zero_reg.Q
alu1.aluout[0]	alu1.aluout_reg_0.Q
alu1.aluout[1]	alu1.aluout_reg_1.Q
alu1.aluout[2]	alu1.aluout_reg_2.Q
alu1.aluout[3]	alu1.aluout_reg_3.Q
alu1.aluout[4]	alu1.aluout_reg_4.Q
alu1.aluout[5]	alu1.aluout_reg_5.Q
alu1.aluout[6]	alu1.aluout_reg_6.Q
alu1.aluout[7]	alu1.aluout_reg_7.Q
decode1.state[0]	decode1.state_reg_0.Q
decode1.state[1]	decode1.state_reg_1.Q
decode1.state[2]	decode1.state_reg_2.Q

2. Exit from the GUI by choosing *Exit* from the *File* menu.

Affirma equivalence checker displays a message box that asks if you want to save your project settings to a file. (See [Figure 3-10](#) on page 24.) When you save your project settings, Affirma equivalence checker saves information about the designs that you have worked on. It also saves the options that you used when comparing the designs. You can load this project file into the GUI at any time, and continue your equivalence checking without recompiling the designs or reselecting your comparison options.

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

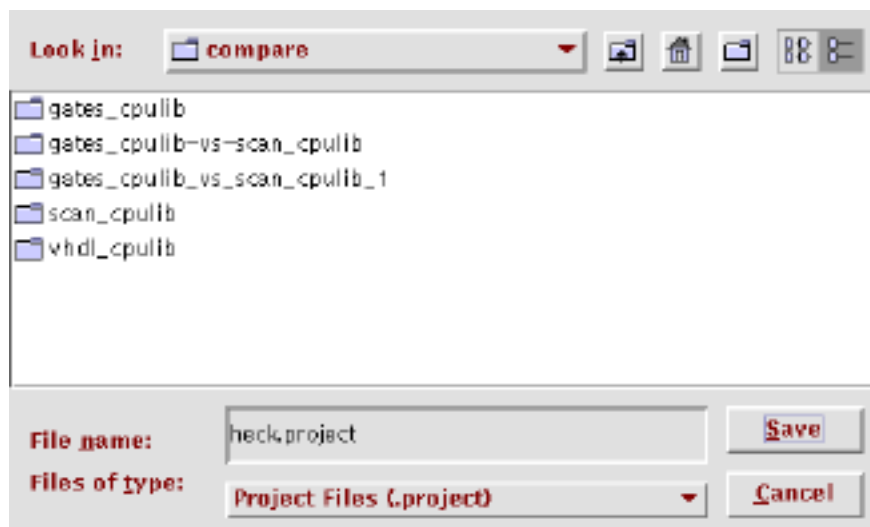
Figure 3-10 Save Project File Message Box



Click Yes.

3. Affirma equivalence checker opens a File Selection form. (See [Figure 3-11](#) on page 24.) By default, the project file is named `heck.project`; it is stored in the `compare` directory. Click *OK* to accept the default project filename.

Figure 3-11 File Selection Dialog Box



More Information

This chapter has shown you the basic features of the GUI. You may also want to try the following other features:

Options – Preferences

Lets you change the default startup options and the appearance of the GUI.

Affirma Equivalence Checker Tutorial

Comparing an RTL and a Gate-Level Design

Reports – Log Files

Lets you display additional output files, including the compilation log files and the `heck.out` file.

Help – Affirma equivalence checker User Guide

Opens the online user guide.

Using Map Commands to Disable Scan Logic

When two designs have different numbers of input ports, Affirma equivalence checker cannot map all of the ports in the designs. This can cause the designs to mismatch. However, additional ports may be connected to a scan chain. If so, you can force the extra ports to a value and disable the additional logic. You may be able to prove that the designs are equivalent under that condition.

In this example, you compare two gate-level netlists. The *implementation* has more ports than the *specification*. These extra ports are the result of scan insertion. To prevent a mismatch, you create a command file that contains a *map command* that forces the scan enable bit to 0 and disables the scan chain.

Compiling the Designs

If you ran the examples in [Chapter 3, “Comparing an RTL and a Gate-Level Design”](#), you created a reference library (`demoLib`) and a design directory for the CPU netlist (`gates_cpulib`). You use this design directory as the specification in this chapter, and you use the reference library to resolve cell references in the implementation.

To compile the *implementation* design, enter the following command from the `compare` directory:

```
dp -Design scan_cpulib -Top cpu -RefLibPath ../library/demoLib \  
../designs/gates_scan.v
```

This command creates a design directory named `scan_cpulib`, and it uses the `demoLib` library to resolve references to the gates that are used in the design.

Note: A `Makefile` in the `compare` directory defines a target for this design. If you prefer, you can use the following command to prepare this design:

```
make compile_scan
```

Detecting a Mismatch

When you do not use map commands, as shown in the following command, the comparison generates a mismatch, as follows:

```
heck -spec gates_cpulib:cpu -impl scan_cpulib:cpu
```

As Affirma equivalence checker maps inputs and outputs, it determines that the implementation has two extra inputs and one extra output. It displays the following warning messages:

```
WARNING: Port SE is an input of impl only.  
WARNING: Port SI is an input of impl only.  
WARNING: Port SO is an output of impl only.
```

As a result, Affirma equivalence checker is unable to verify the designs, as shown in the following messages:

```
RESULT: Implementation DOES NOT satisfy specification.
```

```
Generating verification reports...  
WARNINGS:      3  
ERRORS:        0
```

The extra inputs and outputs represent the scan enable (SE), scan input (SI), and scan output (SO) signals that make up the scan chain. Before you can prove that these designs are equivalent, you need to create a command file to disable the scan logic.

Creating a Command File

Map commands let you explicitly map nets, modules, and instances that Affirma equivalence checker does not otherwise map to each other. You can also use map commands to define general name-mapping rules, which specify how the names of objects in the design are transformed through a process such as synthesis. Other map commands force signals to 0 or 1.

When you use the `heck` command, you place map commands in a text file, and then pass that file to `heck` with the `-CmdFile` option. When you define map commands with the GUI, Affirma equivalence checker makes them part of the project, so that it can include the commands in subsequent comparisons.

To create a command file that disables the scan chain in the CPU design:

1. In the `compare` directory, create an ASCII file with any text editor (such as `vi`), and insert the following map command:

```
force impl:SE 0
```

The `force` command forces the SE signal to 0, which disables the scan chain.

Affirma Equivalence Checker Tutorial

Using Map Commands to Disable Scan Logic

2. Save this file and (for the purposes of this example) name it `mapin.txt`.

Note: The `compare` directory contains a text file called `mapin.txt`, which lists all of the map commands that you need to run the examples in this tutorial and all of the examples in the *Affirma Equivalence Checker User Guide*. For this example, you can uncomment the `force` command and leave the other commands commented out.

