Formality[®] User Guide

Version O-2018.06, June 2018

SYNOPSYS®

Copyright Notice and Proprietary Information

©2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

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.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

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

Copyright Notice for the Command-Line Editing Feature

© 1992, 1993 The Regents of the University of California. All rights reserved. This code is derived from software contributed to Berkeley by Christos Zoulas of Cornell University.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- 2.Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- 3.All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright Notice for the Line-Editing Library

© 1992 Simmule Turner and Rich Salz. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

- 1. The authors are not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
- 2.The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
- 3.Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
- 4. This notice may not be removed or altered.

Customer Support
Introduction to Formality
What is Formal Verification?
General Verification Process
Verifying Designs by Equivalence Checking
Setting Up Designs to Preempt Differences Concept of Guidance Concept of Black Boxes Concept of Constraints
Matching
Verification

2.	Formality Use Model	
	Formality Process Flow	2-2
	Starting Formality	2-3
	Guidance	2-3
	Loading Designs	2-4
	Performing Setup	2-4
	Matching Compare Points	2-5
	Verifying and Interpreting Results	2-
	Debugging	2-6
	Tutorial	2-6
	Library Verification Mode	2-6
3.	Invocation	
	Introduction	3-3
		3-3
		3-3
	Basic Usage	3-3
	Invoking the Formality Shell	3-4
	Synopsys Setup File	3-5
		3-6
	Invoking the Formality GUI	3-6
	Interrupting Formality	3-7
	Viewing CPU Statistics	3-7
	Getting Help	3-8
	Advanced Usage	3-9
		-1(
	Entering Commands	-1(
	Argument Lists	-1(
	•	-12
	,	-12
		-14
		-15 -16
	3	
	GUI Environment	-16

	Windows
	Prompt
	Copying Text
	Saving the Transcript
	Script Files
	Messages
	Controlling Message Types
	Output Files
4.	Tutorial
	Before You Start
	Creating Tutorial Directories
	Tutorial Directory Contents
	Invoking the Formality Shell
	Verifying fifo.vg Against fifo.v
	Loading the SVF File
	Specifying the Reference Design
	Specifying the Implementation Design
	Setting Up the Design
	Matching Compare Points
	Verifying the Designs
	Debugging
	Graphical User Interface
	Verifying fifo_with_scan.v Against fifo_mod.vg
	Verifying fifo_jtag.v Against fifo_with_scan.v
	Debugging Using Diagnosis
	For More Information
5.	Load Guidance
	What is Guidance?
	Basic Usage
	Creating an SVF File
	Using the Automated Setup Mode

	Reading the SVF File	5-6
	Generating Formality Verification Setup Scripts	5-7
	Understanding the Guidance Summary	5-10
	Advanced Usage	5-11
	Guidance Directory and File Structure	5-11
	Guidance Reports	5-12
	SVF File Diagnostic Messages	5-13
	Reading in Multiple Guidance Files	5-13
	Checkpoint Guidance	5-14
6.	Loading Designs	
	Introduction	6-3
	Loading Design	6-3
	Top-Level Design	6-5
	Concept of Containers	6-5
	Basic Usage	6-7
	Loading the Reference Design	6-7
	Reading Technology Libraries	6-7
	Reading Designs	6-8
	Setting the Top-Level Design	6-11
	Loading the Implementation Design	6-12
	Advanced Usage	6-12
	Reading Technology Libraries	6-13
	Using the 'celldefine Verilog Attribute	6-13
	Reading SystemVerilog, Verilog, and VHDL Cell Definitions	6-13
	Verilog Simulation Data	6-14
	Library Loading Order	6-15
	Setting the Top-Level Design	6-16
	Setting Parameters on the Top-Level Design	6-16
	Generating Simulation or Synthesis Mismatch Report	6-17
	Linking the Top-Level Design Automatically	6-17
	Setting Up and Managing Containers	6-18
	Variables Controlled by Setup Free Flow	6-19
	Variables to Control Bus Names	6-19
	Variables to Control Case Behavior	6-20 6-20
	VAUADIES ID COUITOLCASE DEUAVIOI	ひ-/ロ

7. Performing Setup

ommon Operations	
Black Boxes	
	aces
Marking a Design as a	a Black Box for Verification
Reporting Black Boxe	es
	hecks
Setting Pin and Port D	Directions for Unresolved Black Boxes
Specifying Constants	
Defining Constants	
Removing User-Defin	ed Constants
Listing User-Defined (Constants
_	JS
	Constraint
_	Type
Removing an Externa	Il Constraint
	nt Type
	Information
	About Constraint Types
Combinational Design Cha	anges
Disabling Scan Logic	
Disabling Boundary S	Scan in Your Designs
Managing Clock Tree	Buffering
Sequential Design Change	es
Verifying Clock-Gate I	Designs Automatically
Enabling an Inversion	Push
Instance-Based Invers	sion Push
Environmental Inversi	ion Push
Retimed Designs	
Retiming Using Desig	n Compiler
Retiming Using Other	Tools
Loading the UPF File	
Controlling the Interpr	etation of the UPF Files
	With the UPF File
Reporting Over-Const	trained Supply Nets
• •	ch Cells
Verifying Hierarchical	Designs Using Power-Aware Black Boxes

Contents ix

	Verifying Hierarchical Designs Using Power Models	7- 7-
	Less Common Operations	7-
	Managing Asynchronous Bypass Logic	7-
	Asynchronous State-Holding Loops	7-
	Re-encoded Finite State Machines	7-
	SVF file for FSM Re-encoding	7-
	Reading a User-Supplied FSM State File	7
	Defining FSM States Individually	7
	Multiple Re-encoded FSMs in a Single Module	7
	Listing State Encoding Information	7
	FSMs Re-encoded in Design Compiler	7
	Hierarchical Designs	7
	Setting the Flattened Hierarchy Separator Character	7
	Propagating Constants	7
	Nets With Multiple Drivers	7
	Retention Registers Outside Low-Power Design Flow	7
	Single State Holding Elements	7
	Multiplier Architectures	7
	Setting the Multiplier Architecture	7
	Reporting Your Multiplier Architecture	7
	Multibit Library Cells	7
	Preverification	7
8.	Matching Compare Points	
	Introduction	
	Basic Usage	
	Performing Compare Point Matching	
	Reporting Unmatched Points	
	Advanced Usage	
	Advanced Usage	
	Debugging Unmatched Points	
	Undo Matched Points	
	How Formality Matches Compare Points	
	Exact-Name Matching	8
	Name Filtering	8
	Topological Equivalence	8
	ropological Equivalence	C

Contents x

	Signature Analysis	8-11
	Compare Point Matching Based on Net Names	8-12
	Commands and Variables That Cannot be Changed in Match Mode	8-13
9.	Verifying the Design and Interpreting Results	
	Basic Usage	9-3
	Verifying a Design	9-3
	Reporting and Interpreting Results	9-5
	Interrupting Verification	9-7
	Introduction	9-8
	Advanced Usage	9-8
	Saving the Session Information	9-8
	Setting a Threshold to Save Session Files	9-9
	Verifying a Single Compare Point	9-10
	Controlling Verification Runtimes	9-11
	Verification Using Multicore Processing	9-11
	Performing Hierarchical Verification	9-12
	Verifying Feedthroughs in Hierarchical Subdesigns	9-14
	Verifying ECO Designs	9-17
	Modifying the SVF File	9-17
	Uninstantiated Designs in Verilog Libraries	9-20
	Using Batch Jobs	9-20
	Starting Verification Using Batch Jobs	9-20
	Controlling Verification During Batch Jobs	9-21 9-21
	Verification Progress Reporting for Batch Jobs	9-21
	Verifying Blocks Under a Certain Level Independently	
	Removing Compare Points From the Verification Set	9-22
	Verification Using Checkpoint Guidance	9-23
	Controlling the Checkpoint Verification Flow	9-23 9-23
	Known Limitations	9-24
	Identifying Inferred Register Names With Register Mapping	9-24
10.	Debugging Verification	
. • •		10-3
	Introduction	10-3
	Debug Process Flow	10-3
	CACHELLING HINCHIANON	10-5

Contents xi

Debugging a Failing Verification	10-5
Determining Unread Failing Compare Points	10-6
Determining Failure Causes	10-7
Debugging Using Diagnosis	10-8
Debugging Using Logic Cones	10-9
Eliminating Setup Possibilities	10-11
Black Boxes	10-11
Unmatched Points	10-11
Design Transformations	10-19
Schematics	10-20
Viewing Schematics	10-20
Traversing Design Hierarchy	10-24
Finding an Object	10-25
Generating Lists	10-26
Zooming In and Out of a View	10-27
Viewing RTL Source Files in the Design Browser	10-28
Hierarchical Design Browser	10-28
Queuing Setup Commands	10-31
Logic Cones	10-31
Viewing Combinational Feedback Loops	10-36
Pruning Logic	10-36
Grouping Hierarchy in a Logic Cone	10-37
Setting Probe Points	10-38
Multicolor Highlighting	10-39
Cell Coloring	10-39
Viewing, Editing, and Simulating Patterns	10-40
Debugging a Hard Verification	10-42
Checking the Guidance Summary	10-44
Creating a List of Hard Points	10-44
Determining the Cause of Hard Points	10-45
Analyzing Fanin Logic Cones of a Hard Compare Point	10-46
Alternate Strategies to Resolve Hard Verifications	10-50
Verifying Designs Using Alternate Strategies	10-51
Verifying Designs Using an Alternate Strategy Manually Verifying Designs by Automated Parallel Deployment of	10-51
Alternate Strategies	10-53

Contents xii

Formality[®] User Guide

11. Using Formality Ultra

The Formality Ultra Flow	11-3
Analyzing Differences Between the RTL and the Netlist	11-5 11-5
Finding Equivalent Nets	11-6 11-7
Modifying the Implementation Design	11-8 11-9
Using High-Level Editing Commands	11-11 11-12 11-13 11-13
Default Names for Nets, Cells, and Ports	11-13 11-15 11-15
Using Edit Files	11-16 11-16
Loading Edit Files	11-16 11-16 11-17 11-17
Displaying Modifications to the Design	11-17 11-18 11-18
Verifying ECO Modifications	11-20 11-22 11-22
Exporting ECO Modifications	11-22
Integration With Verdi nECO	11-25 11-26 11-26 11-26 11-27
Integration With the IC Compiler Tool	11-27 11-27 11-28

Contents xiii

	RTL Cross-Probing	11-28
12.	Library Verification Mode	
	Introduction	12-2
	Library Verification Mode	12-3
	Loading the Reference Library	12-4
	Loading the Implementation Library	12-5
	Listing the Cells	12-5
	Specifying a Customized Cell List	12-6
	Elaborating Library Cells	12-7
	Performing Library Verification	12-7
	Reporting and Interpreting Verification Results	12-9
	Debugging Failed Library Cells	12-10
_		
App	pendix A. Querying Design Objects and Collections	
	Lifetime of a Collection	A-2
	Iteration	A-2
	Managing Collections Using Commands	A-3
	Filtering	A-4
	Sorting Collections	A-5
	Implicit Query of Collections	A-5
	The Collections Manager GUI	A-8
	Creating Collections	A-9
	Filtering Collections	A-11
	Operating on Collections	A-13
	Finding a Design Object in a Collection	A-14
App	pendix B. Tcl Syntax as Applied to Formality Shell Commands	
	Using Application Commands	B-3
	Summary of the Command Syntax	B-3
	Using Special Characters	B-4

Contents xiv

Using Return Types	B-4
Quoting Values	B-5
Using Built-In Commands	B-5
Using Procedures	B-6
Using Lists	B-6
Using Other Tcl Utilities	B-7
Using Environment Variables	B-8
Nesting Commands	B-9
Evaluating Expressions	B-9
Using Control Flow Commands Using the if Command Using while and for Loops Using while Loops Using for Loops. Iterating Over a List: foreach Terminating a Loop: break and continue Using the switch Command	B-9 B-10 B-10 B-11 B-11 B-12 B-12
Creating Procedures Setting Defaults for Arguments	B-12 B-13 B-13

Index

Contents xv

Formality[®] User Guide Version O-2018.06

Contents xvi

Preface

This preface includes the following sections:

- About This User Guide
- Customer Support

Formality[®] User Guide Version O-2018.06

About This User Guide

The *Formality User Guide* provides information about Formality concepts, procedures, file types, menu items, and methodologies with a hands-on tutorial to get you started with the tool.

Audience

The Formality User Guide provides information about Formality concepts, procedures, file types, menu items, and methodologies with a hands-on tutorial to get you started with the tool.

Additionally, you need to understand the following concepts:

- Logic design and timing principles
- Logic simulation tools
- Linux operating system

Related Publications

For additional information about the Formality 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:

- Design Compiler[®]
- HDL Compiler™
- PrimeTime[®] Suite
- ESP

Release Notes

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

Formality[®] User Guide Version O-2018.06

To see the Formality Release Notes,

 Go to the SolvNet Download Center located at the following address: https://solvnet.synopsys.com/DownloadCenter

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

Formality® User Guide

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 <code>design_list</code> .
Courier bold	Indicates user input—text you type verbatim—in examples, such as
	<pre>prompt> write_file top</pre>
[]	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.

Formality[®] User Guide Version O-2018.06

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 https://www.synopsys.com/support/global-support-centers.html
- Telephone your local support center.
 - o Call (800) 245-8005 from within North America.
 - Find other local support center telephone numbers at https://www.synopsys.com/support/global-support-centers.html

Formality[®] User Guide

1

Introduction to Formality

This chapter introduces you to the Formality application. It includes the following sections:

- What is Formality?
- General Verification Process
- Verifying Designs by Equivalence Checking
- Interpreting Results

What is Formality?

The purpose of Formality is to detect unexpected differences that might have been introduced into a design during development. It uses a formal verification comparison engine to prove or disprove the equivalence of two given designs and presents any differences for follow-on detailed analysis.

What is Formal Verification?

Formal verification is an alternative to verification through simulation. Verification through simulation applies a large number of input vectors to the circuit and then compares the resulting output vectors to expected values. As designs become larger and more complex and require more simulation vectors, regression testing with traditional simulation tools becomes a bottleneck in the design flow.

The bottleneck is caused by these factors:

- Large numbers of simulation vectors are needed to provide confidence that the design meets the required specifications.
- Logic simulators must process more events for each stimulus vector because of increased design size and complexity.
- More vectors and larger design sizes cause increased memory swapping, thereby slowing down performance.

Formal verification uses mathematical techniques to compare the logic to be verified against either a logical specification or a reference design. Unlike verification through simulation, formal verification does not require input vectors. As formal verification considers only logical functions during comparisons, it is independent of the design's physical properties, such as layout and timing.

The real strength of formal verification is its ability to reveal unexpected differences without relying on vector sets, which verifies large designs faster than simulation while providing 100 percent coverage.

Formal verification consists of two different basic tools: equivalence checkers and model checkers. Equivalence checkers prove or disprove that one design representation is logically equivalent to another. That is to say, they are used to prove that two circuits exhibit the same exact behavior under all conditions despite different representations. They do this using Formal methods and require no simulation vectors. Formality is an Equivalence checker.

Model checkers prove or disprove that a design adheres to a specified set of logical properties.

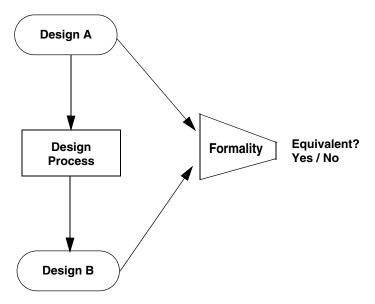
General Verification Process

Formality elaborates and compares two sets of design files before and after some design methodology process has been carried out. Formality is used throughout the design flow to make sure that the integrity of the design descriptions are still logically equivalent as they go through different representations.

Individual Verification

Figure 1-1 shows the basic flow for the verification of a single design process. Formality reads in the files representing the reference Design A, and does the same for the implementation Design B. In doing this, it establishes which points in the design are candidates to be compared, matches them between the two designs as appropriate, and performs the formal equivalence check, reporting back any differences that are detected.

Figure 1-1 Verification Flow Using Formality



ASIC Verification Flow

Each individual verification is just one of many that are performed during a general ASIC verification flow. The following diagram shows how this verification chain parallels that of the design process, originating from the initial RTL description.

Figure 1-2 shows the ASIC verification flow using Formality.

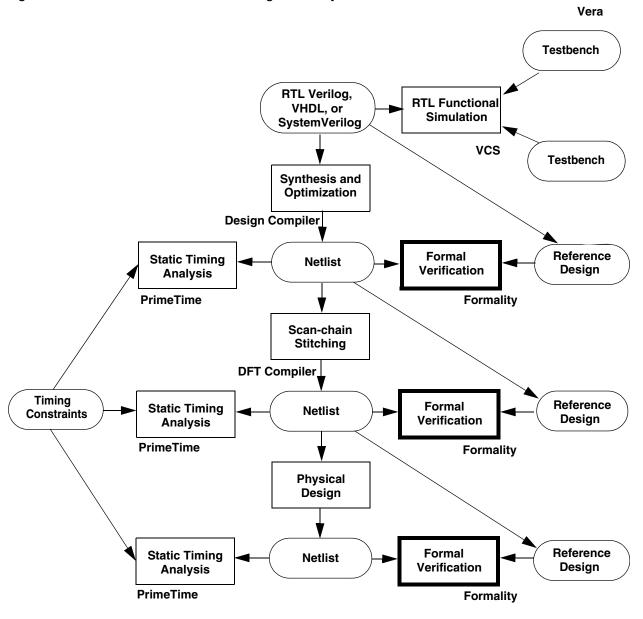


Figure 1-2 ASIC Verification Flow Using Formality

As this design flow accumulates details, the verification chain ensures that each new representation of the design is free of unexpected changes.

Verifying Designs by Equivalence Checking

Design verification using equivalence checking is a four-phase process:

- 1. Read and elaborate language descriptions into logical representations
- 2. Set up to preempt differences
- 3. Map corresponding signals between pairs of designs (Matching)
- 4. Compare the logic cones that drive the mapped signals (Verification)

Reading and Elaborating Designs

Formality begins a verification by reading a set of user-defined design and library files and elaborates them into a format ready for equivalency checking that fully represents the logic of the user-defined top-level model. During this phase, you establish the reference and implementation designs, along with corresponding compare points and logic cones.

Concept of Reference and Implementation Designs

The reference design and implementation design are tested by Formality for equivalence.

Reference design	This design is the golden design, the standard against which Formality tests for equivalence.
Implementation design	This design is the changed design. It is the design whose correctness you want to prove. For example, a newly synthesized design is an implementation of the source RTL design.

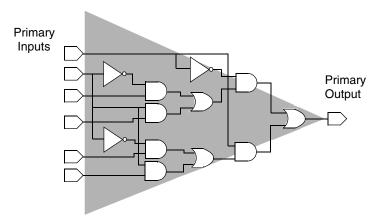
After Formality proves the equivalence of the implementation design to a known reference design, you can establish the implementation design as the new reference design. Using this technique during regression testing keeps overall verification times to a minimum. Conversely, working through an entire design methodology and then verifying the sign-off netlist against the original RTL can result in difficult verifications and in longer overall verification times.

In the Formality command-line interface, fm_shell, or GUI environment, you can designate a design you have read into Formality as the implementation or as the reference design. There are no special requirements to restrict your designation. However, at any given time, you can have only one implementation design and one reference design in the Formality environment.

Concept of Logic Cones

A logic cone consists of combinational logic originating from a specific design object and fanning backward to terminate at certain design object outputs. The design objects where logic cones originate are those used by Formality to create compare points. Compare points are primary outputs, internal registers, black box input pins, or nets driven by multiple drivers where at least one driver is a port or black box. The design objects at which logic cones terminate are primary inputs or compare points. Figure 1-3 illustrates the logic cone concept.

Figure 1-3 Logic Cone



In Figure 1-3, the compare point is a primary output. Formality compares the logic function of this primary output to the logic function of the matching primary output in another design during verification. The shaded area of the figure represents the logic cone for the primary output. The cone begins at the input net of the port and works back toward the termination points. In this illustration, the termination points are nets connected to primary inputs.

Setting Up Designs to Preempt Differences

There might be intended functional differences in the two designs being compared. In these cases, perform setup to account for these differences to avoid false-failures. An example is adding scan logic to the implementation design. You can still check that the non scan functionality of the implementation design matches that of the reference design by setting a constant in the implementation design that disables the scan logic.

Concept of Guidance

Guidance helps an equivalence-checking tool to understand and process design changes caused by other tools that were used in the design flow. Formality uses guidance information to assist compare-point matching, set up verification correctly without user intervention, and understand complex arithmetic transformations better.

Concept of Black Boxes

A black box is an instance of a design whose function is unknown. Black boxes are commonly used for components of a design that are not synthesized. Examples of common black boxes include RAMs, ROMS, analog circuits, and hard IP blocks. The inputs to black boxes are treated as compare points and the outputs of the black boxes are treated as input points to other logic cones.

When black boxes are used in equivalence checking, it is important to make sure that there is a one-to-one mapping in the reference and implementation design; otherwise compare point result in failures. You can specify how the tool handles black boxes. These techniques are outlined in "Black Boxes" on page 7-3.

Concept of Constraints

You can limit the number of input value combinations that are considered during verification in Formality by setting external constraints. Setting constraints can be useful to reduce verification time and eliminate potential false failures that can result from verification that considers unused or illegal combinations of input values.

By setting constraints on the allowed values of and relationships between primary inputs, registers, and black box outputs and by allowing the verification engine to use this information, the resulting verification is restricted to identifying only those differences between the reference and implementation designs that result from the allowed states.

For more information about constraints, see "External Constraints" on page 7-11.

Matching

Prior to design verification, Formality tries to match each primary output, sequential element, black box input pin, and qualified net in the implementation design with a comparable design object in the reference design. For more information about how compare points are matched, see Chapter 8, "Matching Compare Points."

For Formality to perform a complete verification, all compare points must be verifiable. There must be a one-to-one correspondence between the design objects in the reference and implementation designs. There are cases, however, that do not require a one-to-one correspondence to attain complete verification when you are testing for design consistency.

For example,

- An implementation design that contains extra primary outputs.
- Either the implementation design or reference design contains extra registers, and no compare points fail during verification.

Compare points are primarily matched by object names in the designs. If the object names in the designs are different, Formality uses various methods to match up these compare points automatically. You can also manually match these object names when all automatic methods fail.

Concept of Compare Points

A compare point is a design object used as a combinational logic endpoint during verification. A compare point can be an output port, register, latch, black box input pin, or net driven by multiple drivers.

Formality uses the following design objects to create compare points automatically:

- Primary outputs
- Sequential elements
- Black box input pins
- Nets driven by multiple drivers, where at least one driver is a port or black box

Formality verifies a compare point by comparing the logic cone from a compare point in the implementation design against a logic cone for a matching compare point from the reference design, as shown in Figure 1-4.

Concept of Name-Based and Non Name-Based Matching

Compare-point matching techniques in Formality can be broadly divided into two categories:

- Name-based matching techniques
- Non name-based matching techniques

Unmatched design objects from either the implementation or reference design are reported as failing compare points, with a note indicating that there is no comparable design object in the reference design.

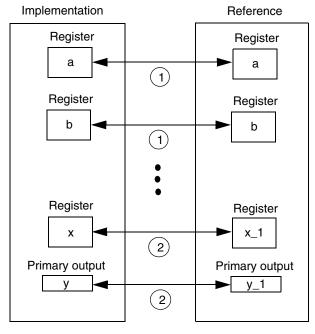
Sometimes you might have to provide information so that Formality can match all design objects before performing verification. For example, the implementation and reference designs might contain design objects that differ in name but are otherwise comparable. However, Formality is not able to match them by using its matching algorithms, including signature analysis. In such cases, you can map design object names yourself using several methods. For more information about matching design objects with different names, see "Debugging Unmatched Points" on page 8-6.

Figure 1-4 shows an example of how the combination of automatic and user-defined compare points results in complete verification. Automatically created compare points result when Formality can match the name and type of two design objects by using normal

matching techniques or signature analysis. User-defined compare points result when you take steps to map names between design objects.

Figure 1-4 Constructing Compare Points

- 1 Automatically defined compare points
- (2) User-defined compare points



For compare point status messages, see "Reporting and Interpreting Results" on page 9-5.