Comparing the Designs

To compare the two designs, invoke `heck` from the `compare` directory. You use the `-CmdFile` option to include the command file in the comparison, as follows:

```
heck -Spec gates_cpulib:cpu -Impl scan_cpulib:cpu -CmdFile mapin.txt
```

When Affirma equivalence checker tries to map the inputs and outputs, it displays the following messages:

```
Reading maps from mapin.txt...
WARNING: Port SE is an input of impl only.
WARNING: Port SI is an input of impl only.
WARNING: Port SO is an output of impl only.
```

However, when Affirma equivalence checker performs the comparison, it applies the `force` command and finds the designs to be equivalent:

```
RESULT: Implementation satisfies specification.
```

```
Generating verification reports...
WARNINGS: 3.
ERRORS: 0.
```

Note: A `Makefile` in the `compare` directory defines a target for performing this comparison. If you prefer, you can use the following command to compare these designs:

```
make compare_gates_scan
```

More Information

This chapter has shown you how to force the value of a net by using a command file. You may want to try the following other ways to control how `heck` maps objects in the two designs:

- | | |
|-------------------------------|---|
| <code>-LatchMapMethod</code> | Lets you specify how Affirma equivalence checker maps latches: <code>name</code> , <code>function</code> , or <code>nameFunction</code> . |
| <code>-AutoLatchInvert</code> | Specifies that you want the equivalence checker to invert latches in both designs so that they contain identical state encoding. |

Affirma Equivalence Checker Tutorial

Using Map Commands to Disable Scan Logic

-Similar

Specifies that you want Affirma equivalence checker to compare two comparison points if they have the same name. This option is recommended for gate-to-gate comparisons.

Using Counterexamples

When two designs are not equivalent, Affirma equivalence checker generates counterexamples, which show the inputs, the outputs, and the logic that produce a mismatch.

In this chapter, you run Affirma equivalence checker with the GUI to examine a *counterexample* and to locate the difference between the designs. You then map the inverted latches and run Affirma equivalence checker again.

Compiling the Specification

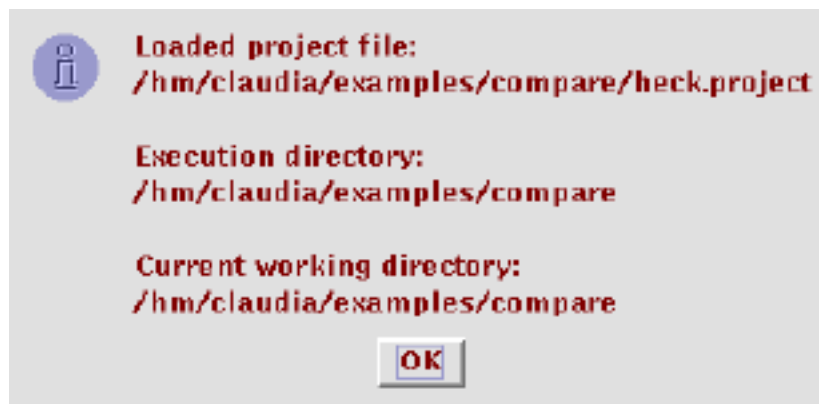
To compile the specification for this example:

1. Invoke the GUI, by entering the following command from the `compare` directory:

```
heck -gui &
```

If you have saved the project file as described in [Chapter 3, “Comparing an RTL and a Gate-Level Design”](#), the user interface displays a message box similar to the one shown in [Figure 5-1](#) on page 30.

Figure 5-1 Project File Message Box



Affirma Equivalence Checker Tutorial

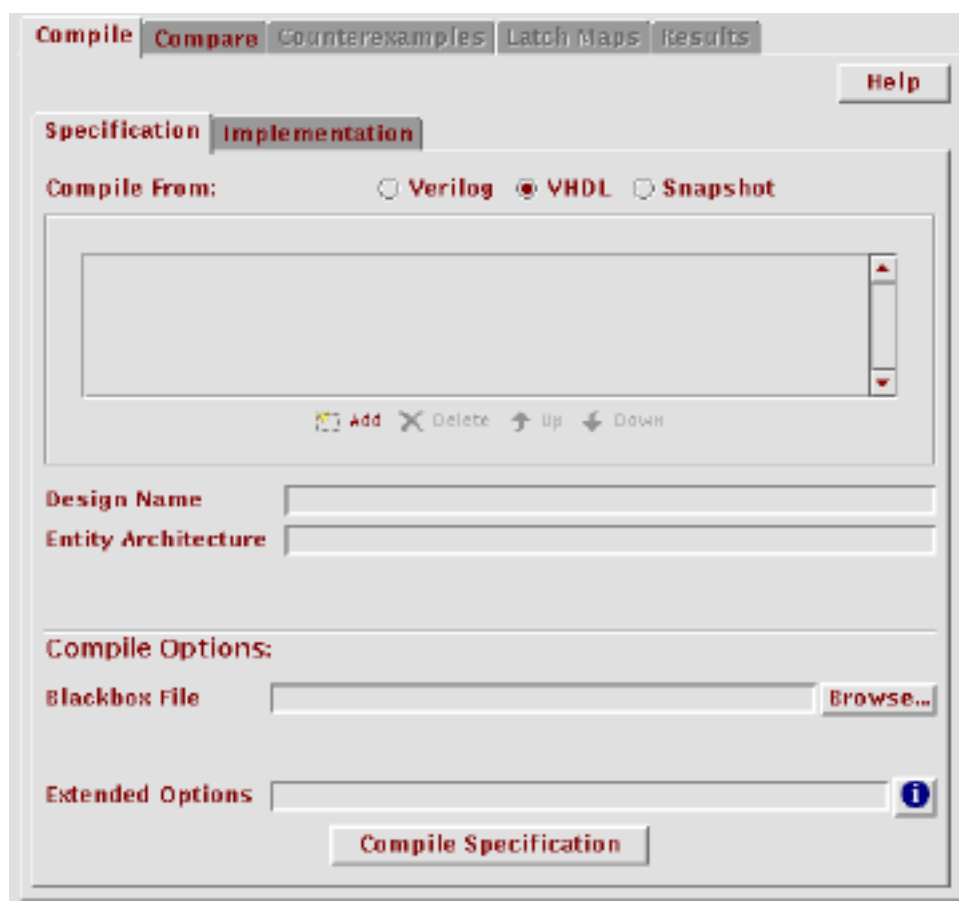
Using Counterexamples

Click *OK* to dismiss the message box, and then choose *File – New Project – Empty* from the menu bar to clear the project information from the user interface.

By default, Affirma equivalence checker loads the project file that you have most recently saved when you start up the user interface. To change this default behavior, choose *Options – Preferences* from the menu bar and enable the *Create a new project* button in the Project tab.

2. In the Compile tab's Specification subtab, click *VHDL*. The GUI displays the tab fields that apply to VHDL designs. (See [Figure 5-2](#) on page 31.)

Figure 5-2 Compile Tab for VHDL Designs

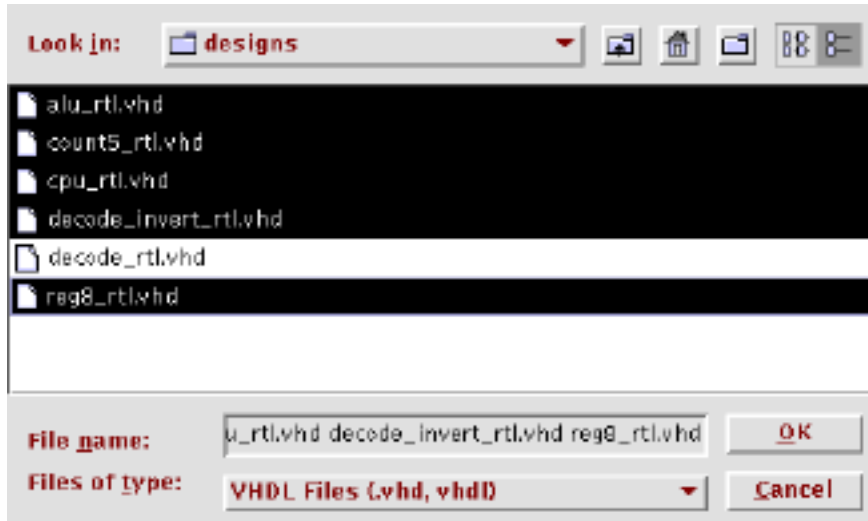


3. Click *Add*, and then click *Browse* to choose the VHDL files that you want to compile from the File Selection form. Use the *Look in* menu to find the `designs` directory, then hold down the *Control* key and select `alu_rtl.vhd`, `count5_rtl.vhd`, `cpu_rtl.vhd`, `decode_invert_rtl.vhd`, and `reg8_rtl.vhd`, as shown in [Figure 5-3](#) on page 32.

Affirma Equivalence Checker Tutorial

Using Counterexamples

Figure 5-3 Selecting VHDL Files



Click *OK* to confirm your selection and close the form.

4. Enter `invert_cpulib` in the *Design Name* field, and enter `cpu` in the *Entity Architecture* field of the *Compile* tab.
5. Click *Compile Specification* to compile the design.

Note: A `Makefile` in the `compare` directory defines a target for this design. If you prefer, you can use the following command to prepare this design:

```
make compile_invert
```

Loading the Implementation into the Compare Tab

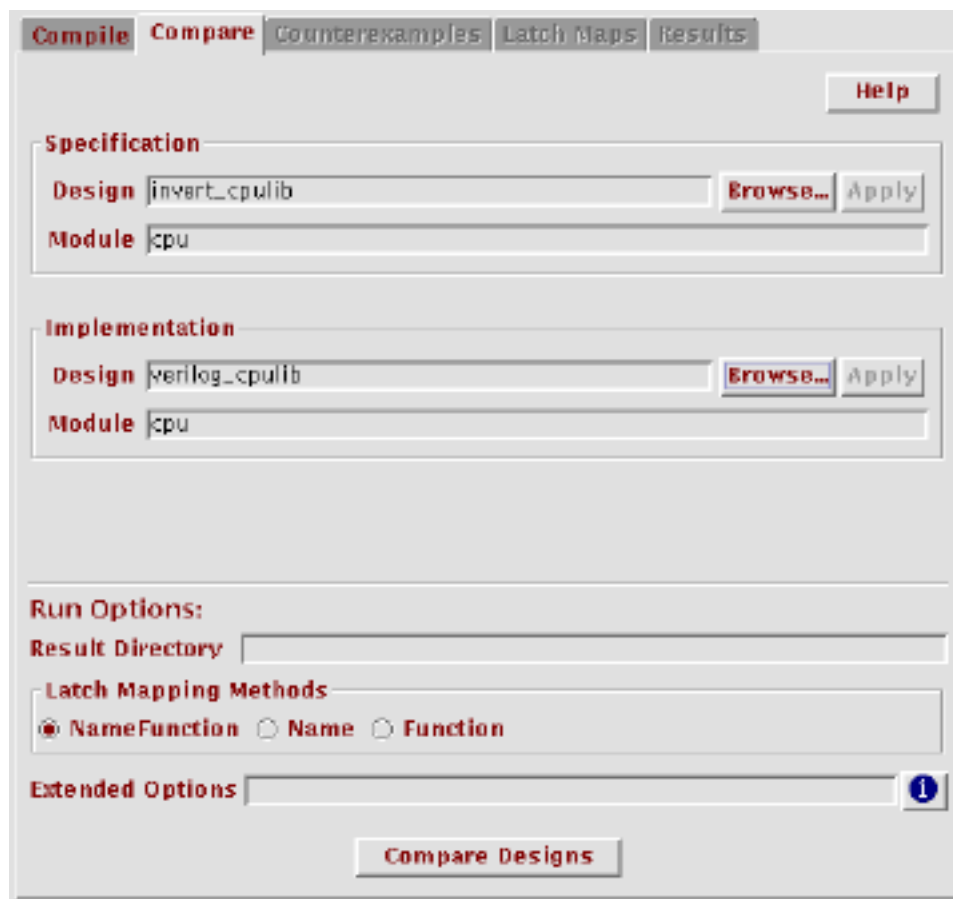
If you have already compiled the `verilog_cpulib` design as described in [Compiling a Verilog Design](#) on page 10, you can load it directly into the *Compare* tab, as follows:

1. Open the *Compare* tab.
2. In the *Implementation Design* field, click *Browse* and select the `verilog_cpulib` design.

[Figure 5-4](#) on page 33 shows the *Compare* tab for this comparison.

Affirma Equivalence Checker Tutorial Using Counterexamples

Figure 5-4 Loading a Design into the Compare Tab



Comparing the Designs

To compare the designs:

1. In the Compare tab, click *Compare Designs*.

When Affirma equivalence checker compares these two designs, it finds that they are not equivalent, and it displays the following message in the Command area of the window:

```
RESULT: Implementation DOES NOT satisfy specification.
```

```
Generating verification reports...
```

```
WARNINGS:      0
```

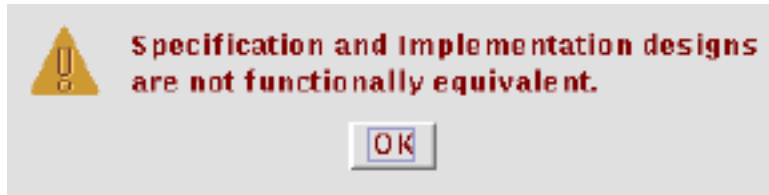
```
ERRORS:        0
```

Affirma equivalence checker also displays a warning message box. (See [Figure 5-5](#) on page 34.) Click *OK* to dismiss the message box.