Concept of User Matches

Formality automatically matches as many ports and components as possible between the implementation design and reference design during verification. If these automatic methods fail to determine a match, you can use commands to create these matches manually.

For example, the implementation and reference designs might contain design objects that differ in name but are otherwise comparable. However, Formality is not able to match them by using its matching algorithms, including signature analysis. In such cases, you can map design object names yourself using several methods. For more information about matching design objects with different names, see "Matching With User-Supplied Names" on page 10-12.

Verification

Verification is the primary function of equivalence checking. By default, Formality checks for design consistency when you verify a design or technology library.

Concept of Consistency and Equality

The term *design equivalence* refers to the verification test objective. Formality can test for two types of design equivalence: design consistency and design equality.

Design Consistency

For every input pattern for which the reference design defines a 1 or 0 response, the implementation design gives the same response. If a don't care (X) condition exists in the reference design, verification passes if there is a 0 or a 1 at the equivalent point in the implementation design.

Design Equality

Includes design consistency with additional requirements. The functions of the implementation and reference designs must be defined for exactly the same set of input patterns. If a don't care (X) condition exists in the reference design, verification passes only when there is an X at the equivalent point in the implementation design.

Interpreting Results

When functions defining the cones of logic for a matched pair of compare points (one from the reference design and one from the implementation design) are proved by Formality to be functionally equivalent, the result is that the compare points in both the reference and implementation designs have a passing status. If all compare points in the reference design pass verification, the final verification result for the entire design is a successful verification.

2

Formality Use Model

The Formality use model follows the same flow as the general verification process discussed in Chapter 1, "Introduction to Formality" and Figure 2-1 illustrates it.

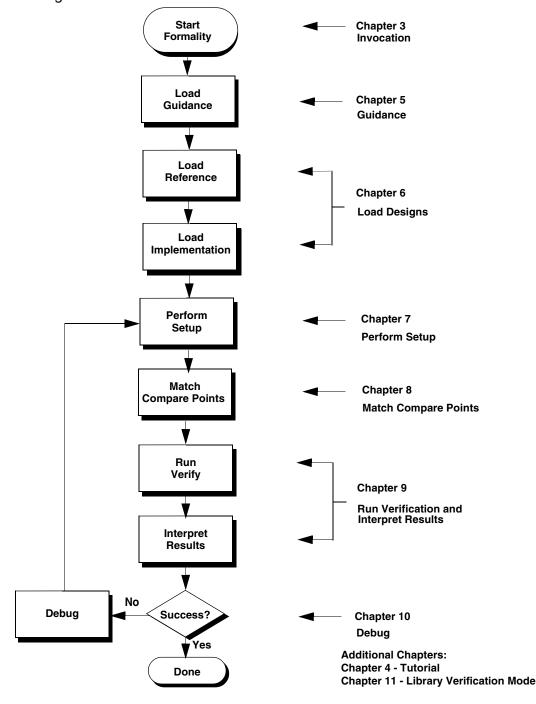
This chapter includes the following sections:

- Formality Process Flow
- Starting Formality
- Guidance
- Loading Designs
- Performing Setup
- Matching Compare Points
- · Verifying and Interpreting Results
- Debugging
- Tutorial
- Library Verification Mode

Formality Process Flow

Figure 2-1 outlines the design verification process flow for Formality.

Figure 2-1 Design Verification Process Flow Overview



Formality[®] User Guide Version O-2018.06

Figure 2-1 represents the steps specific to performing an equivalence check using Formality. Each of the following chapters of this user guide describes one or more steps, first providing the basics and then elaborating with more detail.

Starting Formality

Enter the Formality environment by typing fm_shell at the Linux prompt. You can use the quit or exit commands at any time to exit.

```
% fm_shell
...
fm_shell (setup)>
```

The word (setup) indicates the mode that you are currently in when using commands. The modes that are available are guide, setup, match, and verify. When you invoke Formality, you begin in the setup mode.

Note:

You can also invoke the GUI from the shell command prompt at this point using the start qui command.

You must first set up environment variables, paths, and licenses to do this. These topics, along with other invocation options and basic shell features, are discussed in detail in Chapter 3, "Invocation."

Guidance

The load guidance step of the Formality process flow is the point at which you can opt to provide setup information about design changes caused by other tools used in the design flow.

```
% fm_shell
...
fm_shell (setup)> set_svf design.svf
```

Files containing this guidance information are known as SVF files, and they generally have the .svf extension. An SVF file enables the tool to process the content and store data for use during the matching step that follows. Guidance is recommended in a Synopsys design implementation flow, while it is optional when verifying designs that are modified by third-party tools.

For further information about guidance, see Chapter 5, "Load Guidance."

Formality[®] User Guide Version O-2018.06

Loading Designs

In order for Formality to perform verification, you must first provide it with two designs. The golden design, the one that is known to be functionally correct, is called the reference design. The second design is called the implementation design; it is a modified version of the reference design that you want to verify as functionally equivalent to the reference design.

```
% fm_shell
...
fm_shell (setup)> read_verilog -r top.v
```

Formality can be used to verify two RTL designs against each other, two gate-level designs against each other, or an RTL design against a gate-level design.

The design files that you load into Formality can use only synthesizable SystemVerilog, Verilog, or VHDL constructs or can be in the Synopsys internal database format (.db, .ddc, or Milkyway database).

After designs are loaded into Formality in this step of the process flow, you can control certain aspects of the verification process, such as establishing environmental parameters.

For further information about loading and managing designs, see Chapter 6, "Loading Designs."

Performing Setup

The setup step involves supplying information to Formality to account for design-specific issues that were not taken care of automatically with the guidance step.

```
% fm_shell
...
fm_shell (setup)> set_constant -type port r:/WORK/top/scanmode 0
```

The types of design transformations that might need setup include internal scan, boundary scan, clock-gating, finite state machine (FSM) re-encoding, black boxes, and pipeline retiming. You can use the setup information to accurately verify the designs that have been transformed in a way that would otherwise cause them to be reported as nonequivalent.

For more information about setup possibilities, see Chapter 7, "Performing Setup."

Matching Compare Points

The match compare points step in the Formality flow is the process by which Formality attempts to match each compare point in the reference design with a corresponding compare point in the implementation design.

```
% fm_shell
...
fm_shell (setup)> match
```

Accurate matching is required for accurate verification. Matching ensures that there are no mismatched logic cones and verifies the implementation design for functionality.

For further information about matching compare points, see Chapter 8, "Matching Compare Points."

Verifying and Interpreting Results

The verification step of the Formality process flow follows when all the loading, setup, and compare point matching steps are in place. Formality attempts to prove design equivalence between the implementation design and the reference design.

```
% fm_shell
...
fm_shell (setup)> verify
```

At the end of the verification process or at any point during it if you choose to interrupt the process before completion, Formality reports on the functional equivalence of your design. The verification results are reported as PASS (all compare points are equivalent), FAIL (some compare points are not equivalent), or INCONCLUSIVE (some compare points are either unverified or aborted).

For further information about running verification and interpreting the results, see Chapter 9, "Verifying the Design and Interpreting Results."

Debugging

The debug step of the Formality process flow is required if the design verification is not successful. Debugging is that part of the process in which you use the verification results to pinpoint either failing or inconclusive results. During the debug step you might determine where and possibly why the results were unsuccessful.

The design might have failed due to a setup problem or because of a logical difference between the designs. Different causes of failure require different debugging solutions, so a number of debugging strategies are available. These range from manually matching unmatched compare points to debugging through GUI-based analysis. The same holds true for inconclusive verifications.

For further information about debugging, see Chapter 10, "Debugging Verification."

Tutorial

For a tutorial that shows the Formality process flow, see Chapter 4, "Tutorial."

Library Verification Mode

For further information about library verification, see Chapter 12, "Library Verification Mode."

3

Invocation

Formality offers two working environments: the Formality shell (a command-line-based user interface) and the Formality GUI (a graphical windows-based interface). This chapter describes how to invoke these environments and how to use interface elements, such as the command log file and the help facility.

The chapter includes the following sections:

- Introduction
- Basic Usage
- Advanced Usage

Figure 3-1 outlines the timing of invoking and starting Formality within the design verification process flow. This chapter focuses on how to invoke and start Formality.

Start Formality Load Guidance Load Reference Load Implementation Perform Setup Match **Compare Points** Run Verify

Interpret Results

Success?

Done

Yes

No

Debug

Figure 3-1 Invoking Formality in the Design Verification Process Flow Overview

Introduction

All Formality descriptions and operations assume that Formality was properly installed and licensed and that it meets computational requirements.

Prior to invoking Formality, you need to set up the user environment. You do this by specifying the location of the executable file and setting the license environment variable.

For information about the Synopsys setup file, see the

Specifying the Executable File Location

To set up a new Formality tool user, add the Formality directory containing the executable file to the PATH environment variable.

If you are using the C shell, add the following line to the .cshrc file:

```
set path=(install_dir/bin $path)
```

If you are using the Bourne, Korn, or Bash shell, add the following line to the .profile, .kshrc, or .bashrc file:

```
PATH=install_dir/bin:$PATH export PATH
```

Specifying License Environment Variable

You must install the Synopsys Common Licensing (SCL) software and define the SNPSLMD_LICENSE_FILE variable before you can verify the Formality installation. For information about downloading SCL, installing SCL, or setting the license variable, see *Installing Synopsys Tools* at http://www.synopsys.com/Support/Licensing/Installation/Pages/default.aspx.

Basic Usage

The Formality shell, fm_shell, is the command-line interface. The fm_shell commands are made up of command names, arguments, and variable assignments. Commands use the tool command language (Tcl), which is used in many applications in the EDA industry.

The Formality GUI is the graphical, menu-driven interface, using which you can verify designs. It also provides schematic and logic cone views to help you debug failed verifications.

Invoking the Formality Shell

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

```
% fm_shell
...
fm_shell (setup)>
```

The Formality copyright or license notice, program header, and fm_shell prompt appear in the window where you started Formality.

Table 3-1shows the command-line options you can use when starting fm_shell.

Table 3-1 The fm_shell Command Options

-file filename	Invokes Formality in a shell and runs a batch script.
	For example, % fm shell -file my_init_script.fms
-x command_string	Executes command_string (a string of one or more fm_shell commands separated by semicolons) before displaying the initial fm_shell prompt and before executing a -file script. If the last statement in command_string is
-no_init	quit, no prompt displays and the command shell exits. Prevents setup files from being automatically read upon invocation. This is useful when you have a command log or other script file that you want to use to reproduce a
	previous Formality session. For example,
	% fm_shell -no_init -file fm_shell_command.log.copy
-64bit -32bit	Invokes Formality using the 64-bit binary executable on platforms that support it. The default is 64 bits.
-overwrite	Overwrites existing FM_WORK, formality.log, and fm_shell_command.log files.

Table 3-1 The fm_shell Command Options (Continued)

-name_suffix filename_suffix Appends the suffix to the log files created by Formality.

For example,

% fm_shell -name_suffix tmp files

This command generates files named FM_WORK_tmp,

formality_tmp.log, and fm_shell_command_tmp.

-version Prints the version of Formality and then exits.

-gui Starts the Formality graphical user interface.

-work_path Specifies the location of FM_WORK and other temporary

directories. Using this option, you can specify a Linux path where FM_WORK and other temporary directories are created. If the Linux path you specify does not exist,

Formality creates the specified folder.

Synopsys Setup File

Each time you invoke Formality, it executes the commands in the Formality setup files, all named .synopsys_fm.setup. These setup files can reside in three directories that Formality reads in a specific order. You can use these files to set variables automatically to your preferred settings.

The following list shows the order in which Formality reads the files:

- Synopsys root directory. For example, if the release tree root is /usr/synopsys, the setup file is /usr/synopsys/admin/setup/.synopsys_fm.setup
- 2. Your home directory. The .synopsys_fm.setup file in this directory applies to all sessions that you start.
- 3. The directory where you have invoked Formality (current working directory). Customize the .synopsys fm.setup file in this directory for a specific design.

If a particular variable is set in more than one file, the last file read overwrites the previous setting.

Redirecting Standard Output

Formality writes the full transcript of the verification run to stdout. Save this transcript to a file when invoking the tool by piping the fm_shell command to the Linux command tee -i. Use the -i option with the tee command to pass Ctrl+C or other interruptions to fm_shell . For example,

```
fm_shell -file my_script.tcl | tee -i my_transcript.out
```

Invoking the Formality GUI

When you start Formality, you are provided with a transcript window containing the Formality banner. Immediately after the banner is displayed, Formality lists two key features for the current release.

To invoke the Formality GUI from the fm_shell command, with the Formality shell environment and command-line interface running, execute the following command:

```
% fm_shell (setup)> start_gui
```

Alternatively, you can start the GUI from the fm shell command by executing:

```
% fm shell -qui
```

If you use the Formality GUI, a pop-up window appears, listing all the key features for the current release. You can hide this window for future releases. To access these key features at any time, choose Help > Release Highlights.

You can choose to display or hide primary sections of the GUI session window. For example, to hide or display the toolbar or status bar, use the View menu. In the menu, select an option to display or hide the corresponding area of the session window. A check mark is shown next to the menu item if that section is currently being displayed in the window.

The lower area of the window contains the command console, Formality prompt, and status bar. Use the Log, Errors, Warnings, History, and Last Command options above the Formality prompt to display different types of information in the command console.

You can exit the GUI without exiting the Formality session by selecting File > Close GUI, or issuing the stop_gui command from the command line in the Formality GUI window.

Interrupting Formality

In fm_shell, you can interrupt Formality by pressing Ctrl+C. The response depends on what Formality is doing currently.

- If Formality is processing a script, script processing stops.
- If Formality is in the middle of a process, the following message appears:

```
Interrupt detected: Stopping current operation
```

Depending on the design, it can take Formality one or two minutes to respond to Ctrl+C.

 If Formality is waiting for a command (not in the middle of a process), the following message appears:

```
Interrupt detected: Application exits after three ^C interrupts
```

In this case, you can exit Formality and return to the Linux shell by pressing Ctrl+C two more times within 20 seconds, with no more than 10 seconds between each.

In the GUI, when you run a verification, the status bar indicates the progress. You can interrupt the process by clicking Stop. Processing might not stop immediately.

Viewing CPU Statistics

To view the CPU time used by the Formality shell, use the cputime command. The CPU time is shown in seconds.

```
fm_shell (setup)> cputime
3.73
```

To view the memory used by the Formality shell, use the memory -format -units command. The default unit is kb (kilobyte).

```
fm_shell (setup)> memory
4880
```

Note that when you use the set_host_options -max_cores command for multicore processing and specify two or more cores, the tool reports higher memory for the specific Formality session, including child processes. This is because Linux does not distinguish between the shared and private memory usage during multi core processing. The memory reported might be more than the current memory usage.

For information about multicore processing, see the "Verification Using Multicore Processing" on page 9-11.

Getting Help

Formality provides various forms of online help, such as the help and man commands.

You can use a wildcard pattern as the argument for the help command. The available wildcards are:

*	Matches any number of characters.
?	Matches exactly one character.

Use the help command to list all commands alphabetically:

```
fm_shell (setup)> help
```

The following command uses a wildcard character to display all commands that start with the word *find*:

fm_shell > help find*

find_cells	#Find the specified cells
find_nets	#Find the specified nets
find_pins	#Find the specified pins
find_ports	#Find the specified ports
find_references	#Find design references of the specified design

You can use the -help option to display syntax information for any command:

Man pages are available for every Formality shell command and application variable. For more information about a specific command or variable, use the man command followed by a command name to see the man page, as follows:

```
fm_shell (setup)> man command_name
```

You can also see the man page for a command by selecting it in the transcript window and then either clicking the man page viewer in the toolbar or choosing Man Pages from the Help menu.

To display the current value of a variable, use the printvar command followed by the variable name. For example,

```
fm_shell (setup)> printvar verification_auto_loop_break
verification_auto_loop_break = "true"
```

The following command displays a detailed description of the cputime command:

```
fm_shell (setup)> man cputime
2. Synopsys Commands
                                                              Command
Reference
                                     cputime
NAME
       cputime
              Returns the CPU time used by the tool's shell.
SYNTAX
       cputime
ARGUMENTS
       none
DESCRIPTION
       This command returns the CPU time used by the tool's shell. The
time is rounded off to the nearest one hundredth of a second.
EXAMPLES
       The following example shows the output produced by the cputime
command.
         fm_shell (setup) > cputime
         3.73
         fm_shell (setup)>
```

Advanced Usage

As you have seen above, there are essentially two approaches to invoking Formality through the command line or through the GUI. Consequently, this section on invocation details is broken into the following topics:

- Commands
- GUI Environment
- Script Files
- Messages
- Output Files

Control File Names Generated by Formality

Commands

Working through the fm_shell command line is a powerful way to use the product. You can enter and edit commands, options, and arguments; view and reuse previously-entered commands; create and manipulate aliases; and even redirect the output to another file. You can also keep track of your work in any Formality session by generating a log file.

Entering Commands

Formality considers case when it processes fm_shell commands. All command names, option names, and arguments are case-sensitive. For example, the following two commands are equivalent but specify two different containers, named r and R:

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

Each Formality command returns a result that is always a string. The result can be passed directly to another command, or it can be used in a conditional expression. For example, the following command uses an expression to derive the right side of the resulting equation:

```
fm_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, Formality displays a secondary prompt, the question mark (?). Here is an example:

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

Argument Lists

When you supply more than one argument for a given Formality command, adhere to Tcl rules. Most publications about Tcl contain extensive discussions about specifying lists of arguments with commands. This section highlights some important concepts.

- Because command arguments and results are represented as strings, lists are also represented as strings, but with a specific structure.
- Lists are typically entered by enclosing a string in braces, as shown in the following example:

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

In this example, however, the string inside the braces is equivalent to the following list:

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

Note:

Do not use commas to separate list items.

If you are attempting to perform command or variable substitution, the form with braces does not work. For example, the following command reads a single file that contains designs in the Synopsys internal .db format. The file is located in a directory defined by the DESIGNS variable.

```
fm_shell (setup)> read_db $DESIGN/my_file.db
Loading db file '/u/project/designs/my_file.db'
No target library specified, default is WORK
1
fm_shell (setup)>
```

Attempting to read two files with the following command fails because the variable is not expanded within the braces:

```
fm_shell (setup)> read_db {$DESIGNS/f1.db $DESIGNS/f2.db}
Error: Can't open file $DESIGNS/f1.db.
0
fm_shell (setup)>
```

Using the list command expands the variables.

```
fm_shell (setup)> read_db [list $DESIGNS/f1.db $DESIGNS/f2.db]
Loading db file '/u/designs/f1.db'
No target library specified, default is WORK
Loading db file '/u/designs/f2.db'
No target library specified, default is WORK
1
fm_shell (setup)>
```

You can also enclose the design list in double quotation marks to expand the variables.

```
fm_shell (setup)> read_db "$DESIGNS/f1.db $DESIGNS/f2.db"
Loading db file '/u/designs/f1.db'
No target library specified, default is WORK
Loading db file '/u/designs/f2.db'
No target library specified, default is WORK
1
fm_shell (setup)>
```

Editing From the Command Line

You can use the command-line editing capabilities in Formality to complete commands, options, variables, and files that have a unique abbreviation. This line-editing capability is useful by allowing you to use the shortcuts and options available in the Emacs or vi editor. Use the <code>list_key_bindings</code> command to display current key bindings and the current edit mode. To change the edit mode, set the <code>sh_line_editing_mode</code> variable in either the .synopsys_fm.setup file or directly in the shell. To disable this feature, you must set the <code>sh_enable_line_editing</code> variable to <code>false</code> in your .synopsys_fm.setup file. It is set to true by default.

If you type part of a command or variable and then press the Tab key, the editor completes the words or file for you. A space is added to the end if one does not already exist to speed typing and provide a visual indicator of successful completion. Completed text pushes the rest of the line to the right. If there are multiple matches, all matching commands and variables are automatically listed. If no match is found (for example, if the partial command name you have typed is not unique), the terminal bell rings.

History

The history command with a numeric argument (n) lists the last n commands that you entered. By default, the history command without an argument lists the most recent 20 commands.

The following syntax is used for the history command:

The options and variables used for the history command are explained as follows:

keep number_of_lines	Changes the length of the history buffer to the number of lines you specify.
info number_of_entries	Limits the number of lines displayed to the specified number.
-h	Shows the list of commands without loading numbers.
-r	Shows the history of commands in reverse order.

For example, use the following command to review the 20 most recent commands entered:

```
4 help set
5 history -help
6 alias warnings_only "set message_level_mode warning"
7 warnings_only
8 ls -al
9 unalias warning_only
10 unalias warnings_only
11 history
fm_shell (setup)>
```

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:

```
fm_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

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

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

```
fm_shell (setup)> redirect my_script { history -h }
```

Recalling Commands

Use these Linux-style shortcuts to recall and execute previously entered commands:

11	Recalls the last command.
!-n	Recalls the n th command from the last.
!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 Formality shell displays the mode that you are currently in when using a particular command. The common modes that are available are guide, setup, match, and verify. The following example recalls and runs the most recent verification command:

```
fm_shell (verify)> !ver
verify ref:/WORK/CORE impl:/WORK/CORE
```

```
fm_shell (verify)>
```

This example recalls and starts the most recently run command:

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

Aliasing

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

```
fm_shell (setup)> alias err_only "set message_level_mode error"
```

After creating the alias, you can use it by entering err_only at the fm_shell prompt.

The following points apply to alias behavior and use:

- Aliases are recognized only when they are the first word of a command.
- Alias definitions take effect immediately and last only while the Formality session is active.
- Formality reads the .synopsys_fm.setup file when you invoke it; therefore, define commonly used aliases in the setup file.
- You cannot use an existing command name as an alias name. However, aliases can specify other aliases.
- You can supply arguments when defining an alias by surrounding the entire definition for the alias in quotation marks.

Using the alias Command

alias [name [definition]]

Use the following syntax for the alias command:

```
name Represents the name (short form) of the alias you are creating (if a definition is supplied) or listing (if no definition is supplied).
```

(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 specified, definition
	overwrites the existing definition. If no <i>definition</i> is specified, the definition of the named alias is displayed.