Affirma Equivalence Checker Tutorial

Using Counterexamples

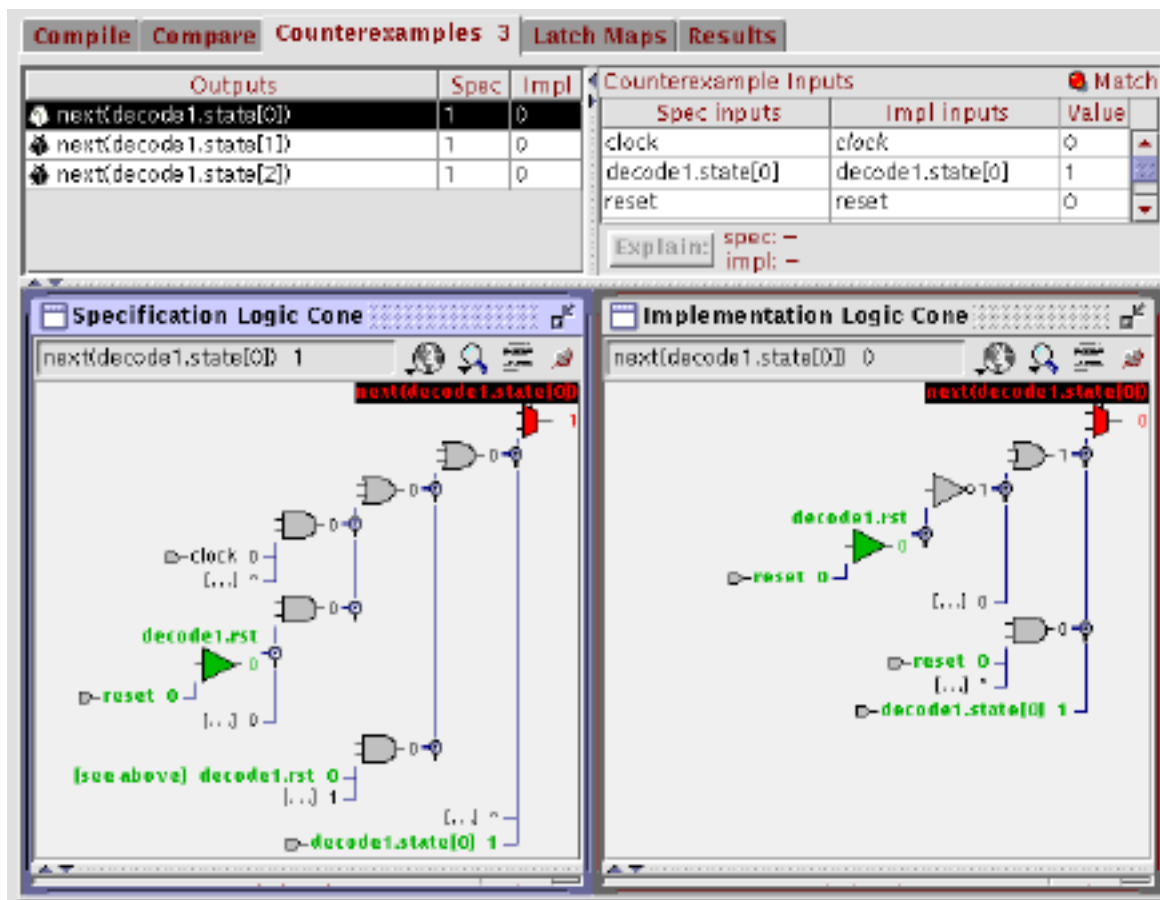
Figure 5-5 Mismatch Warning Message Box



Debugging the Designs

When the designs do not match, Affirma equivalence checker opens the Counterexamples tab (Figure 5-6 on page 34). This tab contains a table of the outputs that do not match, a table of the corresponding inputs, and the *logic cones* that drive the outputs.

Figure 5-6 Counterexamples Tab



Affirma Equivalence Checker Tutorial

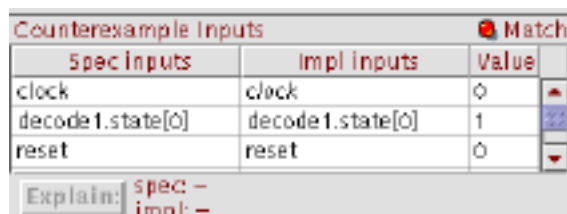
Using Counterexamples

Locating the Appropriate Logic Cone

When you use functional latch mapping, the difference between the designs can sometimes occur within the logic cone of the counterexample. Other times, it can occur in the logic that drives the logic cone. In these cases, you need to generate counterexamples for the upstream logic to locate the difference between the designs.

To locate the appropriate logic cone, you begin by looking at the Inputs table. (See [Figure 5-7](#) on page 35.) The user interface displays a red light, labeled *Match*, when there are inputs that Affirma equivalence checker could not map or when an input in one of the designs does not contribute to the output of the logic cone. Unmapped inputs are displayed in red, and they have no corresponding inputs in the other column of the table. Inputs that do not contribute to the output are displayed in italics.

Figure 5-7 Inputs Table



Counterexample Inputs		Match
Spec Inputs	Impl Inputs	Value
clock	clock	0
decode1.state[0]	decode1.state[0]	1
reset	reset	0

Explain: spec: -
impl: -

When the table shows unmapped inputs, the difference in the designs is likely to occur in upstream logic. When all of the inputs map, the difference is likely to occur in the current logic cone. In this example, all of the inputs have corresponding inputs in the other column of the table. Therefore, the difference between the designs is probably in the current logic cone.

Examining Logic Cones

A logic cone is made up of gate symbols, which are connected in a tree-like structure. Outputs are labeled with a name and a value, either 0, 1, X (for don't-care values), Z (for high-impedance values), or * (for values that do not contribute to the output of the logic cone). If a subcone is displayed as [. . .], it does not contribute to the result of the logic cone, and it is elided. Its output value is displayed as <*>.

You can expand and collapse each branch of the tree (subcone) by clicking on the button that connects the subcone to the tree. Expanding and collapsing subcones can help you to concentrate on specific areas of the logic cone.

By looking at the corresponding subcones in the two designs, you can identify the subcones that contribute to the output, and identify those that cause the mismatch. For example:

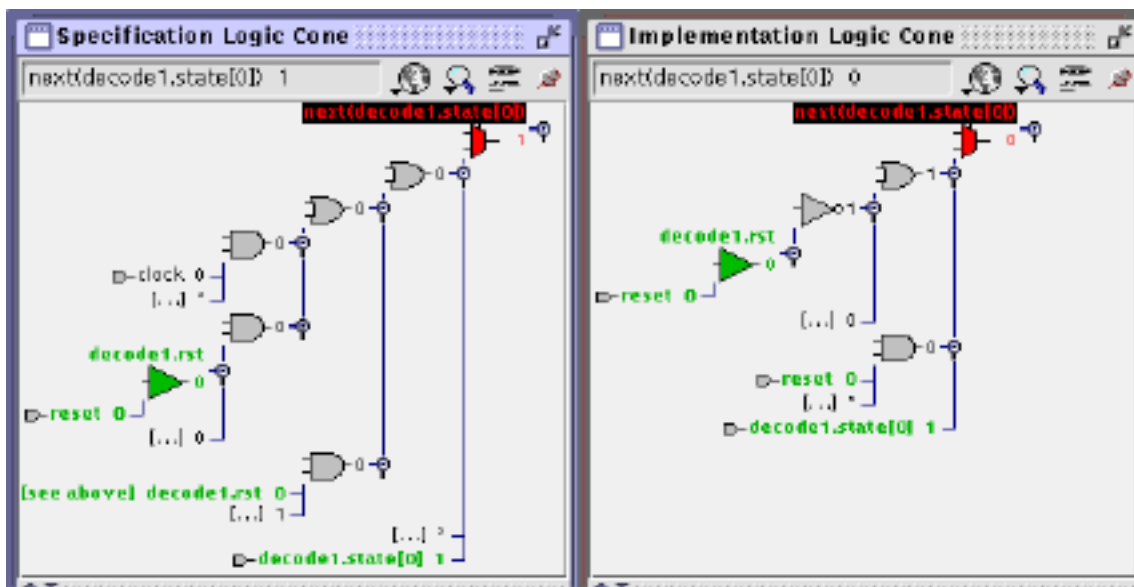
Affirma Equivalence Checker Tutorial

Using Counterexamples

1. Click the down arrowhead at the bottom of the logic cone area for the specification. This enlarges the logic cone display, so that you can see the entire logic cone for this example. Enlarge the implementation logic cone area in the same way.

In [Figure 5-8](#) on page 36, the top-level gate in both designs is a MUX. The first subcone represents the select line. When its value is 1, it selects the second subcone; when its value is 0, it selects the third subcone.

Figure 5-8 Logic Cones for the CPU Design



In the specification, the select value is 0, which selects the third subcone, `decode1.state[0]`. In the implementation, the select value is 1, which selects the second subcone. This subcone is an AND function of the `reset` signal and an elided subgraph. Because a 0 and any other value produces 0, the elided subgraph has no effect on the output value.

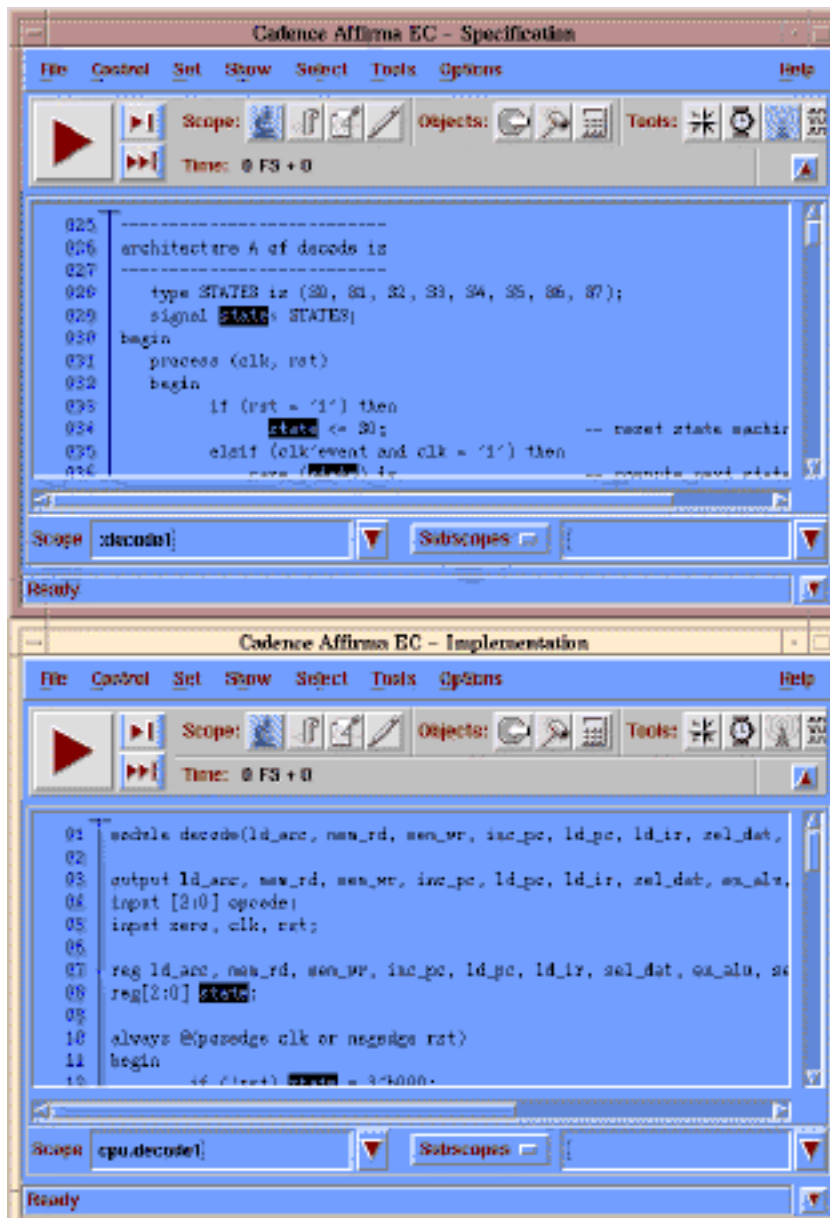
By looking at these logic cones, you can conclude that the specification is loading the data value, and the implementation is in reset mode. It is possible that the reset signals in the two designs are inverted.

2. Click the Source icon in the Main Window tool bar. This opens up two Affirma SimVision source file browser windows, which let you view the source files for these designs. The specification is displayed in the upper window; the implementation is displayed in the lower window.
3. Click on the `decode1.state[0]` signal in either logic cone window, and the Affirma SimVision source file browser displays the portion of the source code in which that signal is defined. (See [Figure 5-9](#) on page 37.)

Affirma Equivalence Checker Tutorial Using Counterexamples

In the specification, the `if` statement is executed if `rst` is 1; the corresponding `if` statement in the implementation is executed if `rst` is 0.