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

Using the unalias Command

The unalias command removes alias definitions. The following syntax for the unalias command applies:

```
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 set_identity_check alias:

```
fm_shell (setup)> unalias set_identity_check
```

Redirecting

You can cause Formality to redirect the output of a command or a script to a specified file by using the Tcl redirect command or using the > and >> operators.

Use the redirect command in the following form to redirect output to a file:

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

Use a command in the following form to redirect output to a file by using the > operator:

```
fm_shell(setup)> command > file
```

If the file does not exist, Formality creates it. If the file does exist, Formality overwrites it with new output.

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

```
fm_shell (setup)> command >> file
```

If the file does not exist, Formality creates it. If the file does exist, Formality adds the output to the end of the file.

Unlike Linux, Formality treats the > and >> operators as arguments to a command. Consequently, you must use spaces to separate the operator from the command and from the target file. In the following example, the first line is incorrect:

```
fm_shell (setup)> echo $my_variable>>file.out
fm_shell (setup)> echo $my_variable >> file.out
```

Note:

The Tcl built-in command puts does not redirect output. Formality provides a similar command, echo, that enables output redirection.

Command Log Files

The Formality command log file is called fm_shell_command n.log (where n is an integer indicating more than one invocation of Formality from the same directory). This command log file records the fm_shell commands in a Formality session, including setup file commands and variable assignments.

You can use the command log file in the following situations:

- After a Formality session to keep a record of the design analysis
- By sourcing it as a script to duplicate a Formality session

If you have problems using Formality, save this command log file for reference when you contact Synopsys. Move the command log file to another file name to prevent it from being overwritten by the next fm_shell session.

GUI Environment

This section includes the following topics that relate to using the Formality GUI:

- Windows
- Prompt
- Copying Text
- Saving the Transcript

Windows

The Formality GUI uses multiple windows to display different types of information, such as schematics and logic cones. These windows are opened by certain menu commands in the GUI.

The Window menu lists the GUI windows that are present and lets you manage those windows. Selecting any window in the list activates that window (restores the window from icon form, if necessary, and moves it to the front).

Prompt

You can use the Formality prompt to run fm_shell commands without closing the GUI.

To run the fm_shell command from within the GUI, follow these steps:

- 1. Enter a command in the text area at the Formality prompt. You can use any of these methods:
 - Type the command directly.
 - Click History, and then copy and paste commands into the text box.
- 2. Press the Enter key to execute the command.

After you perform these steps, Formality runs the command and adds it to the command history list. The transcript area displays the command results.

You can use multiple lines at the prompt by pressing Shift-Enter to move to the next line. Specify a "\" at the end of each line to indicate that the text continues on the next line.

Press the Shift-Up Arrow or Shift-Down Arrow key to cycle through the command history.

Copying Text

You can copy text to another application window by following these steps:

- 1. To display the transcript, click Log.
- 2. Select the text in the transcript area you want to copy.
- 3. Right-click and choose Copy.
- 4. Move the pointer to a shell window outside the Formality tool, or to another open application, and execute the Paste command.

In addition, you can use the Linux-style method of selecting with the left-mouse button and pasting with the middle-mouse button to transfer text into a shell window.

You can copy text from an application window to the Formality prompt by following these steps:

- 1. Select the text you want to copy.
- 2. Use the Copy command to place the highlighted text on the clipboard.
- 3. Locate the pointer in the command bar where you want the text to appear, and execute the Paste command.

In addition, you can use the Linux-style method of selecting with the left-mouse button and pasting with the middle-mouse button to transfer text from a shell window to the prompt line.

Saving the Transcript

To save the transcript area, follow these steps:

- 1. Choose File > Save Transcript to open the Save Transcript File dialog box.
- 2. Enter a file name or use the browser to select the file in which to save the transcript text.
- 3. Click Save.

Script Files

You can use the <code>source</code> command to run scripts in Formality. A script file, also called a command script, is a sequence of fm_shell commands in a text file. The syntax of the <code>source</code> command is:

fm_shell (setup)> source [-echo] [-verbose] script_file_name

-echo	Displays each command in the script as it is run.	
-verbose	Displays the result of each command in the script.	
script_file_name	Represents the name of the script file to be run.	

Table 3-2 lists some of the tasks you can perform with script files.

Table 3-2 Script File Actions

Task	Description	Example
Add comments	Add block comments by beginning comment lines with the pound sign (#).	# # Set the new string #
	Add inline comments by using a semicolon to end a command, and then using a pound sign to begin the comment.	set newstr "New"; # comment
Continue processing after an error	If an error occurs during the script execution, by default Formality discontinues processing the script. To force Formality to continue processing in this situation, set the sh_continue_on_error variable to true. (The results might be invalid if an error has occurred.)	<pre>set_app_var sh_continue_on_error true</pre>

Table 3-2 Script File Actions (Continued)

Task	Description	Example
Find scripts using the search_path variable	Set the sh_source_uses_search_path variable to true.	<pre>set_app_var sh_source_uses_search_path true</pre>

Messages

In fm_shell, you can interrupt Formality by pressing Ctrl+C. The response depends on what Formality is doing currently.

- If Formality is processing a script, script processing stops.
- If Formality is in the middle of a process, the following message appears:

```
Interrupt detected: Stopping current operation
```

Depending on the design, it can take Formality one or two minutes to respond to Ctrl+C.

 If Formality is waiting for a command (not in the middle of a process), the following message appears:

```
Interrupt detected: Application exits after three ^C interrupts
```

In this case, you can exit Formality and return to the Linux shell by pressing Ctrl+C two more times within 20 seconds, with no more than 10 seconds between each press.

In the GUI, when you run a verification, a progress bar appears in the status bar. You can interrupt the process by clicking Stop. Processing might not stop immediately.

Controlling Message Types

Formality issues messages in certain formats and during certain situations. You can control the types of messages Formality displays.

Formality generates messages in one of two formats:

```
severity: message (code)
severity: message

severity: message (code)

Represents the level of severity (note, warning, or error) as described in Table 3-3.

severity: message

The text of the message.
```

code	Helps identify the source of the message. The code is separated into a prefix and a number. The prefix is two, three, or four letters, such as INT-2. For information about a particular message code, use the man command (for example, man INT-2).	
	Formality has three specific message prefixes, FM-, FMR-, and FML The prefix indicates the type of Formality function involved: a general Formality function, the Verilog RTL reader, or the Verilog library reader, respectively.	

In the following example, Formality displays an error-level message as a result of an incorrectly entered read_db command:

```
fm_shell (setup)> read_db -myfile
Error: unknown option '-myfile' (CMD-010)
Error: Required argument 'file_names' was not found (CMD-007)
fm_shell (setup)>
```

Table 3-3 describes the different error message levels.

Table 3-3 Message Severities

Severity	Description	Example
Note	Notifies you of an item of general interest. No action is necessary.	^C Interrupt detected: Stopping current operation.
Warning	Appears when Formality encounters an unexpected, but not necessarily serious, condition.	Warning: License for "DW-IP-Consultant" has expired. (SEC-6)
Error	Appears when Formality encounters an unexpected condition that is more serious than a warning. Commands in progress are not completed when an error is detected. An error can cause a script to terminate.	Error: Required argument "file_names" was not found (CMD-007).

Each message is identified by a code, such as CMD-010. To obtain more information about a message, see the man page for the code. For example, if Formality reports "Error: Can't open file xxxx (FM-016)," you can obtain more information by entering man FM-016.

Set Thresholds

You can establish a message-reporting threshold that remains effective during the Formality session. This threshold can cause Formality to display error messages only, warnings and error messages only, or notes, warnings, and error messages.

By default, Formality issues the three levels of messages described in Table 3-3. A fourth message type, fatal error, occurs when Formality encounters a situation that causes the tool to exit. Regardless of the threshold setting, Formality always issues a fatal error message before it exits the tool and returns control to the shell.

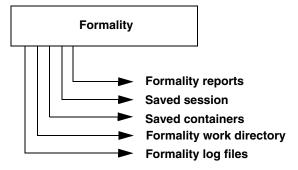
To set the message threshold, use the Formality shell or the GUI as shown:

fm_shell	G	UI
Specify:	1.	Choose Edit > Formality Tcl Variables. The Formality Tcl Variables dialog box appears.
<pre>set_app_var message_level_mode threshold Specify error, warning, info, or none for threshold.</pre>	2.	From Setup, select the message_level_mode variable.
	3.	In the Choose a value text box, select error, warning, info, or none.
	4.	Choose File > Close.

Output Files

Formality generates several types of output files, as illustrated in Figure 3-2.

Figure 3-2 Generated Output



The output files generated by Formality are described as follows:

Generated Output	Description
Formality reports	ASCII files you produce by redirecting output from the Formality reporting feature. These reports contain information about all aspects of the verification and diagnosis.
Saved session	A file that contains the state of the verification session. You create this file by saving the Formality session.

Generated Output	Description
Saved containers	The Formality internal representation of a container. You create these files by saving individual containers.
	For information about saving containers, see "Setting Up and Managing Containers" on page 6-18.
Formality work directory	The Formality work directory named FM_WORK. Formality creates this directory upon invocation. It contains containers and shared technology libraries.
Formality log files	The Formality tool maintains the following log files: formality.log, fm_shell_command.log, and formality_svf.log.
	The formality.log file contains verbose information not printed to the transcript. For example, during verification, the transcript might print an information message indicating that constants were propagated in the reference design and directing you to the formality.log file for more information.
	The fm_shell_command.log file contains a history of Formality shell commands that have been run during the session.
	The formality_svf directory contains all guidance information from any SVF files specified with the set_svf command.
	If multiple sessions of Formality are running, the working directory and log files are named using the following scheme, where n is an integer value:
	FM_WORKn
	formality n.log
	fm_shell_commandn.log
	formalityn.svf
N	

Note:

Exiting abnormally from Formality can clutter your file system with locked files associated with Formality logs and with the Formality working directory. You can safely delete these files when the Formality session associated with them is no longer running.

Control File Names Generated by Formality

The output file names can be appended with a specified suffix for each invocation of the tool. These names can be appended with a unique suffix for each verification run.

Specifying a unique name can be useful for correlating the Formality transcript with the Formality log file when you run multiple verifications within the same directory.

Use the $fm_shell -name_suffix$ command to specify unique file names. Formality constructs the file names and directories as follows:

- formality_suffix.log
- fm_shell_command_suffix.log
- formality_suffix_svf
- FM WORK suffix

You can also use the <code>-overwrite</code> option to overwrite existing files. If you use the <code>-name_suffix</code> option and a file with the same suffix already exists, Formality generates an error message. If you want to overwrite any existing files, use the <code>-overwrite</code> option with the <code>fm shell</code> command.

You can access (read-only) the following two tool command language (Tcl) variables to see the new file names for the formality.log file and the fm_shell_command.log file:

- formality_log_name
- sh_command_log_file

4

Tutorial

This tutorial explains first how to prepare for running Formality and then works through three examples of using the tool.

This chapter includes the following sections:

- Before You Start
- Verifying fifo_with_scan.v Against fifo_mod.vg
- Verifying fifo_jtag.v Against fifo_with_scan.v
- Verifying fifo.vg Against fifo.v
- For More Information

Before You Start

Before you begin this tutorial, ensure that Formality is properly installed on your system. Your .cshrc file should set the path to include the bin directory of the Formality installation. For example, if your installation directory is /u/admin/formality and your platform type is sparcOS5, specify the set path statement, where

/u/admin/formality represents the Formality installation location on your system:

```
set path = ($path /u/admin/formality/bin)
```

You do not need a separate executable path for each platform. The Formality invocation script automatically determines which platform you are using and calls the correct binary. To enable the tool to do this, however, you must make sure all platforms needed are installed in one Formality tree. Install Formality in its own directory tree, separate from other Synopsys tools such as Design Compiler.

Creating Tutorial Directories

After installing Formality, the files needed for the design examples are located in the fm_install_path/doc/fm/tutorial directory. You must copy the necessary files to your home directory.

To create a tutorial directory with all of its subdirectories, do the following:

- 1. Change to your home directory.
 - % cd \$HOME
- 2. Use the following command to copy the tutorial data, where fm_install_path is the location of the Formality software:
 - % cp -r fm_install_path/doc/fm/tutorial \$HOME
- 3. Change to the new tutorial directory.
 - % cd tutorial

Tutorial Directory Contents

The tutorial directory contains the following subdirectories:

- · GATE: Verilog gate-level netlist.
- GATE_WITH_JTAG: Verilog gate-level netlist with scan and Joint Test Action Group (JTAG) insertions.
- GATE WITH SCAN: Verilog gate-level netlist with scan insertion.
- LIB: Technology library required for the gate-level netlists.
- RTL: RTL source code.

Invoking the Formality Shell

To start Formality, enter the following command at the operating system prompt:

```
% fm_shell
...
fm_shell (setup)>
```

The fm_shell command starts the Formality shell environment and command-line interface. From here, start the GUI as follows:

```
fm_shell (setup)> start_gui
```

The word (setup) indicates the mode that you are currently in when using commands. The modes that are available are guide, setup, match, and verify. When you invoke Formality, you begin in the setup mode.

For more information about fm_shell and GUI environments, see Chapter 3, "Invocation."

Verifying fifo.vg Against fifo.v

In this portion of the tutorial you verify a synthesized design named fifo.vg, which is a pure Verilog gate-level netlist, against an RTL reference design named fifo.v.

At any time, you can exit and save the current Formality session by executing the following command:

```
fm_shell> save_session session_file_name
```

To invoke that session again, execute

fm_shell> restore_session session_file_name

Loading the SVF File

Before specifying the reference and implementation designs, you can optionally load an automated setup file (.svf) into Formality. The SVF file helps Formality process design changes caused by other tools used in the design flow. Formality uses this file to assist the compare point matching and verification process. This information facilitates alignment of compare points in the designs that you are verifying. For each SVF file that you load, Formality processes the content and stores the information for use during the name-based compare point matching period.

To load the SVF file, do the following:

```
fm_shell> set_svf svf_file_name.svf
```

Note:

If you want to pass additional constraint and nonconstraint information from Design Compiler to Formality, set the automated setup mode before reading the SVF file.

Note:

This tutorial does not use an SVF file, so this information is given here for reference only.

Specifying the Reference Design

Specifying the reference design involves the reading in of design files, optionally reading in technology libraries, and setting the top-level design.

The reference design is the design against which you compare the transformed (implementation) design. The reference design in this case is the RTL source file named fifo.v.

It is necessary to specify that the DesignWare root directory for fifo.v contains a DesignWare instantiated RAM block. As needed, enter <code>getenv SYNOPSYS</code> at the Formality prompt to obtain the path name of the root directory.

Set the search path to the RTL and LIBS directories as follows,

```
fm_shell> set_app_var hdlin_dwroot path_to_DesignCompiler_install
```

Now load in all the reference Verilog files,

```
fm_shell> read_verilog -r { fifo.v gray2bin.v gray_counter.v
pop_ctrl.v push_ctrl.v rs_flop }
```

Note the reference does not need any specific technology file to which to map, so the top-level design for the reference can now be defined.

Setting the top-level design starts the linking and elaboration process on all files and reports if there are any missing files. Formality searches for the DesignWare RAM automatically:

```
fm_shell> set_top fifo
```

Specifying the Implementation Design

The procedure for specifying the implementation design is identical to that for specifying the reference design. In this case though, there is no need for a technology library to which to map.

```
fm_shell> read_db -i lsi_10k.db
```

Use a Verilog gate-level design for the GATE directory to compare to the reference.

```
fm_shell> read_verilog -i GATE/fifo.vg
```

To define the top level of the implementation use this command:

```
fm_shell> set_top fifo
```

Setting Up the Design

You often need to specify additional setup information to account for designer knowledge not contained in the design netlist or to achieve optimal performance.

This step involves supplying information to Formality. For example, you might need to set constants if the design underwent transformations such as scan or JTAG insertion. In this case, only fifo.vg was synthesized; therefore, you can move on to the next step, Match.

For more information about setup possibilities, see Chapter 7, "Performing Setup."

Matching Compare Points

Match compare points is the process by which Formality segments the reference and implementation designs into logical units, called logic cones. Each logic cone feeds a compare point, and each compare point in the implementation design must match each compare point in the reference design or else verification fails. Matching ensures that there are no mismatched logic cones and verifies the implementation design for functionality.

For conceptual information about compare points, see "Concept of Compare Points" on page 1-8. For more information about how Formality matches compare points, see Chapter 8, "Matching Compare Points."

To match compare points between fifo.v and fifo.vg, do the following:

fm_shell> match

Verifying the Designs

To verify the designs:

fm_shell> verify

In this case, verification fails. This test case includes a deliberate design error to introduce you to the debug capabilities of Formality.

Debugging

The challenge for most users is debugging failed verifications. That is, you must find the exact points in the designs that exhibit the difference in functionality and then fix them.

Using the GUI for debugging is much more intuitive. The following command invokes the GUI:

```
fm_shell> start_gui
```

Before proceeding with debugging in this tutorial, the next section briefly goes over some aspects of the GUI.

Graphical User Interface

This section explains how to use the graphical user interface in Formality in the following subsections:

- Main GUI Session Window
- Debugging Using the GUI
- Verifying fifo_with_scan.v Against fifo_mod.vg
- Verifying fifo_jtag.v Against fifo_with_scan.v
- Debugging Using Diagnosis
- For More Information

Main GUI Session Window

The main GUI session window contains the following window areas, as shown in Figure 4-1.

Figure 4-1 GUI Session Window

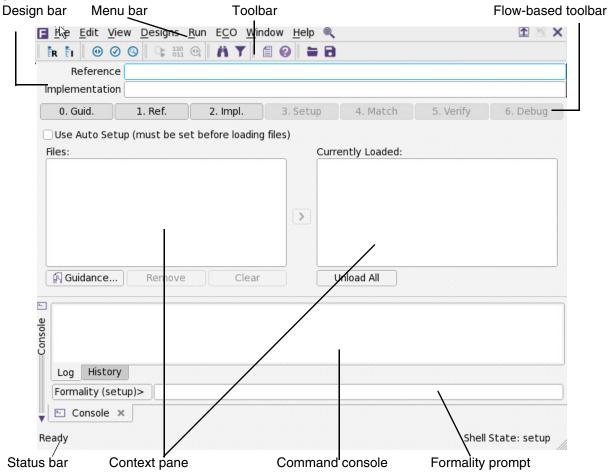


Table 4-1 Window Areas

Window area	Description
Design bar	Displays the path for the reference and implementation WORK libraries.
Menu bar	GUI commands, some of which are duplicated in the toolbar and right-click options.
Toolbar	GUI commands. The toolbar changes depending on the view displayed in the context pane. You can rearrange the icons on the toolbar and move the toolbar to any edge of the window. Right-click on the toolbar and select or deselect to view the desired toolbar menu.

Table 4-1 Window Areas (Continued)

Window area	Description
Flow-based toolbar	Options that indicate the correct flow to employ to perform formal verification. The options are highlighted to indicate where you are in the flow. Each option displays a new view in the context pane. By default, the GUI opens at the first step, Guidance, with the guidance work area displayed in the context pane.
	When you use fm_shell to perform steps and invoke the GUI, the GUI opens with the option highlighted to indicate where you are in the flow. This also occurs when you continue a previously saved Formality session.
Context pane	The main working area. From here, you perform the actions necessary to perform verification. You can also view reports here. You can resize and detach the context or report pane to view larger reports.
Command console	Displays transcripts and other information, depending on the command used at the Formality prompt. You can resize and detach the command console to view large command transcripts.
Formality prompt	The text box where you can enter Formality commands and variables that are not available through the GUI interface.
Status bar	Current state of the tool.

Debugging Using the GUI

To debug the implementation design, fifo.vg,

- Start the Formality tool and open the Formality GUI.
 For information on how to start Formality, see Invoking the Formality Shell.
- 2. On the flow-based toolbar, click the Debug tab if it is not already selected.

The context pane displays the Failing Points report. Groups of failing points with similar names might appear, except for the last elements. For example, you might see *_[reg0], *_[reg1], *_[reg2], and *_[reg3]. Typically, a group of failing points is caused by a single error.

3. To run diagnosis on the failing points, click Analyze.

During diagnosis, Formality analyzes a set of compare points and finds the error candidates. Click on the error candidates. The Error Candidates window appears displaying the error candidates found in your design.

Note:

While debugging, if you get an error stating there was a diagnosis failure due to too many errors (and you know the error is not caused by setup problems), select a group

of failing points with similar names and right-click and choose Diagnose Selected Points. This might help to direct diagnosis to a specific section of the design.

4. From the Error Candidates Window, right-click the error U81 and select Show Logic Cones.

The window displays a list of related failing points for that error, from which you can select one of those points (for this example, use push_logic/pop_count_svd_reg[0]) and double-click it to view the logic cone.

The Cone Schematics window appears, displaying reference (top screen) and implementation (bottom screen) schematics for the logic cone. It highlights and zooms to the error candidate inverter, U81, in the implementation cone. The reference schematic highlights the matching region corresponding to the error candidate in the implementation design.

The error candidate is highlighted in orange. The corresponding matching region in the reference design is highlighted in yellow. To view the error region in isolation,

- Right-click and choose Prune/Restore > Isolate Error Candidates
 Or
- Choose Edit > Prune/Restore > Isolate Error Candidates
 This prunes away all the logic and shows the error inverter.

You can view the cone inputs that have been pruned away in the Pattern window.

Colors in the schematics window have different meanings depending on the color mode selected. The color modes are none (the default), constants, simulation values, and error candidates.

- None: The default color mode.
- Constants: Nets with a constant logic value 0 are blue, nets with logic 1 are orange, and the remaining nets are gray. The remaining objects are colored in the default color mode.
- Simulation values: Nets with simulation logic 0 are blue, nets with simulation logic 1 are orange, and the remaining objects are colored in the default color mode.
- Error candidates: Error drivers corresponding to the error candidates are highlighted in orange. The corresponding matching region is highlighted in yellow.
- 5. Observe the patterns annotated on the CLK net. The reference design shows logic 0, while the implementation design shows logic 1.

To find the cause for this functional difference,

- Select the net in the implementation design.
- o Right-click and choose Prune/Restore > Isolate Subcone.

- Select the net in the reference design.
- Right-click and choose Prune/Restore > Isolate Subcone.

The screens change to display only the net with errors. Note that the logic driving the implementation CLK pin includes an inverter. During synthesis, an inverter might be inserted to fix hold time problems.

You can

- Zoom in by clicking the Zoom In icon on the toolbar or by clicking on the schematic.
 Deselect the option to return to the pointer.
- Copy selected objects in the design and cone schematics. From the context pane, you can highlight the object, select Edit > Copy, and choose one of the following menus: Instance Name, Library Name, or Design Name.
- Paste these names into the Formality prompt or any other editable text box by pressing Ctrl+V or by right-clicking and choosing Paste.
- 6. Fix the error by editing the netlist or resynthesizing the design to generate a new netlist free of errors in clock tree manipulations.

The fifo_mod.vg file in the GATE directory contains the corrected netlist. Execute the following command at the Formality prompt to view the difference:

```
% diff fifo.v fifo.mod.vg
```

You can see that the modified netlist removes the inverter.

- 7. After closing the Cone Schematics window, verify the corrected implementation design, fifo_mod.vg, against the reference design. Specify fifo_mod.vg again as the new implementation design as follows:
 - Click the Implementation tab.
 - By default, the Read Design Files and Verilog tabs are active.
 - Click the Verilog tab.
 - Click Yes to remove the current implementation design data.

Note:

Clicking Yes permanently removes the current implementation design data. In practice, you must save the data before specifying a new implementation (or reference) design.

- Navigate to the GATE subdirectory and select the fifo_mod.vg design file.
- Click Open and click Load Files
 - Skip the Read DB Libraries tab because the technology library is shared.

Click the Set Top Design tab and make sure that WORK and fifo are selected.

- o Click Set Top.
- 8. Skip the Setup step. In this tutorial, you can also skip the Match step because you did not change the setup that alters compare points, and you did not appreciably change the implementation design by removing the inverter. In addition, you know that all the compare points matched previously.
- 9. From the Verify tab, click Verify.

Formality performs automatic compare point matching before verification when you do not perform the Match step beforehand. Verification is successful.

Now that you have completed this section of the tutorial, prepare the GUI as follows for the next section:

- 1. From the Designs menu, choose Remove Reference and click Yes.
- 2. From the Designs menu, choose Remove Implementation and click Yes.

Note:

Clicking Yes permanently removes the current reference and implementation data. Always make sure to save (as required) before removing any design data.

3. At the Formality prompt, enter the following command:

```
remove_library -all
```

The transcript says "Removed shared technology library 'LSI 10K".

You now have the equivalent of a fresh session with which to execute the next section of the tutorial.

Verifying fifo_with_scan.v Against fifo_mod.vg

Note:

At any time, you can exit and save the current Formality session by choosing File > Save Session. To invoke that session again, choose File > Restore Session.

In this tutorial, the load reference and implementation steps are done using the GUI. Though not typical, it is done this way here to show the GUI steps since it was done from the shell in the first tutorial section. Doing it from the fm_shell is left as an exercise for the user.

In the following tutorial, specify the successfully verified netlist, fifo_mod.vg, as the reference design and the fifo_with_scan.v design that went through a design transformation as the implementation design. The fifo_with_scan.v design includes a scan logic.

To perform the verification steps (reference, implementation, setup, match, verify, and debug) in one continuous flow,

- On the flow-based toolbar, click the Reference tab.
 By default, the Read Design File tab and Verilog tab are active.
- 2. Click Verilog.

The Add Verilog Files dialog box appears.

- 3. Navigate to the GATE directory and select the fifo_mod.vg design file.
- 4. Click Open and click Load Files .
- 5. Click the Read DB Libraries tab and select Read as a shared library.

Because this is a gate-to-gate verification, the logic library must be available for both the fifo_mod.vg and fifo_with_scan.v designs. By default, DB logic libraries are shared.

If you use a Verilog or VHDL technology library, you must specify the <code>read_verilog-technology_library</code> or <code>read_vhdl-technology_library</code> command at the Formality prompt, because they are not shared libraries.

6. Click DB.

The Add DB Files dialog box appears.

- 7. Navigate to LIB directory and select the lsi_10k.db logic library file.
- 8. Click Open and click Load Files 🕥 .
- 9. Click the Set Top Design tab and select the fifo design in the WORK library to set it as the top-level design.
- 10. Click Set Top.

Next, you specify the implementation design, a procedure similar to the one described in "Specifying the Implementation Design" on page 4-5.

11. Click the Implementation tab.

By default, the Read Design Files and Verilog tabs are selected.

12. Click Verilog.

The Add Verilog Files dialog box appears.

13. Navigate to the GATE_WITH_SCAN directory and select the fifo_with_scan.v design file.

- 14. Click Open and click Load Files .
- 15. Click Set Top Design tab and select the fifo design in the WORK library to set it as the top-level design.
- 16. Click Set Top.

Skipped the Read DB Libraries step because you had already specified lsi_10k.db as a shared technology library.

17. Click the Setup tab.

Unlike the verification you performed between fifo.vg and fifo.v, where you skipped the setup phase, the implementation design you just specified must have its inserted scan disabled before verification.

18. Click the Constants tab and click Set.

The Set Constant dialog box appears. It lists all the input, output, and bidirectional ports within the fifo with scan.v design file.

- 19. Click the Implementation tab and select fifo, and see ports appears in the drop-down box near the top of the display area.
- 20. Deselect the Inputs box under Hide Objects > Ports.
- 21. Scroll or search for the port named test_se and select it.

You can also use the Search text box to locate the signal that you want to change.

22. In the Constant Value area at the bottom of the dialog box, select 0 and click OK.

Setting the test signal, test_se, to a constant zero state disables the scan logic in the fifo with scan.v design file. Note that test se now appears in the Command console.

23. Click the Match tab and click Run Matching.

Matching yields one unmatched compare point that you need to analyze and fix if necessary.

24. Click OK to remove the Information dialog box and click the Unmatched Points tab.

You see a report on the unmatched points, test_se, test_si1, and test_si2. These are extra compare points in the implementation design, related to the inserted scan that you previously disabled. In this case, extra compare points are expected in the implementation design. Therefore, you can ignore them and continue to the verification process.

Note:

Extra compare points in the reference design are not expected. Therefore, you must debug them as outlined in "Debugging" on page 4-7.

25. Click the Verify tab and click Verify.

The verification is successful. The scan insertion did not alter the implementation design features. However, if you had not disabled the test signal test_se in step 19, verification would have failed.

Now that you have completed this section of the tutorial, prepare the GUI as follows for the next section:

- 1. From the Designs menu, select Remove Reference and click Yes.
- 2. From the Designs menu, select Remove Implementation and click Yes.

Note:

Clicking Yes permanently removes the current reference and implementation data. Always make sure to save (as required) before removing any design data.

3. At the Formality prompt, enter the following command:

```
remove_library -all
```

The transcript says "Removed shared technology library 'LSI_10K'."

This is now the equivalent of a fresh session with which to execute the next section of the tutorial.

Verifying fifo_jtag.v Against fifo_with_scan.v

In this tutorial, specify the successfully verified scan-inserted netlist, fifo_with_scan.v, as the reference design and the fifo_jtag.v design that went through a design transformation as the implementation design. The fifo_jtag.v includes a JTAG insertion and a scan insertion.

To perform the following verification steps (reference, implementation, setup, match, verify, and debug) in one continuous flow,

- 1. On the flow-based toolbar, click the Reference tab.
 - By default, the Read Design File tab and Verilog tab are active.
- 2. Click Verilog.

The Add Verilog Files dialog box appears.

- 3. Navigate to the GATE_WITH_SCAN and select the fifo_with_scan.v file design file.
- 4. Click Open and click Load Files .
- 5. Click the Read DB Libraries tab and select Read as a shared library.

Because this is a gate-to-gate verification, the logic library must be available for both the fifo_with_scan.v and fifo_jtag.v designs.

6. Click DB

The ADD DB Files dialog box appears.

- 7. Navigate to the LIB directory and select the lsi 10k.db logic library file.
- 8. Click Open and click Load Files .
- 9. Click the Set Top Design tab and select the fifo design in the WORK library to set it as the top-level design.
- 10. Click Set Top.
- 11. Click the Implementation tab.

By default, the Read Design Files tab and Verilog tab are active.

12. Click Verilog.

The Add Verilog Files dialog box appears.

- 13. Navigate to the GATE_WITH_JTAG directory and select the fifo_jtag.v design file.
- 14. Click Open and click Load Files .
- 15. Click the Set Top Design tab and select the fifo design in the WORK library to set it as the top-level design.
- 16. Click Set Top.

You might have to scroll down to find the fifo design, because the inserted JTAG modules are listed at the top of the choose-a-design pane.

Note

If you set accidentally a wrong design as the top-level design, redefine the implementation (or reference) design by first removing the reference and implementation designs and starting again.

Skip the Read DB Libraries step because you have already specified lsi_10k.db as a shared logic library in Step 7.

17. Click the Setup tab.

For this verification, you must disable the scan in fifo_with_scan.v in the reference design and disable JTAG signals in the implementation design.

18. Click the Constants tab and click Set.

The Set Constant dialog box appears.

19. Click the Reference tab, select fifo, and see ports displayed in the drop-down box near the top of the display area.

- 20. Deselect the Inputs box under Hide Objects > Ports.
- 21. Scroll or search for the test_se port and select it.

You can also use the Search text box to locate the signal that you want to change.

- 22. In the Constant Value area at the bottom of the dialog box, select 0 and click Apply.
- 23. Click the Implementation tab, select fifo, and see ports displayed in the drop down box near the top of the display area.
- 24. Repeat Step 20-Step 22 to disable the test_se test signal for the implementation design. Similarly, disable the jtag_trst and jtag_tms JTAG signals by setting them to constant 0 and click Apply.
- 25. Close the Set Constant dialog box.

The Constants report lists the four disabled signals, one for the reference design and three for the implementation design.

26. Click the Match tab and click Run Matching.

Matching yields 171 unmatched compare points that you must analyze and fix, if necessary.

27. Click OK to close the Information dialog box and the Unmatched Points tab.

You see that the extra compare points are located in the implementation design and it is related to the inserted JTAG that you previously disabled. Specifically, JTAG insertion results in the addition of a large logic block called a tap controller. Therefore, extra compare points are expected in the implementation design. You can ignore them and move to verification.

28. Click the Verify tab and click Verify.

The verification is successful. The JTAG insertion did not alter the implementation design features.

Debugging Using Diagnosis

In some designs, you can reach a point where you have fixed all setup problems in your design or determined that no setup problems exist. Therefore, the failure must have occurred because Formality found functional differences between the implementation and reference designs.

Use the following steps to isolate the problem (assuming that you are working in the GUI).

- 1. On the flow-based toolbar, click the Debug tab.
- 2. Click the Failing Points tab to view the failing points.

During verification, Formality creates a set of failing patterns for each failing point. These patterns show the differences between the implementation and reference designs. Diagnosis is the process of analyzing these patterns and identifying error candidates that might be responsible for the failure. Sometimes the design can have multiple errors and, therefore, an error candidate can have multiple locations.

3. Select all failing points and click Analyze to run a diagnosis on all of the failing points listed in this window.

Note:

After clicking Analyze, you might get a warning (FM-417) stating that too many error locations caused the diagnosis to fail (if the error locations exceed five). If this occurs and you have already verified that no setup problems exist, select a group of failing points (such as a group of buses with common names), and right-click and choose Diagnose Selected Points. This can help the diagnosis by paring down the failing points to a specific section in the design. Finally, if the group diagnosis fails, select a single failing point and run the selected diagnosis.

When the diagnosis is complete, the Error Candidate window appears.

4. Click the Error Candidates tab to view the error candidates.

You see a list of error candidates in this window. An error candidate can have multiple distinct errors associated with it. For each of the errors, the number of related failing points is reported.

There can be alternate error candidates apart from those that are shown in this window. You can inspect the alternate candidates by using Next and Previous. You can reissue the error candidate report anytime after running the diagnosis by using the report error candidates command.

5. Select an error with the maximum number of failing points. Right-click that error and choose View Logic Cones.

If there are multiple failing points, a list appears, from which you can choose a particular failing point to view. Errors are the drivers to the design whose function can be changed to fix the failing compare point.

The schematic shows the error highlighted in the implementation design along with the associated matching region of the reference design.

Note:

Changing the function of an error location can sometimes cause previously passing input patterns to fail.

Examine the logic cone for the driver causing the failure. The problem driver is highlighted in orange. To view the error region in isolation, click Isolate Error Candidates Pruning Mode. You can also prune the associated matching region of the reference design. You can undo this pruning mode by choosing the Undo option from the Edit menu.

Note:

You can employ the previous diagnosis method by setting the diagnosis_enable_error_isolation variable to false and then rerunning the verification.

For More Information

For more information about each stage of the formal verification process demonstrated in the tutorial, see the following chapters:

- Chapter 3, "Invocation," which describes the user interfaces and describes how to invoke the tool.
- Chapter 6, "Loading Designs," which describes how to read in designs and libraries, and how to define the reference and implementation designs.
- Chapter 7, "Performing Setup," which describes how to set design-specific parameters to help Formality perform verification and to optimize your design for verification.
- Chapter 8, "Matching Compare Points," which describes how to match compare points.
- Chapter 9, "Verifying the Design and Interpreting Results," which describes how to perform verification.
- Chapter 10, "Debugging Verification," which describes diagnostic procedures that can help you locate areas in the design that caused failure.
- Chapter 12, "Library Verification Mode," which describes how to compare two technology libraries.
- Appendix B, "Tcl Syntax as Applied to Formality Shell Commands," which describes Tcl syntax as it relates to more advanced tasks run from fm_shell. Topics include application commands, built-in commands, procedures, control flow commands, and variables.

5

Load Guidance

Guidance is the process by which an implementation tool, such as Design Compiler, provides setup information for formal verification. This chapter describes how to setup designs for verification.

The chapter includes the following sections:

- · What is Guidance?
- Basic Usage
- Advanced Usage

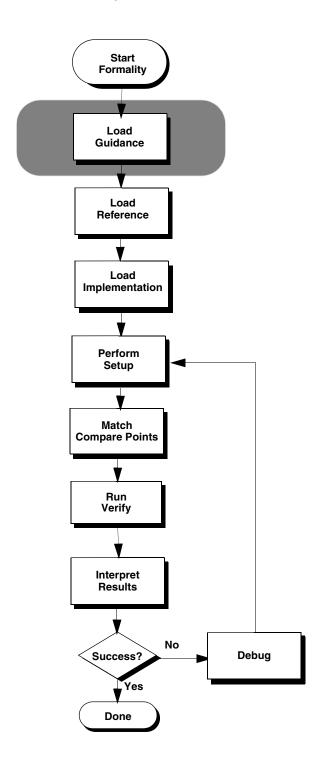
What is Guidance?

Guidance is the process by which an implementation tool, such as Design Compiler, provides setup information for formal verification. This is supplied in the form of an automated setup file (.svf).

Guidance helps Formality understand and process design changes made by other tools that are in the design flow. Formality uses this information to assist compare point matching and correctly set up verification without user intervention. It eliminates the need to enter setup information manually, a task that is time consuming and error prone. For example, during synthesis, the phase of a register might be inverted. This change is recorded in the SVF file. When the SVF file is read into Formality, the tool can account for the phase inversion during compare point matching and verification.

Figure 5-1 outlines the load guidance step in the Formality design verification process flow.

Figure 5-1 Load Guidance in the Design Verification Process Flow



Basic Usage

Note that loading your SVF file information and enabling the Automated Setup Mode is an important part of the process which comes before these steps. These steps are discussed in detail in Chapter 3, "Invocation."

This section on guidance basics is broken into the following topics:

- · Creating an SVF File
- Using the Automated Setup Mode
- Reading the SVF File
- Generating Formality Verification Setup Scripts
- Understanding the Guidance Summary

Creating an SVF File

The first step in the automated setup flow is to create the SVF file during synthesis. Design Compiler records data in the SVF file that Formality can use. Formality reads this file at the start of the verification process.

Design Compiler generates an SVF file that describes the design changes called default.svf during a synthesis session.

To change the name of the file, use the set svf command as follows:

```
dc_shell> set_svf myfile.svf
```

When Design Compiler performs an optimization, it adds the relevant Formality guidance commands to the SVF file. Use the set_svf command to specify the name of the SVF file before optimization. Otherwise, those optimizations are not written to the user-specified SVF file.

To append the setup information to an existing SVF file, use the following command:

```
dc_shell> set_svf -append myfile2.svf
```

Use the -append option of the set_svf command when you compile a design with multiple invocations of Design Compiler. In this situation, you might want to keep all the setup information in a single file rather than using a separate SVF file for each invocation.

Using the Automated Setup Mode

To use the automated setup mode in Formality, set the <code>synopsys_auto_setup</code> Tcl variable to <code>true</code> before reading in the SVF file.

When the synopsys_auto_setup variable is set to true, Formality:

- Sets the following variables:
 - o hdlin dyn array bnd check

The tool changes this variable only when this variable is used in the guide_environment command in the SVF file.

```
o hdlin_error_on_mismatch_message = false
```

```
o hdlin_ignore_embedded_configuration = false
```

```
o hdlin_ignore_full_case = false
```

```
o hdlin_ignore_parallel_case = false
```

- o signature_analysis_allow_subset_match = false
- o svf_ignore_unqualified_fsm_information = false

The tool changes this variable only when the <code>guide_fsm_reencoding</code> command is used in the SVF file.

```
O verification_set_undriven_signals = synthesis
```

```
o verification verify directly undriven output = true
```

To preserve the defaults of these variables in the automated setup mode, use the synopsys_auto_setup_filter variable. For more information, see Preserving the Variable Defaults in the Automated Setup Mode.

Reads the following commands, if they appear in the SVF file.

```
o guide_environmento guide_scan_inputo guide_port_constant
```

 Uses the setup information in the SVF file, including information about clock-gating and disabling scan insertion.

 Generates a summary report and includes it in the Formality transcript that lists the variable settings that are not default, and external constraint information passed using the SVF file.

Preserving the Variable Defaults in the Automated Setup Mode

To preserve the defaults of the variables that the tool changes during the automated setup mode, use the <code>synopsys_auto_setup_filter</code> variable before you set the <code>synopsys_auto_setup</code> variable to <code>true</code>.

Apart from specifying the variables that the tool changes during the automated setup mode, you can also specify,

- scan_input—to ignore all guide_scan_input and guide_dont_verify_scan commands in the automated setup mode.
- clock_gating—to ignore the following guide_environment commands in the automated setup mode,

```
clock_gating_latch_andclock_gating_latch_orclock_gating_and
```

o clock_gating_or

Reading the SVF File

To read an SVF file into Formality, use the <code>set_svf</code> command. The SVF file must be read in before the design. Formality uses the information in the setup file during matching as well as verification. It creates a directory named formality_svf, which contains the file, svf.txt representing all the SVF files read in and the subdirectories for the SVF file netlists.

The following example reads in the SVF file, myfile.svf.

```
fm_shell> set_svf myfile.svf
SVF set to '/home/my/designs/myfile.svf'.
1
```

If you use the <code>set_svf</code> command without specifying the automated setup file (.svf) to use, Formality resets the SVF file. However, the appropriate method for removing the stored setup data is to use the <code>remove_guidance</code> command.

You can also invoke the set_svf command from the Guidance tab in the GUI.

Generating Formality Verification Setup Scripts

You can generate Formality scripts for verification setup from the SVF files that Design Compiler generates when synthesizing the design. The generated script simplifies the verification setup by passing the design file, the library, and the verification information to the tool. The script generation supports a typical Design Compiler synthesis flow defined by the reference methodology.

The script also contains information about the path to design files, file formats, the design read parameters, and setup-free flow variable values for reading and elaborating both reference and implementation designs.

To generate a Formality script for verification setup, use the following command at the Linux shell prompt:

```
prompt> fm_mk_script svf_file [-output script_file]
```

The svf_file argument specifies the name of the source SVF file.

The -output option specifies the name of the generated script. If you do not specify the -output option, the tool writes the generated script to a file named fm mk script.tcl.

Using the Generated Formality Script

Before using the generated script, review, and modify it as applicable.

- The generated script lists design and library information. If multiple versions of the implementation design are created during synthesis, the design information is commented out.
- By default, the script runs in a directory structure similar to where synthesis is run. The search_path variable in the generated script stores the directory structure. If the directory structure is not found, the search_path variable in the script is commented out. Edit the search_path variable to specify the correct path, and uncomment the search_path variable in the script.

After you have reviewed and modified the generated script, you can use it in one of the following ways:

At the Linux shell prompt,

```
prompt> fm_shell -file script_file
```

This command starts Formality and runs the specified script to setup the designs for verification.

• At the fm shell prompt,

```
fm_shell> source -echo -verbose script_file
```

Example 5-1 Example Script Generated by the fm mk script Command

```
# Formality Verification Script generated by:
   fm_mk_script -output setup.fms "default.svf"
# Formality (R) Version E-2010.12 -- Oct 19, 2010
# Copyright (C) 2007-2010 Synopsys, Inc. All rights reserved.
# Synopsys Auto Setup Mode
set_app_var synopsys_auto_setup true
# Note: The Synopsys Auto Setup mode is less conservative than the
# Formality default mode, and is more likely to result in a successful
# verification out-of-the-box.
# Setting synopsys auto setup changes the values of the variables
# listed here below. You may change any of these variables back to
# their default settings to be more conservative. Uncomment the
# appropriate lines below to revert back to their default settings:
   # set_app_var hdlin_ignore_parallel_case true
   # set_app_var hdlin_ignore_full_case true
   # set app var verification verify directly undriven output true
   # set_app_var hdlin_ignore_embedded_configuration false
   # set_app_var svf_ignore_unqualified_fsm_information true
# Setup for instantiated or function-inferred DesignWare components
set_app_var hdlin_dwroot /sw/synth/D-2010.03-SP5
# Search path
set search_path " /users/test/ . /users/libraries/ /users/rtl/ "
# Read in the SVF file(s)
set_svf default.svf
# Define design libs
define_design_lib -r -path ./work work
```

Formality® User Guide

```
# Read in the libraries
read_db -technology_library lsi_10k.db
# set_app_var link_library * lsi_10k.db dw_foundation.sldb
# Read in the Reference Design as Verilog or VHDL source code
read_vhdl -r -libname work test.vhd
set top r:/WORK/top
# Read in the Implementation Design created from Design Compiler
# Choose the file that you want to verify
#read ddc -i example.ddc
#read ddc -i postscan.ddc
set_top i:/WORK/top
# Verify and Report
# If the verification is not successful, the session is saved and reports
# are generated to help debug the failed or inconclusive verification.
if { ![verify] } {
 set DESIGN_NAME "top" ;# The name of the top-level design
 set FMRM_FAILING_SESSION_NAME ${DESIGN_NAME}
                     ${DESIGN_NAME}.fmv_failing_points.rpt
 set FMRM_FAILING_POINTS_REPORT
                     ${DESIGN_NAME}.fmv_aborted_points.rpt
 set FMRM_ABORTED_POINTS_REPORT
 set REPORTS DIR "reports"
 file mkdir ${REPORTS DIR}
 save_session -replace ${REPORTS_DIR}/${FMRM_FAILING_SESSION_NAME}
 report_failing_points > ${REPORTS_DIR}/${FMRM_FAILING_POINTS_REPORT}
 report_aborted_points > ${REPORTS_DIR}/${FMRM_ABORTED_POINTS_REPORT}
```

For information about the setup free flow variables, see "Variables Controlled by Setup Free Flow" on page 6-19.

Understanding the Guidance Summary

The SVF file guidance summary table lists all the guide commands in the SVF file. A table similar to the one following is generated at the end of SVF file processing:

Figure 5-2 SVF file Guidance Summary Table

Status						
command		Accepted	Rejected	Unsupported	Unprocessed	Total
DEC THE COL AND THE THE DEC THE DEC AND THE					~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~	~~~~~~~~
fsm_reencodi	ng:	1	0	0	0	1
reg_constant transformati		0	3	0	0	3
share	:	0	1	0	0	1
tree	:	3	0	0	0	3
ungroup	;	2	0	0	0	2

Note: If verification succeeds you can safely ignore unaccepted guidance commands.

```
SVF files read:
  /very/long/path/name/file1.svf
  /very/long/path/name/file3.svf
SVF files produced:
  formality_svf/
svf.txt
```

This table is generated using the report_guidance -summary command.

The results of the status fields are

- Accepted Formality validated and applied the guide command to the reference design.
- Rejected Formality either could not validate or could not apply the guide command to the reference design.
- Unsupported Formality does not currently support the guide command.
- Unprocessed Formality has not processed the guide commands yet. This usually happens when a checkpoint verification has paused the processing.

Advanced Usage

This section on guidance details is broken into the following topics:

- Guidance Directory and File Structure
- Guidance Reports
- SVF File Diagnostic Messages
- Reading in Multiple Guidance Files
- Checkpoint Guidance

Guidance Directory and File Structure

Regardless of the number of SVF files read in, Formality creates a single decrypted SVF file (svf.txt), which represents the ordered automated setup guide commands that are read. All messages related to the guide commands reference this file. This file, along with the decrypted netlists, is placed under a single directory (formality_svf) in the current working directory.

The name of the formality_svf directory matches the name of the log file and follows the same numbering suffix as shown:

```
set_svf mylog1.svf mylog2.svf mylog3.svf
```

Formality creates:

```
formality.log
formality_svf/
    svf.txt
    netlists/...
...
```

The formality_svf directory is self-contained and can be moved elsewhere without need of modification.

Guidance Reports

Several commands in Formality aid in reporting SVF file information.

report_guidance

There are two main uses of the report_guidance command.

1) It produces a summary table.

This is the same as what is produced automatically after SVF file processing.

```
report_guidance -summary
```

2) It produces a user-defined text version of the SVF file.