Figure 5-9 Locating the Differences in the Source Files



Affirma Equivalence Checker Tutorial

Using Counterexamples

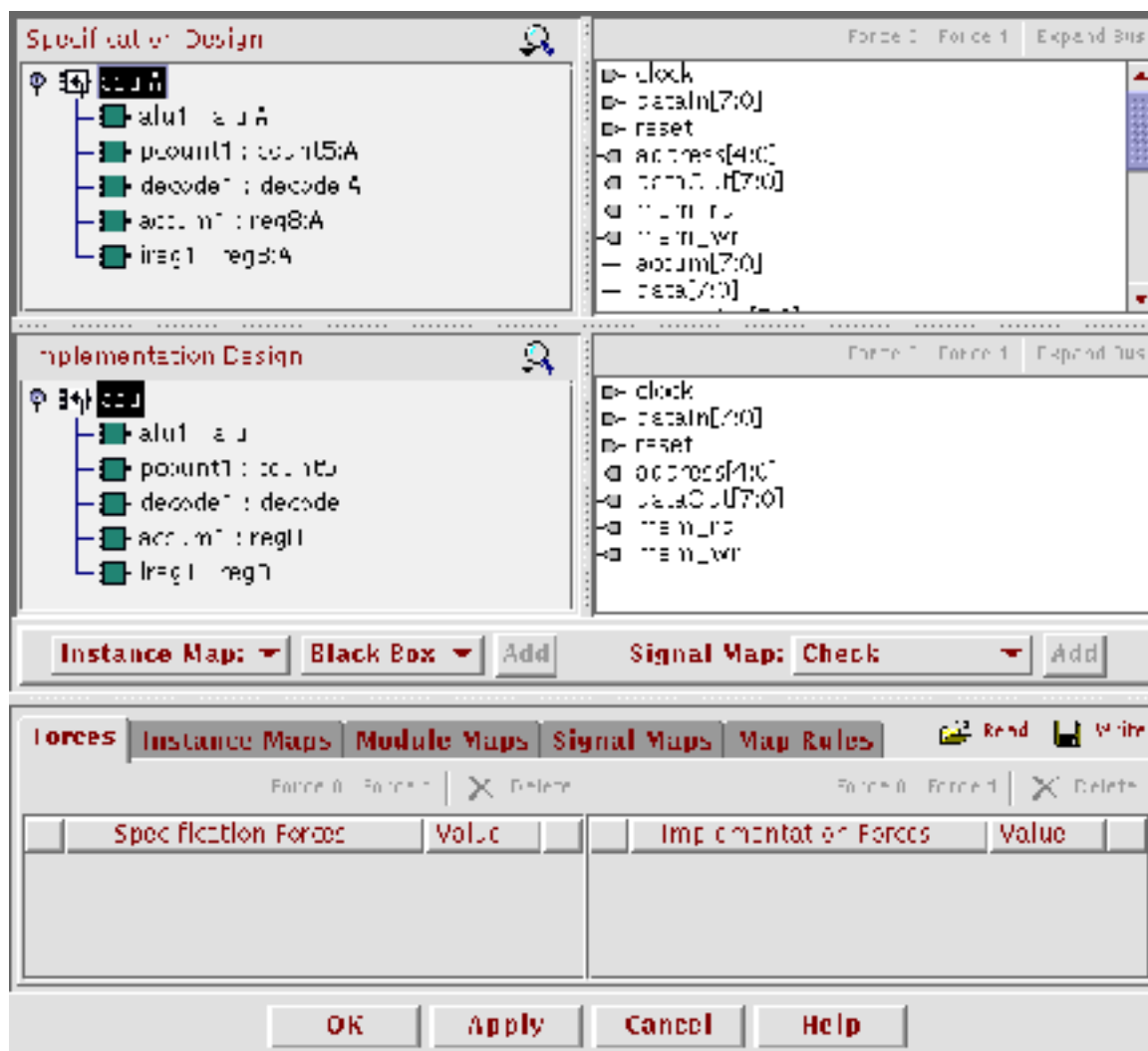
Defining Assumptions

When you know where the designs differ, you can supply Affirma equivalence checker with some assumptions in the form of map commands, and then try to prove that the designs are equivalent based on those assumptions. For example, you can map the `decode1.rst` signal in the specification to the inverse of the `decode1.rst` signal in the implementation, and see if the designs are equivalent under that condition.

To map and invert two signals:

1. Choose *Forces & Maps* from the *Project* menu. Affirma equivalence checker opens the Forces & Maps form. (See [Figure 5-10](#) on page 38.)

Figure 5-10 Forces & Maps Form



Affirma Equivalence Checker Tutorial

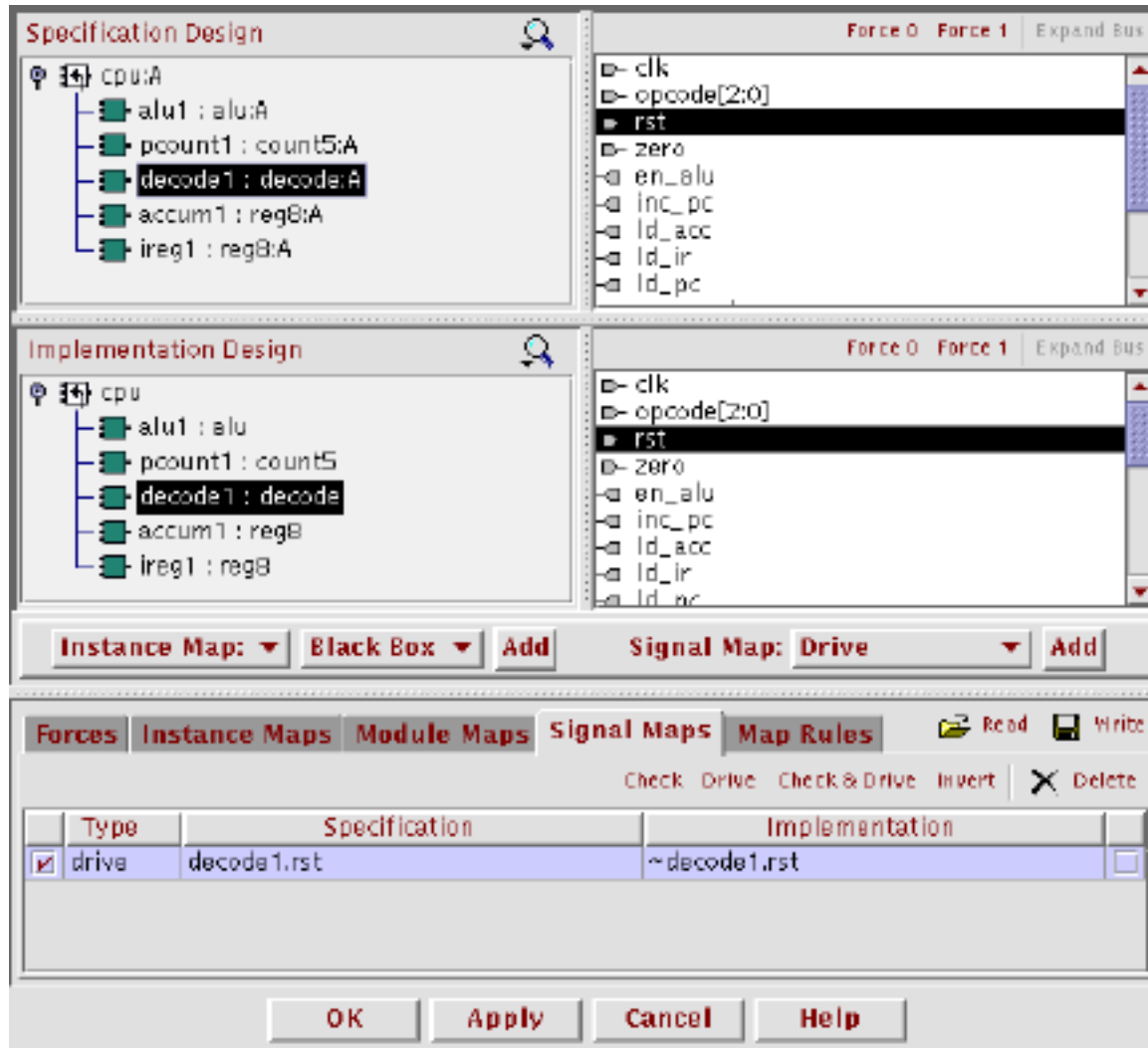
Using Counterexamples

2. Select the `decode1:decode:A` entity from the specification Design Hierarchy area. This shows the inputs and outputs of the modules in the list of specification signals on the right. Then select `decode1:decode` from the implementation Design Hierarchy area. This shows the inputs and outputs of the module in the list of implementation signals.
3. Select the `rst` signal in both the specification and the implementation signal lists.
4. Select *Drive* from the *Signal Map* menu. The drive command causes Affirma equivalence checker to assume that this pair of nets is driven by the same logic. Click *Add* to add the signals to the Signal Maps table at the bottom of the form. However, this command does not invert the signals.
5. Select the row in the Signal Maps table and click *Invert* above the table. This inverts the implementation signal, indicated by the tilde (~) before the signal name. (See [Figure 5-11](#) on page 40.)

Affirma Equivalence Checker Tutorial

Using Counterexamples

Figure 5-11 Inverting a Signal in the Signal Maps Table



6. Click *OK* to apply this map command and to dismiss the Forces & Maps form.
7. Open the Compare tab and click *Compare Designs*. This time, Affirma equivalence checker finds the designs to be equivalent.
8. Exit from Affirma equivalence checker.

Affirma Equivalence Checker Tutorial

Using Counterexamples

More Information

This chapter has shown you how to use the `drive` command and the `invert` operator to provide Affirma equivalence checker with some initial assumptions. You may also want to try the following other commands:

<i>Force</i>	Forces a net to a value, either 0 or 1.
<i>Check</i>	Checks the equivalence of the logic that drives a pair of nets.
<i>Check and drive</i>	Checks the equivalence of the logic that drives a pair of nets, and assumes that the nets are driven by the same logic value.
<i>Black box</i>	Ignores a pair of modules or instances when performing the comparison.
<i>Glass box</i>	Verifies a pair of modules or instances separately from their context.
<i>Pipe</i>	Performs pipeline retiming optimizations on the specified modules or instances.

Glossary

assumption

An assumption is a condition that you apply to your designs during a comparison to determine the conditions under which they prove equivalent. For example, an assumption can map two ports, force a signal to a value, or map two nets.

See also *[map command](#)*.

black box

A black box represents a module that you want to exclude from the comparison, such as a memory module, an IP module, or a module that has not yet been implemented.

comparison directory

The comparison directory is the directory in which you perform a comparison. Affirma equivalence checker places all of its output files in a subdirectory of the comparison directory. When you run the GUI, you can change the comparison directory, but you cannot change your current working directory.

See also *[result directory](#)*.

comparison point

A comparison point is an output, an internal state element, a combinational signal, a module boundary, or a latch in one design that Affirma equivalence checker tries to prove equivalent to a corresponding comparison point in the other design.

counterexample

A counterexample consists of the input values, output values, and the logic cones in both the specification and the implementation that do not match.

See also *[logic cone](#)*, *[specification](#)*, and *[implementation](#)*.

design directory

A design directory is a collection of modules that have been compiled, linked, and initialized. For equivalence checking, the design directory also contains an SMV representation of your design, which forms the basis of the comparison.

Affirma Equivalence Checker Tutorial

Glossary

See also *library name*.

equivalence checking

Equivalence checking is a static verification technique that proves whether two designs are functionally equivalent.

functional latch mapping

When it uses functional latch mapping, Affirma equivalence checker maps latches in one design to latches in the other design based on the functions that these latches perform. The names of latches do not need to be the same in both designs.

glass box

A glass box represents a module that you want to check without regard to the logic that encloses it. Glass boxing represents a more restrictive form of equivalence checking.

implementation

The implementation is one of the designs that is involved in an equivalence check. The implementation is usually derived from the specification in some way. For example, the implementation can be a netlist that you have synthesized from an RTL specification, or a netlist in which you have inserted a scan chain or clock tree, or an ASIC design that you have derived from an FPGA design.

See also *specification*.

instance mapping

Instance mapping is the process by which Affirma equivalence checker determines which instances in the designs should be compared. Affirma equivalence checker supports the following types of instance mapping: black boxing, glass boxing, and pipeline retiming.

See also *black box*, *glass box*, and *pipeline retiming*.

latch

For the purposes of equivalence checking, a latch can be any sequential element, such as a register, flip-flop, memory, or level-sensitive latch.

library name

Design library names for Verilog designs have the following format:

library.cell:view

Affirma Equivalence Checker Tutorial

Glossary

The *library* is the name of the top-level directory that contains the design. The *cell* is the name of the top-level module in the design. The *view* describes the type of cells that are stored in the library. For Affirma equivalence checker, the default view name is *heck*.

Design library names for VHDL designs have the following format:

```
library.entity:entity  
library.entity:architecture
```

The *library* is the name of the top-level directory that contains the design. The *entity* is the name of the top-level entity in the design. The keyword *entity* specifies the most recently elaborated architecture. The second form of the library name lets you specify an architecture name.

logic cone

A logic cone is the combinational logic that drives a comparison point.

See also *comparison point*.

name latch mapping

When you use name latch mapping, Affirma equivalence checker maps latches that have the same names in both designs.

map command

A map command defines the assumptions that Affirma equivalence checker makes when it performs a comparison. There are several types of map commands that force a signal to a value, map nets, and map modules and instances.

See also *assumption*.

map file

A map file is an ASCII file that contains map commands for use with Affirma equivalence checker in command-line mode.

See also *map command*.

mapping

Mapping is the mechanism by which Affirma equivalence checker determines which comparison points to compare.

See also *comparison point*.

Affirma Equivalence Checker Tutorial

Glossary

module mapping

Module mapping is the process by which Affirma equivalence checker determines which modules in the designs should be compared. Affirma equivalence checker supports the following types of module mapping: black boxing, glass boxing, and pipeline retiming.

See also *[black box](#)*, *[glass box](#)*, and *[pipeline retiming](#)*.

pipeline retiming

Pipeline retiming is a sequential optimization that repositions the registers in a circuit. Pipeline retiming can improve the area of a design by reducing the number of registers. It can also improve the timing of a design. Affirma equivalence checker can verify pipeline-retimed designs that have no circularity in their latch dependencies.

project

A project is a collection of information about a pair of designs that you want to compare, such as the pointers to the design directories, the settings that you want to use when creating cycle models, the settings that you want to use when comparing the designs, and the information that the comparison returns. As you work with Affirma equivalence checker, you create a project. You can save a project to a file and load it into Affirma equivalence checker whenever you want to work with the same pair of designs.

reference library

A reference library is a collection of related cells that are used by one or more designs. A cell describes an individual building block of a chip or a system.

result directory

The result directory is the directory in which Affirma equivalence checker places its output files.

See also *[comparison directory](#)*.

SMV file

An SMV file contains the Symbolic Model Verifier (SMV) representation of a design that you want to compare. The `dp` command generates the SMV file, which is the basis for the comparison.

specification

The specification is one of the designs that is involved in an equivalence check. You must fully test the specification before you do equivalence checking. The specification can be

Affirma Equivalence Checker Tutorial

Glossary

a design that is written at the register-transfer level (RTL), a gate-level netlist, or a physical netlist.

See also *implementation*.

Index

A

AND gate
 in logic cone [36](#)
architecture name
 for heck command [10](#)
assumption [42](#)
 see also map command
-AutoLatchInvert option, heck
 command [28](#)

B

black box [42](#)
 command [41](#)

C

CDS_LIC_FILE environment variable [7](#)
check and drive command [41](#)
check command [41](#)
-CmdFile option, heck command [27](#) to [28](#)
Command area
 comparison messages
 mismatch [33](#)
 progress [21](#)
 compilation messages [19](#)
Compare tab
 comparing designs that do not
 match [33](#)
 default settings [21](#)
 loading a compiled design into [20](#), [32](#)
 with map commands [40](#)
comparison [7](#)
 default settings [21](#)
 RTL-to-gate [15](#)
 RTL-to-RTL [9](#)
 using counterexamples [30](#)
 using map commands [26](#)
comparison directory [42](#)
comparison point [42](#)
compilation [7](#)
 with dp command
 Verilog netlist [26](#)

Verilog RTL [10](#)
VHDL RTL [9](#)
 with the GUI [16](#)
Compile tab [16](#)
counterexample [4](#), [8](#), [30](#), [42](#)
Counterexamples tab [34](#)
CPU design [5](#)

D

design directory [7](#), [42](#)
 specifying name in Compile tab [19](#)
Design Hierarchy area [20](#)
 selecting modules in [39](#)
-Design option, dp command
 for Verilog [10](#)
 for VHDL [9](#)
directory structure
 of CPU design [6](#)
don't-care value [35](#)
dp command [7](#)
 for Verilog [10](#)
 for VHDL [9](#)
drive command [39](#)

E

entity
 for dp command [9](#)
entity name
 for heck command [10](#)
environment variables [7](#)
equivalence checking [4](#), [7](#), [43](#)

F

File Selection form [24](#), [31](#)
First-Time User message box [18](#)
force command [27](#), [41](#)
Forces & Maps form [38](#)
functional latch mapping [43](#)
 effect on counterexamples [35](#)

Affirma Equivalence Checker Tutorial

G

gate symbols [35](#)
gate-level netlist
 compiling with the GUI [16](#)
gate-to-gate comparison [4](#)
 comparing with command file [28](#)
 compiling the designs [26](#)
 creating a command file [27](#)
 detecting a mismatch [27](#)
glass box [43](#)
 command [41](#)
graphical user interface
 see GUI
GUI [8](#)
 compiling gate-level netlist [16](#)
 defining map commands with [27](#)
 exiting [23](#)
 other features [24](#)
 Preferences menu [31](#)

H

heck command [8](#), [10](#)
 creating command file for [27](#)
 extended options in Compare tab [21](#)
 other options [14](#)
 for mapping objects [28](#)
heck -gui command [16](#), [30](#)
heck.project file [24](#)
heck.report file [11](#)
 viewing in Results tab [22](#)
Help menu [25](#)
-Help option, heck command [14](#)
high-impedence value [35](#)

I

-Impl option, heck command [10](#)
implementation [4](#), [43](#)
 Compile tab [16](#)
Inputs table, Counterexamples tab [34](#)
instance mapping [27](#), [43](#)
invert button
 in Signal Maps form [39](#)
~ invert operator [39](#)

K

-Keep option, heck command [14](#)

L

latch [43](#)
 mapping in Compare tab [21](#)
Latch Maps tab [22](#)
-LatchMapMethod, heck command [28](#)
LD_LIBRARY_PATH environment
 variable [7](#)
libprep command [7](#), [15](#)
library [7](#)
 see also reference library
library name [43](#)
logic cone [44](#)
 Counterexamples tab [34](#)
 examining [35](#)
 locating the appropriate [35](#)
Look in menu [31](#)

M

Main window [16](#)
map command [4](#), [27](#), [44](#)
 defining [38](#)
 disabling scan logic [26](#)
 other [41](#)
map file [44](#)
map/map.detail file
 for RTL-to-RTL comparison [12](#)
mapping [27](#), [44](#)
 results in map/map.detail file [12](#)
 signals [38](#)
message box
 First-Time User [18](#)
 Mismatch warning [33](#)
 Project File [30](#)
 Save a Project File [23](#)
 Tip of the Day [18](#)
messages
 comparison [11](#)
 in the GUI [21](#)
 mismatch [27](#), [33](#)
 compilation
 dp command [9](#)
 in the GUI [19](#)

Affirma Equivalence Checker Tutorial

library preparation [15](#)
Mismatch Warning message box [33](#)
module mapping [27](#), [45](#)
module, top-level
 for dp command [10](#)
 for heck command [10](#)
 in Compile tab [19](#)

N

name latch mapping [44](#)
name-mapping rule [27](#)
net
 driving in Forces & Maps form [39](#)
 mapping [27](#)

O

Options menu [24](#)
 creating a new project on startup [31](#)
Outputs table, Counterexamples tab [34](#)

P

PATH environment variable [7](#)
pipe command [41](#)
project [17](#), [45](#)
 map commands in [27](#)
 saving [23](#)
Project File message box [30](#)
Project menu
 Forces & Maps choice [38](#)

R

reference library [7](#), [45](#)
 preparing [15](#)
 specifying path in Compile tab [19](#)
Reports menu [25](#)
result directory [45](#)
 in Compare tab [21](#)
-ResultDir option, heck command [14](#)
Results tab [22](#)
RTL-to-gate comparison [4](#), [15](#)
 comparing the designs [21](#)
 compiling the implementation [16](#)
 loading the specification into the

 GUI [20](#)
 preparing the reference library [15](#)
RTL-to-RTL comparison [4](#)
 comparing designs [10](#)
 compiling Verilog [10](#)
 compiling VHDL [9](#)

S

Save a File message box [23](#)
scan chain
 disabling logic [26](#)
 forcing to 0 [27](#)
Select Design form [20](#)
SHLIB_PATH environment variable [7](#)
signal
 forcing [27](#)
 inverting [38](#)
Signal Maps table [39](#)
-Similar option, heck command [29](#)
SimVision source file browser window [36](#)
SMV file [45](#)
 creating for VHDL [9](#)
-Spec option, heck command [10](#)
specification [4](#), [45](#)
 Compile tab [16](#)
synthesizable cell [7](#)

T

Tip of the Day message box [18](#)
-Top option, dp command
 for Verilog [10](#)
 for VHDL [9](#)

V

-v option, libprep command [15](#)
Verilog
 compiling [10](#)
 design library name [43](#)
 specifying files in Compile tab [18](#)
-Version option, heck command [14](#)
VHDL
 Compile tab [31](#)
 compiling [9](#)
 design library name [44](#)

Affirma Equivalence Checker Tutorial

W

-work option, libprep command [15](#)