```
report_guidance -to ascii.svf.txt
```

This version is very similar to the automatically generated formality_svf/svf.txt file but not formatted exactly the same way. For this reason using this file is not a reliable way to correlate error messages for the current run, but it could be used as input for any subsequent runs.

report_svf_operation

The report_svf_operation command reports detailed information about a specific SVF operations, or operations in the logic cone of a specified compare point.

Usage:

```
report_svf_operation #Report information about specified operations
[-command] #List of guide_* commands to search for
[-status] #List of ID numbers of commands that have the specified status
[-guide] #Report only the guide command
[-message] #Report only the messages associated with the operation
[-summary] #Report a summary table of the specified operations
operationID list #List of operation ID numbers
```

find_svf_operation

The find_svf_operation command takes guidance command names and SVF file processing status as arguments and returns a list of operation IDs.

Usage:

```
find_svf_operation #Get a list of SVF file operation IDs
[-command ] #Find operations of the specified command types
[-status ] #Find operations with the specified statuses
```

For command arguments, use what is found in the SVF file summary table. Note that you do not include the <code>guide_</code> prefix. When specifying transformation types, simply use the values map, tree, share, or merge.

For status arguments, use one of the following values: unprocessed, accepted, rejected, unsupported, or unaccepted.

SVF File Diagnostic Messages

Formality places detailed SVF file diagnostic messages in the formality.log file. Only messages pertaining to unaccepted guidance are produced and the line numbers correspond to line numbers in the formality_svf/svf.txt file.

The following example is a formality.log file message:

```
SVF Operation 4 (line 47) - fsm
Info: Cannot find reference cell 'in_cur_reg[3]'.
```

Reading in Multiple Guidance Files

The commands in the SVF files describe transformations in an incremental way. The transformations occur in the order in which the commands were applied as the RTL design was processed through design implementation or optimization. Therefore, the ability to read in multiple SVF files is important because no command in the file can be viewed completely independently. It describes the incremental transformation and relies on the context in which it is applied.

You can read multiple SVF files into Formality using the set_svf command. To read multiple SVF files, use the following syntax:

```
fm_shell> set_svf mysvf1.svf mysvf2.svf mysvf3.svf
```

By default, Formality reads the files in order of the file timestamps. Use the <code>-ordered</code> option to indicate that the list of SVF files you specify is already ordered and that the list should not be ordered according to the timestamp. If you use the <code>-ordered</code> option and list a directory or directories where the setup files are located, Formality can order the directory files in any order.

The following example sets the order of two SVF files, bot.svf and top.svf, for Formality to process:

```
set_svf -ordered bot.svf top.svf
```

Checkpoint Guidance

Checkpoint guidance provides a mechanism for the Formality and Design Compiler tools to synchronize on an intermediate netlist to simplify the verification flow.

The Design Compiler tool creates an intermediate netlist and writes the <code>guide_checkpoint</code> guidance command to the SVF file when

- Retiming a design using the set_optimize_registers command before running the compiler_ultra command
- Performing placement-aware multibit mapping of replicated registers, using the create_register_bank command

Note:

The Formality tool supports placement-aware multibit banking of nonreplicated registers without requiring checkpoint guidance.

The Formality tool verifies, then uses the checkpoint guidance from the SVF file generated by the Design Compiler tool. Using the checkpoint guidance to verify designs removes the need for a manual two-pass verification using commands to generate and verify the intermediate netlists. It also results in higher completion rates, and enables better QoR.

For more information, see "Verification Using Checkpoint Guidance" in Chapter 9.

6

Loading Designs

To run Formality, you must read in both a reference and an implementation design and any related technology libraries. This chapter describes loading designs into Formality.

This chapter contains the following sections:

- Introduction
- Basic Usage
- Advanced Usage

Figure 6-1 illustrates loading reference and implementation designs in the Formality design verification process flow.

Start Formality Load Guidance Load Reference Load Implementation Perform Setup Match Compare Points Run Verify Interpret Results No Success? Debug Yes Done

Figure 6-1 Loading Designs in the Design Verification Process Flow

Introduction

The following topics describe some of the necessary setup processes that you must do before loading the reference and implementation designs.

- Loading Design
- Top-Level Design
- Concept of Containers

Loading Design

This section describes the steps you use to load in your libraries and designs. Specific commands are described in the sections that follow.

Figure 6-2 expands upon the relevant portion of Figure 6-1 to elaborate the steps involved in the load reference and load implementation process flow in Formality.

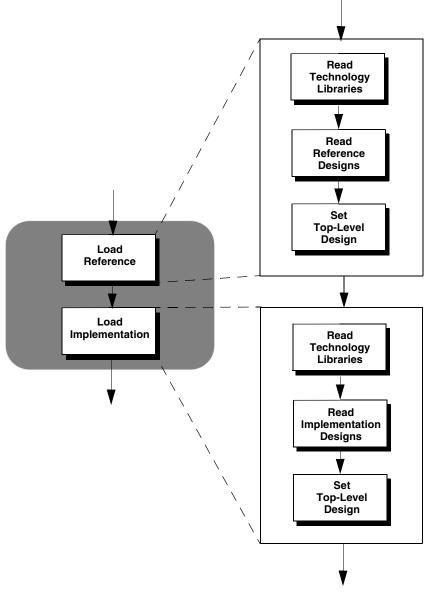


Figure 6-2 Formality Read Design Process Flow

To run Formality, you must read in both a reference and an implementation design and any related technology libraries. Note, optionally, you can first pass additional setup information from Design Compiler to Formality by using the SVF file information by using the set_svf command or by setting the automated setup mode as described in Chapter 5, "Load Guidance."

As shown in Figure 6-2, you first read in the libraries and designs that are needed for the reference, and then immediately specify its top-level design. You must set the top-level design for the reference design before proceeding to the implementation design. Next, you read in the libraries and designs that you need for the implementation design, and then immediately specify its top-level design.

Top-Level Design

Specifying the top-level design causes Formality to resolve named references, which is crucial for proper verification. This linking process appears in the transcript window. If Formality cannot resolve references, the tool issues a link error by default. When Formality resolves all references, linking is completed successfully. If the design is an RTL (VHDL or Verilog) design, Formality then performs elaboration.

You can use the hdlin_unresolved_modules variable to cause Formality to create black boxes when it encounters unresolved or empty designs during linking.

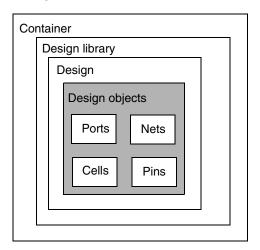
Concept of Containers

A container is a complete, self-contained space into which Formality reads designs. It is typical for one container to hold the reference design while another holds the implementation design. In general, you do not need to concern yourself with containers. You simply load designs in as either reference or implementation. This is described in "Loading the Reference Design" on page 6-7.

A container typically includes a set of related technology libraries and design libraries that fully describe a design that is to be compared against another design. A technology library is a collection of parts associated with a particular vendor and design technology. A design library is a collection of designs associated with a single design effort. Designs contain design objects such as cells, ports, nets, and pins. A cell can be a primitive or an instance of another design.

Figure 6-3 and Figure 6-4 illustrate the concept of containers.

Figure 6-3 Containers in a Hierarchical Design

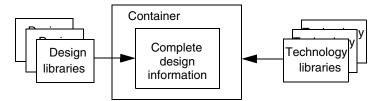


In general, to perform a design comparison, you should load all of the information about one design into a container (the reference), and all the information about the other design into another container (the implementation).

You can create, name, reuse, delete, open, and close containers. In some cases, Formality automatically creates a container when you read data into the Formality environment.

Each container can hold many design and technology libraries, and each library can hold many designs and cells. Components of a hierarchical design must reside in the same container. Figure 6-4 illustrates this concept.

Figure 6-4 Containers



In Formality, one container is always considered the current container. Unless you specifically set the current container, Formality uses the last container into which a design is read. That container remains the current container until you specifically change it or you create a new container. Many Formality commands operate on the current container by default (when you do not specify a specific container).

For more information about containers, see "Setting Up and Managing Containers" on page 6-18.

Basic Usage

Loading designs into Formality consists of three main steps:

- Load the technology libraries (optional, as needed)
- · Load the design files
- Set the top-level block to compare

These three steps are done for both the reference and implementation designs and are nearly identical in process. This section is, therefore, broken into the following two subsections, with most of the details captured solely in the load reference design section.

- Loading the Reference Design
- Loading the Implementation Design

Loading the Reference Design

This section describes in detail the steps required for loading the reference design, as shown in Figure 6-2. These steps include reading the technology libraries, reading the reference designs, and setting the top-level design.

Reading Technology Libraries

As needed, read in the technology libraries that support your reference design. If you do not specify a technology library name with the commands described in the following section, Formality uses the default name, TECH_WORK.

Reading Synopsys (.db) Format

Synopsys internal database (.db) library files are shared by default. If you read in a file without specifying whether it applies to the reference or implementation design, it applies to both.

To read cell definition information contained in .db format files,

fm shell

```
read_db file_list
[-libname library_name ]
[-merge ]
[-replace_black_box ]
```

Formality can read in other formats as technology libraries, see "Reading SystemVerilog, Verilog, and VHDL Cell Definitions" on page 6-13 for details.

Reading Designs

Read a reference design into Formality based on the language that represents it. At its most basic, (where the -r option indicates the reference design.) Specify one of the following, depending on the design type:

fm_shell

```
read_verilog -r files
read_sverilog -r files
read_vhdl -r files
read_ddc -r files
read_milkyway -r files
read_db -r files
```

For more information about the fm_shell command options, see the man pages.

In the Formality shell, you represent the design hierarchy by using the designID argument. The designID argument is a path name whose elements indicate the container (by default, r or i), library, and design name.

Reading Verilog and SystemVerilog Designs

Verilog and SystemVerilog descriptions information must be in the form of synthesizable RTL or a structural netlist.

fm shell

```
Specify:
read_verilog
[-r | -i | -container containerID ]
[-libname library_name ]
[-netlist ]
[-95 | -01 | -05] file_list
or
```

fm_shell

```
read_sverilog
[-r | -i | -container containerID ]
[-technology_library ]
[-libname library_name ]
[-3.1a | -05 | -09 | -12] file_list
```

When reading in Verilog designs, set the hdlin_auto_netlist variable to true to automatically use the Verilog netlist reader instead of the default reader. Using the Verilog netlist reader might improve the design loading time. If the Verilog netlist reader is unsuccessful, Formality uses the default reader.

If you have Verilog simulation libraries or design modules that you want to link to the reference or implementation designs, use the $\neg v$ and $\neg y$ options from VCS. These options specify the library or file for the module references. They do not support Verilog technology library cells with mixed user-defined primitives and synthesizable constructs.

Note:

The SystemVerilog standard specified by using the read_sverilog command overrides the standard specified using the hdlin_sverilog_std variable.

To ignore SystemVerilog module names while reading Verilog designs, specify the module name with the hdlin_sv_blackbox_modules variable. The tool ignores the specified modules when reading the RTL file. Therefore, if another design with the same name exists in the container (for example, a .db file), the tool links to that design.

Reading VHDL Designs

VHDL cell definition information must be in the form of synthesizable RTL or a structural netlist.

fm shell

```
Specify:
read_vhdl
[-r | -i | -container containerID]
[-libname library_name]
[-87 | -93 | -2008]
file_list
```

The default is 2008. When you specify more than one VHDL file to be read with a single read_vhdl command, Formality automatically attempts to read your files in the correct order. If the list of files includes VHDL configurations, this feature does not work. Disable it

by setting the hdlin_vhdl_strict_libs variable to false before using the read_vhdl command. If you are using multiple read_vhdl commands, you must issue them in the correct compilation order.

Note:

The VHDL standard specified by using the read_vhdl command overrides the standard specified using the hdlin_vhdl_std variable.

Reading .ddc Format Designs

To read design netlists and technology libraries from .ddc format databases, use the read_ddc command. This command reads design information, including netlists and technology libraries, from .ddc databases produced by Design Compiler.

To read designs from a .ddc format database into a container,

fm shell

```
Specify:
    read_ddc
[-r | -i | -container containerID ][-libname library_name ]
[-technology_library ] file_list
```

Formality reads in files formatted as Synopsys .ddc format database designs. Formality returns a 1 if the design is successfully read; it returns a 0 if the design is not successfully read into the destination container. Existing designs in the destination container are overwritten with the designs that are read.

Reading Milkyway Designs

To read design netlists and technology libraries from Milkyway, use the read_milkyway command. This command reads design information, including netlists and technology libraries, from Milkyway databases.

fm_shell

```
Specify:

read_milkyway

[-r | -i | -container containerID ]

[-libname library_name ][-technology_library ]

[-version version_number ]

-cell_name mw_cell_namemw_db_path
```

Use the mw_logic0_net and mw_logic1_net variables to specify the name of the Milkyway ground and power net, respectively.

Reading Block Abstractions

Block abstractions improve verification of blocks that use optimizations where the boundary logic of the block has changed. To use a block abstraction during the verification of gate-level designs, read the block abstraction as a subblock design into the reference container and the modified or optimized block abstraction into the implementation container. You can read block abstractions of designs that are optimized using Design Compiler.

To read a block abstraction, use the read_ddc -block_abstraction or read_milkyway -block_abstraction command.

Example 6-1 The Block Abstraction Flow

```
fm_shell> read_verilog -r top.v
fm_shell> read_ddc -r -block_abstraction sub_compile1_bam.ddc
fm_shell> set_top r:/WORK/top
fm_shell> read_verilog -i top.v
fm_shell> read_ddc -i -block_abstraction sub_compile2_bam.ddc
```

Block abstractions are not useful when verifying an RTL netlist against a gate-level netlist because the boundary points of the abstracted block cannot be matched with the RTL netlist.

Reading .db Format Designs

See "Reading Synopsys (.db) Format" on page 6-7 for information about reading in .db design files.

Setting the Top-Level Design

To set the top-level design for the reference design,

fm_shell

```
Specify:
set_top
[-vhdl_arch name ]
[moduleName | designID | -auto ]
[-parameter value ]
```

If you are elaborating VHDL and you have more than one architecture, use the $-vhdl_arch$ option.

The set_top command tells Formality to set and link the top-level design. If you are using the default r and i containers, this command also sets the top-level design as the reference or implementation design.

For additional information about setting parameters, see "Setting the Top-Level Design" on page 6-11.

Loading the Implementation Design

This section provides an overview of the read-design process flow for the implementation design. The process for loading the implementation design is broadly similar to that described in "Loading the Reference Design" on page 6-7.

Note:

If you already specified a .db library for the reference design, it is automatically shared with the implementation design.

Many Formality shell commands can operate on either the reference or implementation design. These commands all have a switch to indicate which design container is used for that command. The -r switch refers to the reference design or container. The -i switch refers to the implementation design or container. Use the -i option to specify the implementation container or use the $-container\ container\ name$ option to provide a specific container name. From within the GUI, use the Implementation tab to read an implementation design.

For information about the fm_shell commands and their options, see the man pages. For information about special Verilog considerations, see "Verilog Simulation Data" on page 6-14. Otherwise, if you have Verilog simulation data, use the -vcs options with the read_verilog command.

Advanced Usage

There is a range of optional functionality available to you through use of the containers into which the Formality designs are read. You can use the Setup Free Flow to control certain variables. The functionality associated with these options are discussed under the following headings:

- Reading Technology Libraries
- Setting the Top-Level Design
- Setting Up and Managing Containers
- Variables Controlled by Setup Free Flow

Reading Technology Libraries

SystemVerilog, Verilog, and VHDL cell definition information must be in the form of synthesizable RTL or a structural netlist. In general, Formality cannot use behavioral constructs or simulation models, such as VHDL VITAL models.

Using the 'celldefine Verilog Attribute

When reading libraries in Formality, you use the `celldefine Verilog attribute to indicate that a logic description is a library cell. This attribute might be necessary to take advantage of the extra processing needed to build the correct logical behavior. However, because the `celldefine attribute is not required by Verilog, many libraries do not include it in the source file. Using it would require modifications to your source file, which is not always possible.

Reading SystemVerilog, Verilog, and VHDL Cell Definitions

To read cell definition information contained in SystemVerilog, Verilog, or VHDL RTL files, do the following:

fm_shell

```
Specify:

set_app_var hdlin_library_file file

set_app_var hdlin_library_directory
directory

read_verilog

[-r | -i | -container containerID ]

[-technology_library ]

[-libname library_name ]

[-95 | -01 | -05] file_list

or

read_sverilog

[-r | -i | -container containerID ]

[-technology_library ]

[-technology_library ]

[-technology_library ]

[-libname library_name ]

[-3.1a | -05 | -09 | -12] file_list
```

Formality[®] User Guide Version O-2018.06

fm_shell

```
or
read_vhdl
[-r | -i | -container containerID ]
[-technology_library ]
[-libname library_name ]
[-87 | -93 ]
file_list
```

The hdlin_library_file variable designates all designs contained within a file or set of files as technology libraries. The value you set for this variable is a space-delimited list of files.

The hdlin_library_directory variable designates all designs contained within directories as technology libraries. The value you set for this variable is a space-delimited list of directories. After you mark a design for library processing, any subdesign would also go through that processing.

The fm_shell commands are not listed with all their options. The options listed in this table pertain to reading in technology library data only.

Use the <code>-technology_library</code> option to specify that the data goes into a technology library rather than a design library. This option does not support mixed Verilog and VHDL technology libraries.

Verilog Simulation Data

You generally read in Verilog simulation libraries by specifying VCS options with the read_verilog command when you read in designs, as discussed in "Reading Designs" on page 6-8.

To read cell definition information contained in Verilog simulation library files,

fm shell

```
Specify:
read_verilog -technology_library
-vcs VCS options
```

The reader extracts the pertinent information from the Verilog library to determine the gate-level behavior of the design and generates a functional description of the Verilog library cells.

To generate the gate-level models, the reader parses the Verilog modules and user-defined primitive descriptions. With this information it creates efficient gate-level models that can be used for verification.

A Verilog simulation library is intended for simulation, not synthesis. Therefore, the reader might make certain assumptions about the intended gate-level functions of the user-defined primitives in the simulation model. The reader generates comprehensive warning messages about these primitives to help you eliminate errors and write a more accurate model.

Each warning message is identified by a code. To obtain more information, look at the man page for the code. For example, if Formality reports 'Error: Can't open file xxxx (FM-016),' use the man FM-016 command for information.

The library reader supports the following features:

- Sequential cells (each master-slave element pair is merged into a single sequential element)
- Advanced net types: wand, wor, tri0, tri1, and trireg
- Unidirectional transistor primitives: pmos, nmos, cmos, rpmos, rnmos, and rcmos
- Pull primitives (a pull-up or pull-down element is modeled as an assign statement with a value of 1 or 0)

Library Loading Order

Formality has the ability to load and manage multiple definitions of a cell, such as synthesis .db format files, simulation .db format files, and Verilog or VHDL netlists. The order in which the library files are loaded determines which library model is used by Formality. If the libraries are not loaded in the correct sequence, it can lead to inconsistent or incorrect verification results.

If you are a library provider, you should deliver explicit Formality loading instructions for multiple libraries. One way to do this is to provide a Formality script that loads the library files (such as .db, .v, and .vhd) in the correct order.

Single-Source Packaging

It is better to provide all the required functionality in a single source, either a synthesis (.db) or simulation (.v) file. Using a single source reduces support costs and maintenance requirements. However, you might choose to use multiple sources of functional information.

Multiple-Source Packaging

If you are a silicon vendor who wants to use multiple library sources or augment your synthesis libraries with simulation or RTL descriptions, you should specify the order in which the libraries are to be loaded.

Setting the Top-Level Design

When setting the top-level design, be aware of the following factors:

- The tool must read the reference or implementation design before you run the set_top command. Do not read in the implementation design until you have specified the set_top command for the reference design.
- The set_top command always applies to the design data previously read into Formality (whether it is the implementation or reference design). An error is issued if the design is not loaded.
- You cannot save, restore, or verify a design until you have specified the set_top command.
- Be sure that the module or design you specify is your top design. If not, you must remove the reference design and start over. This also holds true when you are loading the implementation design.
- Use the -auto option if the top-level design is unambiguous. You generally specify a module or design by name in cases where you do not want the actual top-level design to be considered the top for the current session or when you have multiple modules that could be considered at the top level.
- Set the top-level design to the highest level you plan to work with in the current session.
- After you set the top-level design, you cannot change it, whereas you can change the reference or implementation design to be verified using the set_reference_design, set_implementation_design, or verify command. The design you specify must reside within the top-level design.

Setting Parameters on the Top-Level Design

To set parameters in your top-level design, use the <code>set_top</code> <code>-parameter</code> command. Use the <code>-parameter</code> option to specify a new value for your design parameters. You can set the parameter only on the top-level design. Parameters must be Verilog or VHDL generics. The parameter values can either be integers or specified in the format <code>param_name</code> <code>hexadecimal value format base</code> 'h <code>value</code>.

For VHDL designs, the generics might have the following data types for the parameter value:

- integer
- bit
- bit_vector
- std_ulogic

- std_ulogic_vector
- std_logic
- std_logic_vector
- signed (std logic arith and numeric std)
- unsigned (std_logic_arith and numeric_std)

For additional information about setting parameters, see the set_top man page.

Generating Simulation or Synthesis Mismatch Report

You can generate a report on any simulation or synthesis mismatches in your design after setting the top level of your design. Formality automatically generates an RTL report summary describing any simulation or synthesis mismatches when you run set_top (or read_container). Running the report_hdlin_mismatches command after set_top (or read_container) generates a verbose report detailing the various constructs encountered and their state.

Linking the Top-Level Design Automatically

If you have straightforward designs, such as Verilog designs, you can use the hdlin_auto_top variable rather than the set_top command to specify and link the top-level module, but only when you specify one read_verilog command for the container.

To set the top-level design with the hdlin_auto_top variable, do the following:

fm_shell Specify: set_app_var hdlin_auto_top true

The hdlin_auto_top variable causes Formality to determine the top-level module and automatically link to it. This variable applies only when you are reading in a Verilog design. If you are reading in technology libraries, Formality ignores this variable. Formality issues an error message if it cannot determine the top-level design. In this case, you must explicitly specify the top design with the set_top command. If there are multiple VHDL configurations or architectures that could be considered the top level, Formality issues a warning and sets the top-level design to the default architecture.

The hdlin_auto_top variable requires you to use a single read command to load the design. You cannot use it for mixed-language designs or for designs that use multiple design libraries or multiple architectures or configurations.

Setting Up and Managing Containers

As described in "Concept of Containers" on page 6-5, a container is a complete, self-contained space into which Formality reads designs. Each design to be verified must be stored in its own container. If you follow the steps described in "Reading Technology Libraries" on page 6-7, Formality uses default containers named r and i.

You generally do not need to work directly with containers. However, you can work with containers,

- To change the name of the reference and implementation containers from the default ${\tt r}$ and ${\tt i}$
- For backward compatibility with existing Formality scripts
- When you apply user-defined external constraints on your designs

Note:

The r and i containers exist by default, even if empty. When you specify them as the container ID with the create_container command, Formality issues a warning that the container already exists.

To create a container, do the following:

fm_shell
Specify:
create_container
[containerID]

Formality uses the containerID string as the name of the new container. If using this command, you must do so before reading in your libraries and designs.

Alternatively, you can specify a container with the <code>-container</code> containerID option to the read_db, read_ddc, read_milkyway, read_sverilog, read_verilog, Of read_vhdl command. If you specify a container ID in which to place a technology library, the library can be seen only in that container. This is called an unshared technology library. If you do not specify a container, the technology library can be seen in all current and future containers. This is called a shared technology library.

When you create a new container, Formality automatically puts the generic technology (GTECH) library into the container. The GTECH library contains the cell primitives that Formality uses internally to represent RTL designs.

In fm_shell, Formality considers one design to be the current design. When you create or read into a container, it becomes the current container.

After the current container is set, you cannot operate on any other container until you either:

- Set the top-level design using the set_top command
- Remove the container and its contents using the remove_container command. For the
 default r and i containers, this command removes only the contents

In the GUI, the concept of a current container does not apply directly. You simply work on the reference and implementation designs. You can execute Formality shell commands that rely on the current container concept. However, the GUI recognizes only the containers that store the reference and implementation designs. To view a third design in the GUI, you must choose it as a reference or implementation design.

Note:

When you create a new container, Formality automatically adds any shared technology libraries. If you do not want a particular shared technology library in the new container, you must specifically remove it.

The save_session command is not executed if you have not already linked the top-level design using the set_top command.

In the GUI, you can view the reference and implementation containers by choosing Designs > Show Reference and Designs > Show Implementation. To save the design, choose File > Save.

Variables Controlled by Setup Free Flow

The following topics describe the setup free flow and the variables that the flow controls:

- Variables to Control Bus Names
- Variables to Control Parameter Names
- Variables to Control Case Behavior

Variables to Control Bus Names

The guide_environment command uses the values specified using the following variables in the SVF file irrespective of whether the synopsys_auto_setup variable is set to true or false:

- bus dimension separator style
- bus extraction style
- bus_range_separator_style

Variables to Control Parameter Names

The guide_environment command uses the values specified using the following variables in the SVF file irrespective of whether the <code>synopsys_auto_setup</code> variable is set to <code>true</code> or <code>false</code>:

- hdlin_naming_threshold
- template_naming_style
- template_parameter_style
- template_separator_style

Variables to Control Case Behavior

The following variables are set to false when the synopsys_auto_setup variable is set to true:

- hdlin_ignore_parallel_case
- hdlin_ignore_full_case
- svf ignore unqualified fsm information

7

Performing Setup

After reading designs into the Formality environment and linking them, set the design-specific options for Formality to perform verification. For example, if you are aware of certain areas in a design that Formality cannot verify, you can prevent the tool from verifying the areas. Or, to improve the performance of verification, you can declare blocks in two separate designs black boxes.

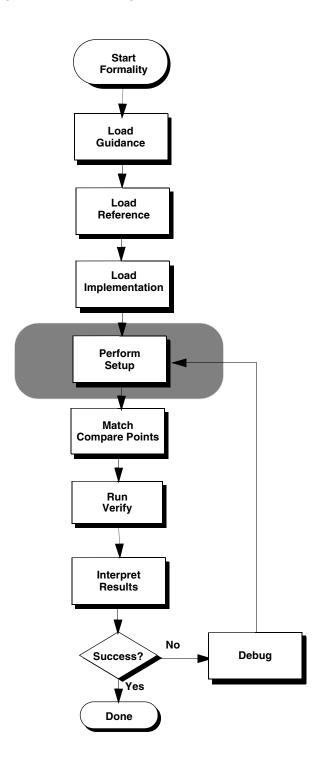
This chapter describes how to setup designs for verification in the following sections:

- Common Operations
- Less Common Operations

Figure 7-1 outlines the timing of performing setup within the design verification process flow.

Formality[®] User Guide Version O-2018.06

Figure 7-1 Performing Setup in the Design Verification Process Flow



Common Operations

Tasks and procedures that are performed often to setup a design for verification are described in the following subsections:

- Black Boxes
- Specifying Constants
- External Constraints
- Combinational Design Changes
- Sequential Design Changes
- Retimed Designs
- Low-Power Designs

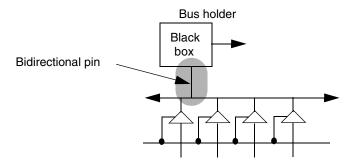
Black Boxes

A black box represents logic whose function is unknown. Black boxes can cause verification failures because input pins become compare points in the design. If black boxes in the reference design do not match those in the implementation design, the compare points are not matched.

In addition, compare point mismatches can occur when black box buses are not normalized in the same manner. When Formality encounters a missing design, it normalizes the bus on the black box to the form WIDTH-1:0. However, when it encounters an empty design, it does not normalize black box buses, and bus indexes are preserved. Therefore, you must either not include a design or use an empty design for both the implementation and the reference design so that buses are normalized in a like manner.

When Formality verifies a design, its default action is to consider a black box in the implementation design equivalent to its counterpart in the reference design. This behavior can be misleading in cases where designs contain many unintentional black boxes, such as in an implementation design that uses black boxes as bus holders to capture the last state placed on a bus. Figure 7-2 shows an example.

Figure 7-2 Black Boxes



In this example a bidirectional pin is used to connect to the bus. Because this pin is bidirectional, the bus has an extraneous driver. If the reference design does not use similar bus holders, the implementation design fails verification. To solve this problem, you can declare the direction "in." Assigning the pin a single direction removes the extraneous driver.

By default, Formality stops processing and issues an error message if it encounters unresolved designs (those that cannot be found during the linking process) and empty designs (those with an interface only). For example, suppose a VHDL design has three instances of designs whose logic is defined through associated architectures. If the architectures are not in the container, Formality halts.

You can use the hdlin_unresolved_modules variable to cause Formality to create black boxes when it encounters unresolved or empty designs during linking.

Note:

Setting the hdlin_unresolved_modules variable to black box can cause verification problems.

The verification_ignore_unmatched_implementation_blackbox_input variable can be used to cause Formality to allow successful verification of unmatched input pins on matched black boxes in the implementation design.

Because of the uncertainty that black boxes introduce to verification, in Formality you can control how the tool handles them. You can,

- Load only the design interface (ports and directions) even though the model exists
- Mark a design as a black box for verification even though its model exists and the design is linked
- Report a list of black boxes
- Perform an identity check between comparable black boxes
- Set the port and pin directions

These techniques are described in the following sections.

Loading Design Interfaces

To mark an object as a black box, specify the hdlin_interface_only variable. Formality benefits from having the pin names and directions supplied by this variable.

Note:

Specify the hdlin interface only variable before reading in your designs.

To load only the pin names and directions for designs, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Reference or Implementation.
	2. Click Options.
set_app_var hdlin_interface_only "designs"	3. Click the Variables tab.
	 In the Read interface only designs (hdlin_interface_only) box, enter the list of designs.
	5. Click OK.

The hdlin_interface_only variable enables you to load the specified designs as black boxes, even when their models exist. This capability is useful for loading in RAM, intellectual property (IP), and other special models. When you specify report_black_boxes, these designs are attributed with an "I" (interface only) to indicate that you specified this variable.

This variable supports wildcard characters. It ignores syntax violations within specified designs. However, if Formality cannot create an interface-only model due to syntax violations in the pin declarations, it treats the specified design as missing.

Modules names must be simple design names. For example, to mark all RAMs named SRAM01, SRAM02, and so on in a library as black boxes, use the following command:

```
fm_shell (setup)> set_app_var hdlin_interface_only "SRAM*"
```

This variable is not cumulative. Subsequent specifications cause Formality to override prior specifications. Therefore, if you want to mark all RAMs with names starting with DRAM* and SRAM* as black boxes, for example, specify both on one line.

```
fm shell (setup) > set app var hdlin interface only "DRAM* SRAM*"
```

Marking a Design as a Black Box for Verification

To mark a design as a black box for verification, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
set_black_box designID	set_black_box designID

You specify this command for a loaded design. When you specify report_black_boxes, these designs are attributed with an "S" to indicate that you specified this command. To remove this attribute, use the remove_black_box command.

Use the $\mathtt{set_black_box}$ command to specify the designs that you want to mark as black boxes. The designs that you specify with the $\mathtt{hdlin_interface_only}$ variable on unresolved references always retain their black box attribute.

Note:

It is also possible to mark a design as a black box through the hierarchy browser. Search the hierarchy browser to locate and select the design that is to made a black box. Then set the block as a black box from the GUI by clicking on the appropriate symbol (shaped like a black chip) on the same hierarchy browser.

Reporting Black Boxes

To report black boxes,

fm_shell	GUI
Specify:	At the Formality prompt, specify:
report_black_boxes	report_black_boxes
[design_list -r -i	[design_list -r -i
-container containerID] [-all]	-container containerID] [-all]
[-unresolved] [-empty]	[-unresolved] [-empty]
[-interface_only]	[-interface_only]
[-set_black_box]	[-set_black_box]
[-unread_tech_cell_pins]	[-unread_tech_cell_pins]

By default, this command lists the black boxes for both the reference and implementation designs. Formality issues an error message if these are not set. You can restrict the report to only the implementation or reference design, or to a container or design that you specify.

In addition, the report lists a reason, or attribute, code for each black box, as follows:

- U: Unresolved design.
- E: Empty design. An asterisk (*) next to this code indicates that the design is not linked with the set_top command. After it is linked, the design appears as a black box if it is not empty.
- I: Interface only, as specified by the hdlin_interface_only variable.
- S: Set with the set_black_box command.

You can report only black boxes of a certain attribute by using the <code>-unresolved</code>, <code>-empty</code>, <code>-interface_only</code>, and <code>-set_black_box</code> options. The default <code>-all</code> option reports all four black box types.

The report output during set_top processing also lists black boxes.

Note:

Formality places black boxes created due to unresolved designs in the FM_BBOX library.

Performing Identity Checks

To perform an identity check between two comparable black boxes,

fm_shell	GUI
Specify:	Choose Edit > Formality Tcl Variables. The Formality Tcl Variables dialog box appears.
<pre>set_app_var verification_blackbox_match_mode identity</pre>	 From Matching, select the verification_blackbox_match_mode variable.
	3. Select Identity.
	4. Choose File > Close.

By default, the <code>verification_blackbox_match_mode</code> variable is set to <code>any</code>, and Formality compares the two black boxes regardless of the library or design names.

When you set the <code>verification_blackbox_match_mode</code> variable to <code>identity</code>, Formality matches the two black boxes only if they have the same library and design names. If the black boxes are identical, they are considered equivalent during verification.

To specify user-defined matches on black boxes with different names, use the set user match command.

Setting Pin and Port Directions for Unresolved Black Boxes

By definition, you do not know the function of a black box. For unresolved black boxes, Formality attempts to define pin direction from the connectivity and local geometries. If the tool cannot determine the direction, it assumes that the pin is bidirectional. This assumption could result in an extra driver on a net in one design that does not exist in the other.

To avoid this failure, you can create a wrapper for the block with the pin directions defined. You can use a Verilog module or VHDL entity declaration. This takes the guesswork out of determining pin direction. You can also use the set_direction command to define pin direction.

To redefine a black box pin or port direction, use either the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
set_direction	set_direction
objectID direction	objectID direction

For *objectID*, supply the object ID for an unlinked port or pin. (You cannot set the direction of a linked port or pin.) For *direction*, specify either in, out, or inout.

Specifying Constants

Formality recognizes two types of constants: design and user-defined. Design constants are nets in your design that are tied to a logic 1 or 0 value. User-defined constants are ports or nets to which you attach a logic 1 or 0 value, using Formality commands.

User-defined constants are especially helpful when several problem areas exist in a circuit and you want to isolate a particular trouble spot by disabling an area of logic. For example, suppose your implementation design has scan logic and you do not want to consider it in the verification process. You can assign a constant to the scan-enable input port to disable the scan logic and take it out of the verification process.

You can apply a user-defined constant to a port or net. However, if you assign a constant to a net with a driver, Formality displays a warning message.

Formality tracks all user-defined constants and generates reports. You can specify how Formality propagates constants through different levels of the design hierarchy.

You can manage user-defined constants by performing the tasks in the following sections.

Defining Constants

To set a net, port, cell, or pin to a constant state of 0 or 1, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Setup > Constants.
set_constant [-type type]	Click Set and click the Reference or Implementation tab.
instance_path constant_value	Navigate through the instance tree view and select the instance.
	 Set a Constant value for the selected instance.
	5. Click Apply.

For *constant_value*, specify either 0 or 1. If more than one design object shares the same name as that of the specified object, use the <code>-type</code> option and specify the object type (either port or net). You can specify an object ID or instance-based path name for *instance_path*. Use the latter to apply a constant to a single instance of an object instead of all instances. In addition, you can use wildcards to specify objects to be set constant.

Removing User-Defined Constants

To remove a user-defined constant, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	1. Click Setup > Constants.
remove_constant -all	2. Select a constant.
or	3. Click Remove.
remove_constant	
[-type ID_type] object_ID	

If more than one design object shares the same name as that of the specified object, use the <code>-type</code> option and specify port or net (whichever applies) for the type. You can specify an object ID or instance-based path name for <code>object_ID</code>. Use the latter if you want to remove a constant on a single instance of an object instead of all instances.

Listing User-Defined Constants

To list user-defined constants, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Setup > Constants.
report_constants	
[instance_path]	

If you omit <code>instance_path</code>, Formality returns a list of all user-defined constants. You can specify an object ID or instance-based path name for <code>instance_path</code>. Each line of the report shows the constant value, design object type, and design object name. For information about this command, see the man page.

Reporting Setup Status

To report design statistics, design read warning messages and user specified setup, use the report_setup_status command in the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
report_setup_status	report_setup_status
[-design_info]	[-design_info]
[-hdl_read_messages]	[-hdl_read_messages]
[-commands]	[-commands]

By default, the report_setup_status command lists the critical design setup before running the match and verify commands. You can run this command after reading and linking both the reference and implementation designs.

Use the report_setup_status -design_info command to report design specific settings that are set using the set command and statistics.

Use the report_setup_status -hdl_read_messages command to report the warning information messages that are issued by Formality during design read and linking.

Use the report setup status -commands command to report the user-specified setup.

When you do not use any of the available options with the report_setup_status command, a consolidated report with all the information is generated.

External Constraints

Sometimes you might want to restrict the inputs used for verification by setting an external constraint. By setting an external constraint, you can limit the potential differences between two designs by eliminating unused combinations of input values from consideration, thereby reducing verification time and eliminating potential false failures that can result from verification with the unconstrained values.

When you define the allowed values of, and relationships between, primary inputs, registers, and black box outputs, and allow the verification engine to use this information, the resulting verification is restricted to identify only those differences between the reference and implementation designs that result from the allowed states.

Typical constraint types that you can set are

- One-hot: One control point at logic 1; others at logic 0.
- One-cold: One control point at logic 0; others at logic 1.
- Coupled: Related control points always at the same state.
- Mutually exclusive: Two control points always at opposite states.
- User-defined: You define the legal state of the control points.

The following paragraphs describe three cases where setting external constraints within verification is important.

In the most common case, your designs are part of a larger design, and the larger design defines the operating environment for the designs under verification. You want to verify the equivalence of the two designs only within the context of this operating environment. By using external constraints to limit the verification to the restricted operating conditions, you can eliminate the false negatives that can arise out of the functions not exercised.

In the second case, one of the designs you want to verify was optimized under the assumption that some control point conditions cannot occur. The states outside the assumed legal values can be true don't care conditions during optimization. If the equivalent behavior does not occur under these invalid stimulus conditions, false negatives can arise during verification. Setting the external constraints prevents Formality from marking these control points as false negatives under these conditions.

In the third case, you want to constrain the allowed output states for a black box component within the designs being verified. Using external constraints eliminates the false negatives that can arise if the black box component is not constrained to a subset of output state combinations.

You can set and remove external constraints, create and remove constraint types, and report information about the constraints you have set.

Defining an External Constraint

To define an external constraint, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
set_constraint type_name	set_constraint type_name
[-name constraint_name	[-name constraint_name
[-map map_list1 map_list2]	[-map map_list1 map_list2]
constraint_type	constraint_type
<pre>control_point_list [designID]</pre>	<pre>control_point_list [designID]]</pre>

For type_name, supply the type of external constraint you want to use. For control_point_list, specify the list of con trol points (primary inputs, registers, and black box outputs) to which the constraint applies. Use the designID argument to specify a particular design; the default is the current design.

Creating a Constraint Type

To create a user-defined constraint type and establish the mapping between the ports of a design that define the constraint and control points in the constrained design, in the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
create_constraint_type	create_constraint_type
type_name	type_name
[designID]	[designID]

For type_name, specify the type of constraint. For designID, specify a particular design; otherwise, the default is the current design.

User-defined constraints allow you to define the allowable states of the control points by specifying a constraint module. The constraint module is a design you create that determines whether the inputs are legal (care) or illegal (don't care) states. When the output of the constraint module evaluates to 1, the inputs are in a legal state. For information about don't care cells, see "Concept of Consistency and Equality" on page 1-10.

When you later reference the user-defined constraint from the set_constraint command, Formality automatically hooks the constraint module design into the target of the

set_constraint command and uses the output of the module to force the verification to consider only the legal states for control points.

A constraint module has the following characteristics:

- One or more inputs and exactly one output
- Outputs in logic 1 for a legal state; otherwise logic 0
- No inouts (bidirectional ports)
- No sequential logic
- No three-state logic
- No black boxes

Removing an External Constraint

To remove an external constraint from the control points of a design, use either the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
remove_constraint	remove_constraint
constraint_name	constraint_name

For constraint_name, specify the name of the constraint to remove.

Removing a Constraint Type

To remove external constraint types, use either the Formality shell or the GUI as shown:

fm_shell	GUI
<pre>Specify: remove_constraint_type type_name</pre>	At the Formality prompt, specify:remove_constraint_type type_name

For *type_name*, specify the type of user-defined constraint to remove.

Reporting Constraint Information

To report information about the constraints set in your design, use either the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
report_constraint	report_constraint
[-long] constraint_name	[-long] constraint_name

For constraint name, specify the name of the constraint you want to obtain a report.

Reporting Information About Constraint Types

To report information about constraint types set in your design, use either the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
report_constraint_type	report_constraint_type
[-long] type_name	[-long] type_name

For more information about report constraint type command, see the man page.

Combinational Design Changes

This section describes how to prepare designs with combinational design changes, such as

- Internal scan insertions
- Boundary-scan insertions
- · Clock tree buffers

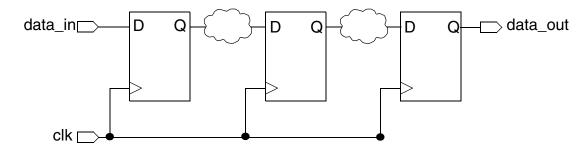
Your design can also include sequential design changes. For more information, see "Sequential Design Changes" on page 7-18.

Disabling Scan Logic

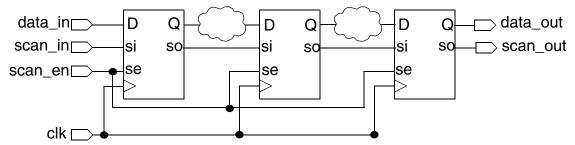
Insert internal scan to set and observe the state of the registers internal to a design. During scan insertion, the scan flops replace flip-flops. The scan flops are then connected into a long shift register. The additional logic added during scan insertion means that the combinational function has changed, as shown in Figure 7-3.

Figure 7-3 Internal Scan Insertion

Pre-Scan



Post-Scan



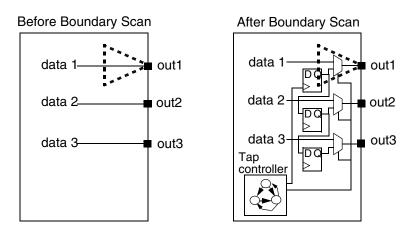
After determining which pins disable the scan circuitry, disable the inserted scan logic by specifying the disabling value (either 0 or 1) with the set_constant command. For more information, see the procedure in "Defining Constants" on page 7-9.

Disabling Boundary Scan in Your Designs

Boundary scan is similar to internal scan in that it involves the addition of logic to a design. This added logic makes it possible to set and observe the logic values at the primary inputs and outputs (the boundaries) of a chip, as shown in Figure 7-4. Boundary scan is also referred to as the IEEE 1149.1 Std. specification.

Formality[®] User Guide Version O-2018.06

Figure 7-4 Boundary-Scan Insertion



Designs with boundary-scan registers inserted require setup attention because

- The logic cones at the primary outputs differ
- The boundary-scan design has extra state-holding elements

Boundary scan must be disabled in your design in the following cases:

- If the design contains an optional asynchronous TAP reset pin (such as TRSTZ or TRSTN), use set constant on the pin to disable the scan cells.
- If the design contains only the four mandatory TAP inputs (TAS, TCK, TDI, and TDO), force an internal net of the design with the set constant command. For example,

```
fm_shell (setup)> set_constant gates:/WORK/TSRTS 0
fm_shell (setup)> set_constant gates:/WORK/alu/somenet 0
```

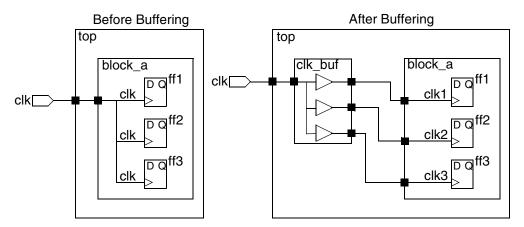
Specify 0 for the set_constant command, as described in the procedure in "Defining Constants" on page 7-9.

Managing Clock Tree Buffering

Clock tree buffering is the addition of buffers in the clock path to allow the clock signal to drive large loads, as shown in Figure 7-5.

Formality[®] User Guide Version O-2018.06

Figure 7-5 Clock Tree Buffer



Without the correct setup, verification of block_a fails. However, it would succeed with top-down verification. As shown in the figure, before buffering, the clock pin of ff3 is clk. After buffering, the clock pin of ff3 is clk3. The logic cones for ff3 are different, resulting in a failing point.

To manage the clock tree buffering, you must use the set_user_match command to specify that the buffered clock pins are equivalent. With the set_user_match command you can match one object in the reference design to multiple objects in the implementation design (1-to-n matching). For example, if you want to match a clock port, clk, in the reference design to three clock ports in the implementation design, clk, clk1, and clk2, you can use

```
set_user_match r: /WORK/design/clk i:/WORK/design/clk i:/WORK/
design/clk1 i:/WORK/design/clk2
```

Alternatively, you can issue multiple commands for each port in the implementation:

```
set_user_match r: /WORK/design/clk i:/WORK/design/clk
set_user_match r: /WORK/design/clk i:/WORK/design/clk1
set_user_match r: /WORK/design/clk i:/WORK/design/clk2
```

If you know a clock port is inverted, use the <code>-inverted</code> option to the <code>set_user_match</code> command. Therefore, if your reference design had a clock port, clk, and your implementation design had a clk port and an inverted clock port, clk_inv, you would use the following command:

```
set_user_match r:/WORK/design/clk i:/WORK/design/clk
set_user_match -inverted r:/WORK/design/clk i:/WORK/design/clk_inv
```

For more information about the set_user_match command, see the man page.

Sequential Design Changes

Similar to the combinational design changes described in "Combinational Design Changes" on page 7-14, sequential design changes also require setup before verification. Sequential design changes include:

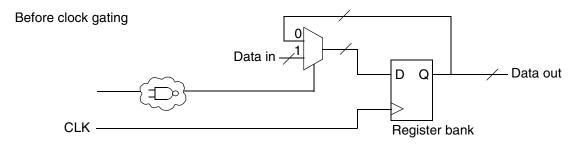
- Clock gating
- Automatic clock gating
- Pushing inversions across registers
- Retiming

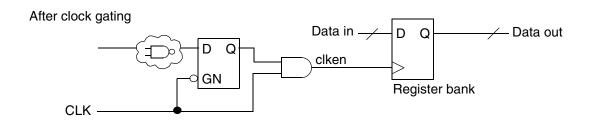
FSM re-encoding and module retiming are also considered sequential design changes. For more information, see "Re-encoded Finite State Machines" on page 7-38 and "Retimed Designs" on page 7-25.

Setting Clock Gating

Clock gating applies to synchronous load-enable registers, which are groups of flip-flops that share the same clock and synchronous control signals. Clock gating saves power by eliminating the unnecessary activity associated with reloading register banks. In its simplest form, clock gating is the addition of logic at the register's clock input path that disables the clock when the register output is not changing, as shown in Figure 7-6.

Figure 7-6 Clock Gating





The correct operation of such a circuit imposes timing restrictions, which can be relaxed if clock gating uses latches or flip-flops to eliminate hazards.

The two clock-gating styles that are widely used in designs are combinational clock gating and latch-based clock gating. They are described later in this section. Both techniques often use a single AND or a single OR gate to eliminate unwanted transitions on the clock signal.

The Formality clock-gating support covers clock gating inserted by Power Compiler. Formality verifies the clock gating inserted by other tools or manually. In general, verification of a design without clock gating against a design with clock gating results in a failure because of the extra logic in the gated design. This possibility exists for both RTL2gate and Gate2Gate verifications.

Clock gating results in the following two failing points:

- A compare point is created for the clock-gating latch. This compare point does not have a matching point in the other design, causing it to fail.
- The logic that feeds the clock input of the register bank changes. Thus, the compare points created at the register bank fail.

To verify designs with clock gating in the tool, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Choose Edit > Formality Tcl Variables.
set_app_var	The Formality Tcl Variables dialog box appears.
verification_clock_gate_hold_mode	2. From Verification, select the
<pre>[none low high any collapse_all_cg_cells]</pre>	<pre>verification_clock_gate_hold_mode variable.</pre>
	From the Choose a value list, select the desired level.
	4. Choose File > Close.

The verification_clock_gate_hold_mode variable has the following values:

- none (off) is the default.
- low to specify clock gating that holds the clock low when inactive.
- high to specify clock gating that holds the clock high when inactive.
- any to specify both the high and low styles of clock gating within the same design.
- collapse_all_cg_cells has the same effect as the low value. In addition, if the clock-gating cell is in the fanin of a register and in the fanin of a primary output port or black-box input pin, the cell is treated as a clock-gating cell in all of those logic cones.

The verification_clock_gate_hold_mode variable affects the entire design. It cannot be placed on a single instance and enabling it causes slower runtime.

When you use combinational logic to gate a clock, the Formality tool cannot detect glitches. You must use a static timing tool such as the PrimeTime tool to detect glitches.

Combinational Gate Clocking

Assume the reference design in Figure 7-7.

Figure 7-7 Reference Design

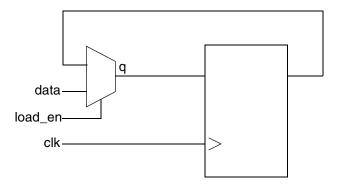
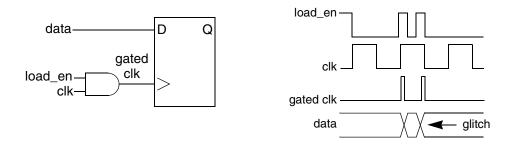


Figure 7-8 shows the typical combinational clock-gating circuitry. The gate has two inputs, enable (en) and clock (clk) the output feeds a register clock. The corresponding waveforms are also shown.

Figure 7-8 Combinational Clock Gating Using AND Gate

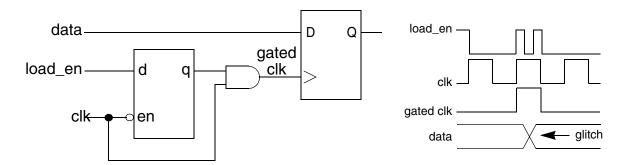


If glitches occur on the signal (load_en), invalid data is loaded into the register. Therefore, this circuit is functionally nonequivalent to that in Figure 7-7. In the default mode, the tool considers this glitch a possible input pattern and produces a failing point. The Formality tool ignores nonequivalence if you set the <code>verification_clock_gate_hold_mode</code> variable to <code>low</code>.

Latch-Based Clock Gating

The typical latch-based clock-gating circuitry, such as that used by the Power Compiler tool, is presented in Figure 7-9. The latch has two inputs, en and clk, and one output, q. The clock (clk) is gated with the output of the latch and then feeds the register clock. You can also see the corresponding waveforms.

Figure 7-9 Latch-Based Clock Gating Using AND Gate



During verification, when the <code>verification_clock_gate_hold_mode</code> variable is set, the tool recognizes clock-gating latches and takes into account their role in the design under verification.

The timing diagram shows when the load_en signal goes low, the gated clock (clk) signal also goes low. Data from the register transitions and continues to remain there until the load_en signal goes high. When you set the <code>verification_clock_gate_hold_mode</code> variable to <code>low</code>, the tool determines the setup is the same as a design that has no clock gating, as shown in Figure 7-7.

Verifying Clock-Gate Designs Automatically

The tool inserts clock edges to the registers of the next state. Using these clock edges, the tool identifies clock-gating latches and different styles of clock-gating circuitry during verification.

To enable automatic verification of the clock-gate designs,

fm_shell > set_app_var verification_clock_gate_edge_analysis true

When you set the <code>verification_clock_gate_edge_analysis</code> variable to true, the tool ignores any occurrence of the <code>verification_clock_gate_hold_mode</code> variable that might exist in the Formality Tcl scripts. You do not need to edit the scripts to remove the <code>verification_clock_gate_hold_mode</code> variable.

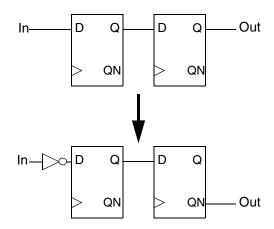
When using this feature, the tool adds annotations to the clock signals indicating their present state and next state values. These annotations are visible in the pattern viewer and logic cone schematics. You can see the following annotations when analyzing failing compare points:

Annotation	Present State	Next State
r (rising edge)	0	1
f (falling edge)	1	0
0->X	0	X
1->X	1	X
X->0	X	0
X->1	X	1

Enabling an Inversion Push

Inversion pushing means moving an inversion across register boundaries, as shown in Figure 7-10.

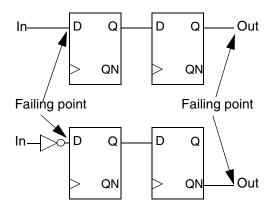
Figure 7-10 Inversion Push



Inversion pushing causes two failing points, as shown in Figure 7-11.

Formality[®] User Guide Version O-2018.06

Figure 7-11 Inversion Push Failing Points



Two techniques are available for handling inversion pushing in Formality: instance-based and environmental. The way you solve the resulting failing points differs depending on the type of inversion push.

Instance-Based Inversion Push

Instance-based inversion push specifies that a specific register has an inversion pushed across it. Formality must push an inversion across the register. This is useful when you know which register has an inverter pushed across it. This method can be applied to library cells. Apply the instance-based inversion push before verification begins. Then the next state and Q or QN pins are inverted.

To remedy the resulting failing points, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
set_inv_push	set_inv_push
[-shared_lib]	[-shared_lib]
objectID_list	objectID_list

For example,

To indicate an inversion push, you might prefer to use the set_user_match command with the -inverted or -noninverted option. This command with either option handles inverted polarity. Polarity conflicts between the set_inv_push and set_user_match commands applied to the same point are resolved using the polarity specified using the set user match command.

For more information about the set_inv_push and set_user_match commands, see the respective man pages.

Environmental Inversion Push

Each compare point matched pair has a compare polarity that is either inverted or noninverted. Inverted polarities can occur due to the style of vendor libraries, design optimizations by synthesis, or manually generated designs. If environmental inversion pushing is not enabled, Formality matches all compare points with a noninverted compare polarity unless you specify otherwise using the set_user_match -inverted command.

Environmental inversion pushing matches all state points automatically with the correct polarity. Environmental inversion pushing is off by default. Enable it only after you resolve all setup issues and ensure that differences in the designs are due to inverted state points. If there are failing compare points and environmental inversion pushing is enabled, the tool can spend a long time attempting to find a set of inverted matches to solve the verification, but this can be impossible because the compare points are not equivalent. Use this variable only if you know an inversion push was used during creation of the implementation design.

Formality can automatically use environmental inversion pushing to match state points with the correct polarity, as shown:

fm_shell	GUI
Specify:	Choose Edit > Formality Tcl Variables.
set_app_var	The Formality Tcl Variables dialog box appears.
verification_inversion_push true	2. From Verification, select the verification_inversion_push variable.
	3. Select Enable inversion push.
	4. Choose File > Close.

In the GUI, compare polarity is indicated by "+" for noninverted, "-" for inverted, and "?" for unspecified. In addition, match-related reports now have a column to indicate polarity. The "-" indicates inverted polarity, a space, "", indicates noninverted polarity. For user match reports a "?" indicates unspecified polarity.

Retimed Designs

Retiming a design moves registers across combinational logic to meet timing or area requirements. Retiming can occur during synthesis or it can be a result of "hand editing" a design. Retiming can change the number of registers in a design and the logic driving the registers.

If the implementation design has been retimed but the reference design has not been retimed, the register compare points cannot be matched. In this case, setup is required to prepare Formality to match and verify the design. If the design has been retimed in Design Compiler, then you can use the SVF file in Formality to handle retiming automatically. If the design has been retimed with another method, you can set a parameter to instruct Formality to take into account the design changes caused by retiming.

Retiming Using Design Compiler

Use the <code>optimize_registers</code> command from Design Compiler to write retiming guidance commands into the SVF file. If you do not specify an SVF file with the <code>set_svf</code> command, a file named default.svf is automatically created.

The retiming guidance commands represent the logic through which the retimed registers have moved. When the svf_retiming variable is enabled (the default), Formality adds logic to both the reference and implementation designs that represents the retiming moves. Additionally, Formality adds black boxes that create cutpoints that allow the designs to be properly matched for verification. The additional logic and black boxes enable Formality to verify both the validity of retiming and the equivalence of the designs.

The retiming guidance commands are as follows:

- guide_retiming
- guide_retiming_decompose
- guide_retiming_multibit
- guide_retiming_finished

To retime a design using Design Compiler, implement the following steps:

- 1. From Design Compiler, use the set_svf command to indicate the SVF file where the retiming guidance commands are written.
 - If the set_svf command is not used, a file named default.svf is automatically created.
- 2. Read into Design Compiler the design you want retimed.
- 3. Apply the appropriate timing constraints and implement the compile strategy.
- 4. Use the optimize_registers command from Design Compiler to retime the design.
- 5. After optimization is complete, write out the retimed netlist.
 - Multiple optimize_registers commands can be accommodated by writing out an additional netlist after each command. After each netlist is written, you must issue a new set_svf command with a unique file name before running the next optimize_registers command. The additional netlists and guide files can then be

used in a verification methodology (RTL versus netlist 1, netlist 1 versus netlist 2, netlist 2 versus netlist 3, and so on).

To verify a retimed design with Formality,

1. Use the set_svf command to tell Formality the location of the SVF file that contains the retiming guidance commands.

```
fm_shell (setup)> set_name.svf
```

2. Read the design data into Formality.

Read the original design as the reference design and the retimed netlist as the implementation design.

- 3. Apply any additional setup requirements for the designs.
- 4. Perform a verification of the design.

Retiming Using Other Tools

For designs that are retimed using other tools or for pipelined DesignWare parts, specify the retiming parameters in the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	1. Click Setup > Design Parameters.
set_parameters -retimed designID	2. Click the Reference or Implementation tab.
	3. Select a library and a design.
	4. Select Design Retimed.

Note:

The pipelined DesignWare parts, DW02_mult_n_stage and DW_pipe_mult, are detected and handled automatically by Formality. Therefore, you do not need to set the retimed parameter for these parts.

Low-Power Designs

Formality verifies and supports designs that use the IEEE 1801, also known as the Unified Power Format (UPF) standard.

Formality reads UPF files that are created at each stage of the design process, allowing verification of the intermediate netlists produced by Design Compiler and IC Compiler.

In UPF verification flow, the tool verifies designs consisting of

A design source file with the UPF file

- A Design Compiler netlist with the generated UPF file
- An IC Compiler netlist with the generated UPF file
- An IC Compiler power and ground connected netlist

Special steps might be required to handle designs that contain retention registers.

Loading the UPF File

To load and use the UPF information file into Formality, set the top design in the container and issue the following command in the setup mode:

with the options explained as follows:

Option	Description
-container container_name	Applies the UPF to the named container.
-r	Applies the UPF to the reference container.
-i	Applies the UPF to the implementation container.
-scope instance_path	Sets the initial scope for the UPF to the named instance.
-version version_string	Specifies the version string for the UPF file. If the upf_version command is in the UPF file, and it does not match version_string, a warning is issued.
filename	Specifies the name of the UPF file to load.

When loading the UPF file, the tool checks and reports the low power libraries for cells that have incorrectly modeled power behavior. To report these errors, use the report_libraries -defects command. You must correct the errors before you proceed with the verification. To automatically fix some of the errors, set the hdlin_library_auto_correct variable to true. By default, this variable is set to false.

To report information about the cells that are implemented after the UPF file is loaded, use the report_upf command.

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

Controlling the Interpretation of the UPF Files

You can specify how Formality interprets the UPF files to match your custom flow. By defining how the UPF files are interpreted, you provide Formality information how the UPF files are implemented in the exact combinations that match your design flow. The UPF file implementations are, typically, either from the file headers or from the constructs.

To specify the UPF constructs that are implemented, set the upf_implementation_based_on_file_headers variable to false and then specify the upf_implemented_constructs {isolation retention power_switching} variable.

Formality interprets the UPF file as defined by the UPF file headers if you do not specify any arguments for the upf_implemented_constructs variable.

Note:

When the upf_implementation_based_on_file_headers variable is set to true (default), Formality interprets the UPF constructs based on the UPF file header and ignores the upf_implemented_constructs variable.

When the upf_implementation_based_on_file_headers variable is set to true, Formality ignores the list specified using the upf implemented constructs variable.

If the upf_implementation_based_on_file_headers variable is set to false, Formality checks for the list specified by the upf_implemented_constructs variable, if it is set in the UPF file. Formality reports an error if the list contains invalid values.

For more information about the upf_implementation_based_on_file_headers variable, see the variable man page.

You can also set the upf_implementation_based_on_file_headers and the upf_implemented_constructs variables in the Formality GUI using the variable editor. You can access the variable editor by choosing Edit > Formality Tcl Variables.

Verifying the Design With the UPF File

The Formality tool uses information from the UPF file to identify the supply nets in the design and verify the design using the UPF power state table information. By default, the <code>verification_force_upf_supplies_on</code> variable is set to <code>true</code>, and the tool verifies the state where all the UPF supplies are on by holding the supplies constant during verification. All other UPF power state table information are ignored. This allows you to find non-power related not equivalent compare points. However, this is not a complete verification of all the power states in the design, and if the power state table does not have a state where all the supplies are enabled, the verification results might include unexpected failing compare points.

Formality[®] User Guide Version O-2018.06

For a complete verification of your design in all power states, you must run verification with the <code>verification_force_upf_supplies_on</code> variable set to <code>false</code>. The Formality tool then uses the power state table information in the UPF file to verify the design using all legal combinations of power states.

Reporting Over-Constrained Supply Nets

In a UPF verification flow, some power supplies might never switch on because of over-constrained power supplies. A power supply can be over-constrained due to incorrect power states, corruption, or feedback. By default, the tool run the <code>analyze_upf</code> command before preverification begins, and if there are over-constrained supplies, the <code>analyze_upf</code> command issues errors and prevents verification from proceeding. You can control this with the <code>upf_auto_analyze</code> variable.

To manually report about UPF supplies, use the <code>analyze_upf</code> command after loading the UPF files.

The command issues a message if there are any over-constrained supply nets. Example 7-1 shows an error message, which also reports a single-point verification command to aid in debugging the faulty supply net.

Example 7-1 Error Message Issued by the analyze_upf Command

```
Formality (verify) > analyze_upf
Container: ref
-----------
Found 1 Supply Net(s) that can never be turned ON
--------------
Supply Net ref:/WORK/top/VDDA can never be 1 (ON value)
Set verification_force_upf_supplies_on to false
Use "verify -constant1 ref:/WORK/top/VDDA" to see a failing logic cone for the supply net.
```

The UPF file defines power states and port states, which are usually applied as constraints. The analyze_upf command performs the following checks for the constraints:

1. Power states that cannot be switched on. The following example shows a message that the command issues if there are constraints that can never be switched on.

```
Formality (verify) > analyze_upf
Found 1 PST Constraint Net(s) that can never be true
----------
Legal power state i:/WORK/CHIPIO/PST_1_UPF_PST in Design i:/WORK/
CHIPIO is unreachable. It can never be true.
Set verification_force_upf_supplies_on to false
Use "verify -constant1 i:/WORK/CHIPIO/PST_1_UPF_PST " to see a failing logic cone for the constraint net (power state).
```

2. Power states that are mutually exclusive. The following example shows a message that the command issues if each state is reachable but are mutually exclusive.

For more information about the analyze_upf command, see the command man page.

Merging Parallel Switch Cells

During power network synthesis, implementation tools might expand a single UPF power switch into many coarse-grained switch cells in a variety of functionally equivalent configurations. Many switches driving the same supply net affects performance during verification and makes debugging difficult.

To avoid this problem, merge parallel switch cells to reduce redundant switches, and improve matching and verification performance. When parallel switch cell merging is enabled, the tool merges equivalent switch cells in nets that are driven by multiple coarse-grained switch cells. The tool merges switch cells in the .db file format libraries, but not the Verilog switch models. After the equivalent switch cells are merged, the tool reports the affected supply nets and the number of eliminated driving switch cells in the log file.

The hdlin_merge_parallel_switches variable is set to true by default. To retain the netlist in its unmerged form, set this variable to false.

Verifying Hierarchical Designs Using Power-Aware Black Boxes

A power-aware black box contains UPF information. Power-aware black boxes enable the verification of a hierarchical design with UPF, in which the power information is incomplete. Use power-aware black boxes to verify a hierarchical design when the subblocks are incomplete and must be black boxed.

When you read the UPF file into a hierarchical design that has black boxes, Formality creates power-aware black boxes by implementing the UPF file in the black boxes. Power-aware black boxes have additional logic for the power behavior of the subdesign ports that are required for accurate verification of the top-level design.

- Read a design using the hdlin_interface_only variable to create black boxes of the subdesigns.
- Load the UPF file into the design. Formality implements the port related supply information and creates power-aware black boxes of the subdesigns.

Verifying Hierarchical Designs Using Power Models

Using Formality power models, you can perform bottom-up verification of designs, including low-power designs. The verification of low-power designs using black boxes is inaccurate when power information is not included in the black box.

To verify a low-power design using power models, synthesize and verify the subblocks independently. The subblocks must contain UPF constructs, so they do not reference or create objects outside the hierarchy level being synthesized and verified. After the subblocks are verified, create Formality power models of both the reference and the implementation designs using the <code>write_power_model</code> command. Formality power models of the verified subblocks are used when the blocks in the next hierarchical level are verified, which improves the performance and accuracy of verification.

Example 7-2 shows how to create Formality power models.

Example 7-2 Creating Formality Power Models

```
fm_shell > read_verilog -r sub.v
fm_shell > set_top -auto
fm_shell > load_upf -r sub.upf
fm_shell > read_ddc -i sub.ddc
fm_shell > set_top -auto
fm_shell > load_upf -i sub.mapped.upf
fm_shell > verify
fm_shell > write_power_model -r sub.ref
fm_shell > write_power_model -i sub.impl
```

The write_power_model command saves the Formality power model in the .fpm file format, which are used instead of the verified subblock modules when verifying the blocks at the higher hierarchy level.

To read in the power models, use the read_power_model command.

To determine when to load the UPF subblock:

- If the top-level UPF power state table references switches in the subblock that are in the Formality power model to add power states, you might need to load this UPF subblock, which was used to create the Formality power model at the block level, into the scope of the Formality power model during the top-level verification.
- If the top-level UPF does not reference any internal switch objects in the Formality power model, you do not need to load the UPF subblock into the Formality power model, because it already contains all related supply information for all the subblock ports.

Example 7-3 shows how to read Formality power models.

Example 7-3 Reading Formality Power Models

```
fm_shell > read_verilog -r top.v
fm_shell > read_power_model -r sub.ref.fpm
fm_shell > set_top -auto
fm_shell > load_upf -r top.upf
fm_shell > read_ddc -i top.ddc
fm_shell > read_power_model -i sub.impl.fpm
fm_shell > set_top -auto
fm_shell > load_upf -i top.mapped.upf
fm shell > verify
```

The read_power_model command reads the power models into a library, named FM_MLIB_0 by default. You can specify a new library or an existing library using the read_power_model command. To ensure successful linking of the power models when you set the top-level design, remove the verified subblocks from the container using the remove_design command.

For information about the write_power_model and read_power_model commands, see their respective man pages.

Golden UPF Flow

The golden UPF flow is an optional method of maintaining the UPF multivoltage power intent of the design. It uses the original "golden" UPF file throughout the synthesis, physical implementation, and verification steps, along with supplemental UPF files generated by the Design Compiler and IC Compiler tools.

Figure 7-12 compares the traditional UPF flow with the golden UPF flow.

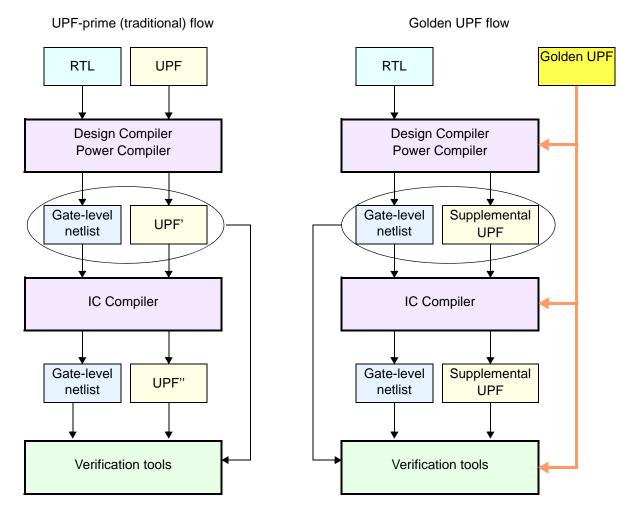


Figure 7-12 UPF-Prime (Traditional) and Golden UPF Flows

The golden UPF flow maintains and uses the same, original "golden" UPF file throughout the flow. The Design Compiler and IC Compiler tools write power intent changes into a separate "supplemental" UPF file. Downstream tools and verification tools use a combination of the golden UPF file and the supplemental UPF file, instead of a single UPF or UPF" file.

The golden UPF flow offers the following advantages:

- The golden UPF file remains unchanged throughout the flow, which keeps the form, structure, comment lines, and wildcard naming used in the UPF file as originally written.
- You can use tool-specific conditional statements to perform different tasks in different tools. Such statements are lost in the traditional UPF-prime flow.
- Changes to the power intent are easily tracked in the supplemental UPF file.

 You can optionally use the Verilog netlist to store all PG connectivity information, making connect_supply_net commands unnecessary in the UPF files. This can significantly simplify and reduce the overall size of the UPF files.

For more information about using the golden UPF mode, see SolvNet article 1412864, "Golden UPF Flow Application Note."

Less Common Operations

The are a number of operations that are less commonly carried out as part of the setting up of the tool.

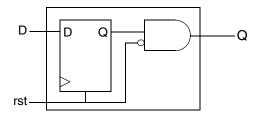
This section includes the following subsections:

- Managing Asynchronous Bypass Logic
- Asynchronous State-Holding Loops
- Re-encoded Finite State Machines
- Hierarchical Designs
- Nets With Multiple Drivers
- Retention Registers Outside Low-Power Design Flow
- Single State Holding Elements
- Multiplier Architectures
- Multibit Library Cells
- Preverification

Managing Asynchronous Bypass Logic

A sequential cell where some of the asynchronous inputs have combinational paths to the outputs, bypassing the generic sequential element SEQGEN, is said to have an asynchronous bypass, as shown in Figure 7-13.

Figure 7-13 Asynchronous Bypass Logic



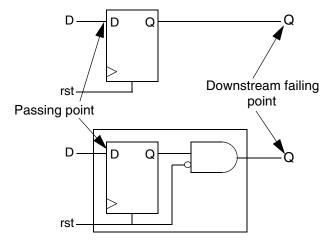
Asynchronous bypass logic can result from

- Mapping from one technology library to another.
- Verilog simulation libraries. The Verilog module instantiates logic, creating a combinational path that directly affects the output of a sequential user-defined primitive (UDP).
- Modeling a flip-flop with RTL code. The RTL has an explicit asynchronous path defined or the RTL specifies that both Q and QN have the same value when Clear and Preset are both active.

Asynchronous bypass logic cannot come from a .lib file that was converted to a .db file. Library Compiler uses a sequential element to model asynchronous behavior to avoid creating explicit bypass paths.

Asynchronous bypass logic results in a failing point, as shown in Figure 7-14.

Figure 7-14 Asynchronous Bypass Failing Point



To prevent aborting verification due to the downstream failing point, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify: 1	Choose Edit > Formality Tcl Variables.
set_app_var	The Formality Tcl Variables dialog box appears.
verification_asynch_bypass true	From Verification, select the verification_asynch_bypass variable.
	 Select Enable asynchronous bypass to set the variable to true.
	4. Choose File > Close.

This procedure creates asynchronous bypass logic around every register in the design. Setting verification_asynch_bypass to true can cause the following:

- Longer verification runtimes
- Introduction of loops into the design
- Aborted verification due to design complexity

Asynchronous bypass affects the entire design and cannot be placed on a single instance. In addition, asynchronous bypass is automatically enabled when you verify cells in a technology library; because of the relative simplicity of library cells, no negative effects occur.

Asynchronous State-Holding Loops

Formality is used to verify synchronous designs. Therefore your design should not contain asynchronous state-holding loops implemented as combinational logic. Asynchronous state-holding loops can cause some compare points to be aborted, thus giving inconclusive results.

Asynchronous state-holding loops affect Formality in the following ways:

- If Formality establishes that an asynchronous state-holding loop affects a compare point, it aborts that compare point, and that point is not proven equivalent or nonequivalent.
- If Formality establishes that an asynchronous state-holding loop has a path that does not affect a compare point, it proves that point equivalent or nonequivalent.
- If Formality cannot establish that an asynchronous state-holding loop has a path that
 does not affect a compare point, it aborts that compare point, and that point is not proven
 equivalent or nonequivalent.

Formality automatically breaks loops during verification if they are identical. To change this behavior, set the <code>verification_auto_loop_break</code> variable to <code>false</code>. For information about this variable, see the man page.

Note:

You can also specify the report_loops command after verification. In this case, Formality reports the original loops even if they were automatically broken during verification.

To report asynchronous state-holding loops, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>report_loops [-ref] [-impl] [-limit N] [-unfold]</pre>	<pre>report_loops [-ref] [-impl] [-limit N] [-unfold]</pre>

By default, the report_loops command returns a list of nets and pins for loops in both the reference and implementation designs. It reports 10 loops per design and 100 design objects per loop unless you specify otherwise with the <code>-limit</code> option. Objects are reported using instance-based path names.

Use the -unfold option to report subloops embedded within a loop individually. Otherwise, they are reported together.

If a loop is completely contained in a technology library cell, this command lists all the nets and pins associated with it. If only part of a loop belongs to a technology library cell, the cell name does not appear in the list. In addition, the report displays the hierarchical structure if a loop crosses boundaries.

For more information about the report_loops command, see the man page.

After you determine the locations of any asynchronous state-holding loops, ensure that Formality successfully verifies the loop circuit by inserting cutpoints.

Re-encoded Finite State Machines

The architecture for a FSM consists of a set of flip-flops for holding the state vector and a combinational logic network that produces the next state vector and the output vector. For more information about finite state machines, see the Design Compiler documentation.

Before verifying a re-encoded FSM in the implementation design against its counterpart in the reference design, you must take steps that allow Formality to make verification possible. These steps define the FSM state vectors and establish state names with their respective encoding.

Without the proper setup, Formality is unable to verify two FSMs that have different encoding, even if they have the same sequence of states and output vectors.

Formality provides several methods to name flip-flops and define encoding. User-defined encoding is not verified by Formality, so take care to specify the encoding correctly. The easiest method is to use the SVF file generated by Design Compiler. You can also use a single fm_shell command to read a user-supplied file that contains all the information simultaneously, or you can use two commands to first name state vector flip-flops and then define the state names and their encoding. These methods are described in the following sections.

SVF file for FSM Re-encoding

The SVF file generated by Design Compiler contains FSM state vector encoding. This encoding is in the form of <code>guide_fsm_reencoding</code> commands. Use the following variable to tell Formality to use the FSM guidance in the Design Compiler SVF file:

```
set_app_var svf_ignore_unqualified_fsm_information false
```

Set this variable before reading the SVF file. For more information, see "Creating an SVF File" on page 5-4. You can also manually perform the guide fsm reencoding commands.

Reading a User-Supplied FSM State File

To name the FSM state vector flip-flops and provide state names with their encoding simultaneously.

fm_shell	GUI
Specify:	At the Formality prompt, specify:
read_fsm_states filename	read_fsm_states filename
[designID]	[designID]

Use this method when your FSM has many states. If your FSM has only a few states, consider the method described in the following section.

Note:

You must supply FSM information for both the reference and implementation designs for verification to succeed.

The file you supply must conform to certain syntax rules. You can generate a suitable file by using the report_fsm command in Design Compiler and redirecting the report output to a file. For information about the file format and the read_fsm_states command, see the man page.

Formality[®] User Guide Version O-2018.06

Defining FSM States Individually

To name a FSM state vector flip-flop first and then define the state name and its respective encoding,

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>set_fsm_state_vector flip-flop_list [designID]</pre>	<pre>set_fsm_state_vector flip-flop_list [designID]</pre>
Then specify:	Then specify:
set_fsm_encoding encoding_list	set_fsm_encoding encoding_list
[designID]	[designID]

Using these commands can be convenient when you have just a few flip-flops in the FSMs that store states. You must use the commands in the order shown.

Note:

You must supply FSM information for both the reference and implementation designs for verification to succeed.

The first command names the flip-flops, and the second command defines the state names with their encoding.

Multiple Re-encoded FSMs in a Single Module

Formality supports multiple re-encoded FSMs in a single module. FSM re-encoding occurs during synthesis, different state registers exist due to different state-encoded machines in the implementation and reference designs. Formality supports these re-encoded FSMs if you provide both the FSM state vector and the state encoding either by using the <code>-name</code> option with the <code>set_fsm_state_vector</code> and <code>set_fsm_encoding</code> commands, or by using the <code>read_fsm_states</code> command with the FSM information provided in a file you specify.

For example,

```
set_fsm_state_vector {ff1 ff2} -name fsm1 set_fsm_encoding {s1=2\#01 s2=2\#10} -name fsm1 set_fsm_state_vector {ff3 ff4} -name fsm2 set_fsm_encoding {s1=2\#01 s2=2\#10 s3=2\#11} -name fsm2
```

When verifying FSM re-encoded designs, Formality

 Modifies the reference design by replacing the original state registers with the new state registers Synthesizes the logic around the new state registers to keep the new reference design functionally equivalent to its original

Formality verifies the FSM re-encoded designs because the new reference and implementation designs have the same state registers.

Listing State Encoding Information

To list FSM state information for a particular design, use either the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
report_fsm [designID]	report_fsm [designID]

Formality produces a list of FSM state vector flip-flops and their encoding.

FSMs Re-encoded in Design Compiler

If you are verifying a design with a FSM that has been re-encoded in Design Compiler, supply the state register mapping and state encoding to Formality first, before matching. If the FSMs are present but the encoding has not been changed, setup information is not required.

Several methods are available for addressing FSM setup in Formality if you used Design Compiler to do the re-encoding. These methods are listed in order of preference.

- Write an SVF file (.svf) from Design Compiler, then read the file into Formality.
- Use the fsm_export_formality_state_info command in Design Compiler to write out the module_name.ref and module_name.impl files, then read these files back into Formality using the read_fsm_states command.
- Use the report_fsm command in Design Compiler for both the reference and implementation designs, then read these reports back into Formality using the read_fsm_states command.

Alternatively, if you manually re-encode your design, or if the re-encoding is completed by a tool other than Design Compiler, use the set_fsm_encoding and set_fsm_state_vector commands in Formality for both the reference and implementation designs to specify the state encoding and register state mapping.

Hierarchical Designs

You can control the following two features of hierarchical design verification: the separator character used to create flattened path names and the operating mode for propagating constants throughout hierarchical levels.

Setting the Flattened Hierarchy Separator Character

Formality uses hierarchical information to simplify the verification process, but it verifies designs in a flat context. By default, Formality uses the slash (/) character as the separator in flattened design path names. If this separator character is not consistent with your naming scheme, you can change it.

To establish a character as the flattened path name separator, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Choose Edit > Formality Tcl Variables.
set_app_var	The Formality Tcl Variables dialog box appears.
ame_match_flattened_hierarchy separator_style character	2. From Matching, select the name_match_flattened_hierarchy_separator_style variable.
	3. In the Enter a single character box, enter the character separator used in path names when designs are flattened and press Enter.
	4. Choose File > Close.

The name_match_flattened_hierarchy_separator_style variable reads in the design hierarchy, and the character separator specifies the hierarchical boundaries.

Propagating Constants

When Formality verifies a design that contains hierarchy, the default behavior is to propagate all constants throughout the hierarchy. For a description of constant types as they apply to Formality, see "Specifying Constants" on page 7-8.

In some cases, you might not want to propagate all constants during hierarchical verification. To determine how Formality propagates constants, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>set_app_var verification_constant_prop_mode mode</pre>	<pre>set_app_var verification_constant_prop_mode mode</pre>

You can use the <code>verification_constant_prop_mode</code> variable to specify where Formality is to start propagation during verification. In <code>auto</code> mode, the default, Formality traverses up the reference and implementation hierarchy in lockstep to identify automatically the top design from which to propagate constants. Therefore, correspondence between the hierarchy of the two designs affects this mode. Specify <code>top</code> to tell Formality to propagate from the design you set as top with the <code>set_top</code> command. Specify <code>target</code> to instruct Formality to propagate constants from the currently set reference and implementation designs.

Set the <code>verification_constant_prop_mode</code> variable to <code>top</code> or <code>target</code> only if your reference and implementation designs do not have matching hierarchy. Setting the mode to <code>auto</code> when you have different levels of hierarchy can cause Formality to propagate from an incorrect top-level design.

For more information about this variable, see the man page.

Nets With Multiple Drivers

During verification, Formality ensures that each net with more than one driver is resolved to the correct function. At the design level, you can use resolution functions to resolve these types of nets. To define net resolution, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Setup > Design Parameters.
	2. Click the Reference or Implementation tab.
set_parameters	3. Select a library and a design.
[-resolution function] designID	4. Select Consensus, Black Box, Wired AND, or Wired OR.

The -resolution function option defines the behavior of nets that have more than one driver. Formality provides a choice of four resolution functions: consensus, black box, AND, and OR. Not all options of the set parameters command are shown.

With the consensus resolution function, Formality resolves each net in the same manner as a four-state simulator. Each driver can have any of four output values: 0, 1, X (unknown), or Z (high-impedance state). Formality uses this function by default.

Table 7-1 shows the net resolution results for a net with two drivers. The top row and left column show the possible driver values, and the table entries show the resulting net resolution results.

Table 7-1 Consensus Resolution for a Net With Two Drivers

	0	1	Х	Z
0	0	Х	Х	0
1	X	1	X	1
Χ	x	X	Х	Χ
Z	0	1	Χ	Z

The consensus resolution function works similarly for nets with more than two drivers. If all drivers on the net have the same output value, the result is the common value. If any two active (non Z) drivers are in conflict, the result is X.

With the AND resolution function, the result is the logical AND of all active (non Z) drivers on the net. Similarly, with the OR resolution function, the result is the logical OR of all active drivers on the net.

Note:

If you want to use AND or OR resolution types, your designs must support wired AND and wired OR functionality. Do not use these resolution types with CMOS technology.

With the black box resolution function, Formality creates a black box for each net with multiple drivers. It connects the net to the output of the black box, connects the net drivers to the inputs of the black box, and makes the net a compare point. The inputs to the black box are treated just like the inputs to any other compare point. In other words, to pass verification, the inputs need to be matched between the two designs and the logic cones feeding these inputs need to be equivalent.

If you do not specify how to resolve nets having more than one driver, Formality looks at the types of drivers on the net. If none of the drivers are primary input ports or black box outputs, Formality uses the consensus resolution function. However, if any driver is a primary input port or the output of a black box, Formality cannot determine the value of that driver. In that case, Formality inserts a black box function at that point, driven by the primary input port or by the existing black box, and uses the consensus resolution function to combine the output of the inserted black box function with any other drivers on the net.

Using the consensus function causes Formality to resolve the value of the net according to a set of consensus rules. For information about these rules, see the set_parameters man page.

In Figure 7-15, a single net is driven by two three-state devices, an inverter, and a black box component. By default, Formality attempts to use the consensus resolution function to resolve the net at the shaded area. In this case, one of the drivers comes from a black box component. Because Formality cannot determine the state of a driver that originates from a black box component or an input port, it cannot use the consensus resolution.

Figure 7-15 Default Resolution Function: Part One

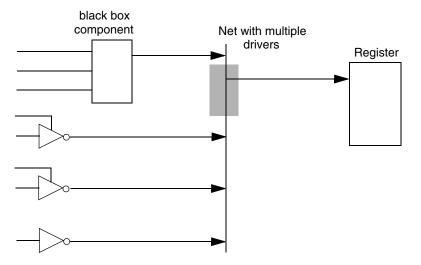


Figure 7-16 shows how Formality resolves the net in this case. The three drivers at the bottom of the circuit can be resolved by the consensus function. That function in turn drives a black box resolution function that ultimately drives the register.

Black box component

Net with multiple drivers

Register

Black box resolution

Consensus resolution

Figure 7-16 Default Resolution Function: Part Two

Retention Registers Outside Low-Power Design Flow

Formality supports the verification of designs with retention registers. For information about retention registers, see the *Power Compiler User Guide*. To verify a netlist with retention registers against RTL code without retention registers, you must disable all retention registers' sleep modes. To disable their sleep mode, set a constant on the sleep pins on the retention registers.

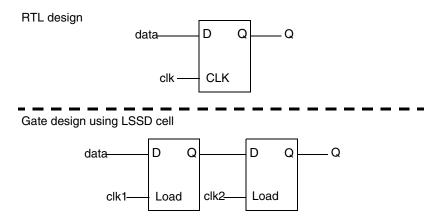
Formality reads design information describing retention registers from RTL, technology libraries, and implementation netlists produced by Power Compiler. During compare point matching, Formality checks retention registers in the reference design against matching registers in the implementation design for the power gating style attribute.

Single State Holding Elements

A level-sensitive scan design (LSSD) cell is a single-state holding element that consists of two latches arranged in a master-slave configuration. LSSD cells occur frequently when you use IBM libraries.

LSSD cells result in two compare points in the gate-level design, as shown in Figure 7-17. The RTL design contains a SEQGEN that results in one compare point. The dotted line separates the reference design from the implementation design.

Figure 7-17 LSSD Cells



Two criteria must be met in order for Formality to determine that a latch is part of an LSSD cell:

- The latch pair must reside within a single technology library cell.
- The latches must be matched to a flip-flop using a name-based solution, such as the exact name, fuzzy name match, rename_object, or compare rule. Signature analysis cannot be used.

The two latches can be verified against a single sequential element if they meet the LSSD cell criteria.

Multiplier Architectures

Formality uses the arithmetic generator feature automatically to improve the performance and ability to solve designs where multipliers have been flattened into gate-level netlists. Use of the arithmetic generator in Formality creates multipliers of a specific type so that the synthesized representation of the reference RTL more closely matches the gate implementation. Therefore, assisting in the verification of difficult datapath problems.

The arithmetic generator can create the following multiplier architectures:

- Carry-save array (csa)
- Non-Booth Wallace tree (nbw)
- Booth-encoded Wallace tree (wall)

Setting the Multiplier Architecture

You can set the multiplier architecture either for your entire design or on particular instances of cells in your design. The following sections describe both methods for setting the multiplier architecture.

Setting the Multiplier Architecture on an Entire Design

You can manually instruct Formality to use a specific multiplier architecture for your entire design file by using your RTL source and the hdlin_multiplier_architecture and enable_multiplier_architecture Tcl variables.

To instruct Formality to use a specific multiplier architecture for a specific design file, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
set_app_var hdlin_multiplier_architecture csa	set_app_var hdlin_multiplier_architecture csa
<pre>set_app_var enable_multiplier_generation true</pre>	<pre>set_app_var enable_multiplier_generation true</pre>
read_verilog myfile.v	read_verilog myfile.v

By default, the hdlin_multiplier_architecture variable is set to none. The arithmetic generator attempts to duplicate the architecture Design Compiler used in determining which architecture is appropriate. Formality uses the value defined in the dw_foundation_threshold Tcl variable to help select the architecture. If you do not want Formality to determine the architecture, set the value of the hdlin multiplier architecture variable to your preferred architecture.

For more information about the hdlin_multiplier_architecture and dw_foundation_threshold variables, see the man pages.

Note:

You also have the choice of setting the multiplier architecture by using the architecture_selection_precedence Tcl variable. With this variable you can define which mechanism takes precedence.

Setting the Multiplier Architecture on a Specific Cell Instance

You can replace the architecture for a specific multiplier ObjectID. While you are in setup mode and after elaboration, use the <code>enable_multiplier_generation</code> variable and the <code>set_architecture</code> command with the specific cell ObjectID and specific architecture to set the desired multiplier architecture.

Formality[®] User Guide Version O-2018.06

To instruct Formality to use a specific multiplier architecture for a specific ObjectID, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>set_app_var enable_multiplier_generation true</pre>	<pre>set_app_var enable_multiplier_generation true</pre>
<pre>set_architecture ObjectID [csa nbw wall]</pre>	<pre>set_architecture ObjectID [csa nbw wall]</pre>

For more information about the enable_multiplier_generation variable and the set_architecture command, see the man pages.

An alternative to setting the multiplier architecture while in setup mode is to set a compiler directive in your VHDL or Verilog source code that sets the multiplier architecture for a specific cell instance. The following section explains how to do this.

Setting the Multiplier Architecture by Using Compiler Directives

You can use a compiler directive to set the multiplier architecture by annotating your RTL source code with the architecture desired for a given instance. This compiler directive is a constant in the RTL source that appears immediately before the multiplier instance when you set

```
formality multiplier [csa | nbw | wall ]
```

When present in a comment, the compiler directive causes Formality to use the specified architecture to synthesize the next multiplier instance in the RTL source. If multiple compiler directives are present before a single multiplier instance, the arithmetic generator builds the architecture with the compiler directive preceding it.

The compiler directive can be in Verilog or VHDL source. The following shows an example of each type:

Verilog

```
// formality multiplier nbw
z <= a*b;

VHDL

-- formality multiplier nbw
z <= a*b;</pre>
```

In both instances, this compiler directive informs the arithmetic generator to use a non Booth Wallace tree architecture (nbw) for the "a * b" multiplier instance.

Reporting Your Multiplier Architecture

To report the architecture used to implement a specific ObjectID, use the report_architecture command.

To report on the multiplier architecture used in your design, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
report_architecture -all	report_architecture -all

For more information about the report_architecture command and its options, see the man page.

Multibit Library Cells

Formality supports the use of multibit library cells. You can control multibit component inference in Design Compiler by using the hdlin_infer_multibit variable. For more information, see the man page on the hdlin_infer_multibit variable in Design Compiler. If you choose not to use this capability in Design Compiler, and you manually group register bits into library cells instead, then you need to follow certain naming rules. Otherwise, Formality can encounter difficulties in matching compare points where the multibit components are used.

The following naming rules apply for manually grouping register bits into library cells:

- When you group registers into multibit cells, use the syntax name_number to number to name the grouped cell. For example, the name my_reg_7to0 maps to the eight registers named my_reg_0, my_reg_1, ... my_reg_7 in the other design.
- If the grouped register contains multiple elements that are not in sequential order, you can use syntax in the form of name_number to number, number, number... For example, the name treg_6to4,2 maps to the four registers named treg_6, treg_5, treg_4, and treg_2 in the other design. In this syntax, a comma separates the individual elements of the multibit cell.

Preverification

Setup commands are inherently instance-based. In preverify mode, you can access the final instance objects during setup. The final instance objects are the instance objects of a design on which modifications such as UPF, SVF, and ECO are applied. Only setup operations that do not modify the design database can be performed in preverify mode. Commands that change the design database are not allowed and the tool issues an error message if these commands are used.

In preverify mode, you can

- Process the UPF and SVF files
- Apply setup commands on post SVF and post UPF object names
- Use design object query commands both before and after running the preverify command

When the tool starts, it starts in setup mode in which you can load SVF files, load design files, elaborate designs using the <code>set_top</code> command, load and execute UPF files, remove containers, and perform ECO edits.

To enter preverify mode, use the <code>preverify</code> command. The tool enters preverify mode and discards existing match and verify results. In preverify mode, you can run setup commands that do not modify the design database such as setup operations on post SVF modified design objects. In setup mode, the <code>match</code> and <code>verify</code> commands automatically run the <code>preverify</code> command. In preverify, match, or verify modes, the <code>preverify</code> command removes views, reprocesses the SVF file, and creates new views.

The following commands are not available in preverify mode. The other setup commands are available in the preverify mode.

change_link ¹	read_ddc	remove_parameters
commit_edits	read_fsm_states	remove_port ¹
connect_pin ¹	read_milkyway	remove_resistive_drivers
connect_net1	read_power_model	rename_object
create_cell ¹	read_sverilog	rewire_connection
create_container	read_verilog	set_architecture
create_cutpoint_blackbox	read_vhdl	set_clock
create_net	remove_cell ¹	set_direction
create_port ¹	remove_clock	set_equivalence
create_primitive ¹	remove_constraint	set_fsm_encoding
define_design_lib	remove_constraint_type	set_fsm_state_vector
define_primitive_pg_pins	remove_container	set_implementation_design
disconnect_net1	remove_design	set_inv_push
elaborate_library_cells	remove_design_library	set_parameters
group	remove_equivalence	set_power_gating_style
insert_inversion	remove_inv_push	set_reference_design
invert_pin	remove_inversion	set_svf
load_upf	remove_library	set_svf_retiming
read_container	remove_net ¹	set_top
read_db	remove_object	set_vsdc

^{1.} The edit commands are available in the preverify, match, and verify modes only for designs created with the edit_design command.

8

Matching Compare Points

After you have prepared your verification environment and set up your design, you are ready to match compare points.

This chapter includes the following sections:

- Introduction
- Basic Usage
- Advanced Usage

Figure 8-1 outlines the placing of compare point matching in the Formality design verification process flow. This chapter focuses on matching compare points in Formality.

Start Formality Load Guidance Load Reference Load Implementation Perform Setup Match Compare Points Run Verify Interpret Results No Success? Debug Yes Done

Figure 8-1 Compare Point Matching in the Design Verification Process Flow

Introduction

Prior to verification, Formality must match compare points in the designs as described in "Matching" on page 1-7. This matching occurs automatically when you specify the <code>verify</code> command. If automatic matching results in unmatched points, you must then view and troubleshoot the results. Unmatched compare points can result in nonequivalence of the two designs.

You can match compare points in a separate step before verification by running the match command. Consequently, you can iteratively debug unmatched compare points, as follows:

- 1. Perform compare point matching.
- 2. Report unmatched points.
- 3. Modify or undo results of the match, as needed.
- 4. Debug the unmatched compare points.
- 5. Repeat these steps incrementally, as needed, until all compare points are matched.

Performing compare point matching changes the operational mode from setup to match even if matching was incomplete. Ensure that you have properly set up your design as specified (see Chapter 7, "Performing Setup").

You can return to setup mode by using the setup command, but this causes all points matched during match mode to become unmatched.

Basic Usage

At its most basic, the steps involved in compare point matching are as follows:

- Performing Compare Point Matching
- Reporting Unmatched Points

Performing Compare Point Matching

To match compare points, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	1. Click the Match tab.
match	2. Click Run Matching.

This command matches only unmatched points. Previously matched points are not processed again. Prior to compare point matching, you can create compare rules. For more information, see "Matching With Compare Rules" on page 10-14.

The matching results from incremental matching can differ from those you receive when you run the <code>match</code> command after fixing all setup problems. For example, suppose your last setup change implements a compare rule that helps match the last remaining unmatched points. This same rule can force incorrect matches or prevent matches if you had implemented it at the beginning of the matching process.

You can interrupt matching by pressing Ctrl+C. All matched points from the interrupted run remain matched.

To return to setup mode, specify the <code>setup</code> command in the Formality shell or at the Formality prompt within the GUI. You can use commands and variables disabled in the matched state. This command does not remove any compare rules or user matches. Use the <code>remove_compare_rules</code> command and the <code>remove_user_match</code> command to get rid of those previously set values. Existing compare rules and user matches are used again during the next match.

Reporting Unmatched Points

An unmatched point is a compare point in one design that was not matched to a corresponding point in the other design. You must match all compare points before a verification succeeds unless the unmatched compare points do not affect downstream logic. After each match iteration, examine the results to see which compare points remain unmatched.

To report unmatched points, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Match > Unmatched Points
report_unmatched_points	
[-compare_rule] [-datapath]	
[-substring string]	
[-point_type point_type]	
[-status status]	
[-except_status status]	
[-method matching_method]	
[-last]	
[[-type ID_type] compare_point]	

This command reports compare points, input points, and higher-level matchable objects that are unmatched. Use the options to filter the report as desired.

Note that the same can be done for matched points by executing the report_matched_points command or (in the GUI) clicking Match > Matched. This report shows matched design objects (such as inputs) as well as matched compare points. You can specify a filter to report only the matched compare points or (in the GUI) click Match > Summary.

Advanced Usage

At a more advanced level, the steps involved in compare point matching are as follows:

- Debugging Unmatched Points
- Undo Matched Points
- How Formality Matches Compare Points

Debugging Unmatched Points

Unmatched compare points are often caused by design changes during Design Compiler optimization. The intent of these changes is to optimize the design for speed or by area, or to prepare the design for back-end tools. Unfortunately, such design changes might cause compare point matching problems because the object names often change significantly.

Common design changes include moving features up and down the design hierarchy, explicitly applying name rules to objects in the design, and eliminating constant registers.

Note:

In Verilog and VHDL files, unmatched compare points can be caused by a difference between the bus naming scheme and the default naming conventions.

If the number of unmatched points in the reference and implementation designs is the same, the likely cause is an object name change.

If the number of unmatched points in the reference and implementation designs is different, you might need to perform additional setup steps. For example,

- You might have a black box in one design but not in the other.
- An extra compare point in the implementation design can be caused by a design transformation that created extra logic.
- An extra compare point in the reference design can be a result of ignoring a full_case directive in the RTL code.

Table 8-1 shows the actions you can take for unmatched compare points.

Table 8-1 Unmatched Compare Points Action

Symptom	Possible cause	Action
Same number of unmatched points in reference and implementation designs	Names have undergone a transformation	Use set_user_match command Write and test compare rule Modify name match variables Turn on signature analysis For all, see "Reporting Unmatched Points" on page 8-5
More unmatched points in reference than in implementation design	Unused cells	No action necessary
	full_case directive in RTL code ignored	Set hdlin_ignore_full_case to false
	Black box created for missing cells	Reread reference design, including the missing cells
		Make black box in implementation design
More unmatched points in the implementation design than in the reference design	Design transformation created extra logic	Account for design transformation; see "Design Transformations" on page 10-19
	Black box created for missing cells	Reread reference design, including the missing cells
		Reread reference design, includi

Undo Matched Points

To undo the results of the match command, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
undo_match [-all]	undo_match [-all]

This command is especially useful when you have made changes that did not achieve the results you desired for compare point matching. It returns all points matched during the most recent match command back to their unmatched state. Use the -all option to undo all matches.

You remain in the matched state even if you undo the first match command or specify the -all option.

To return to the setup state, specify the setup command in fm_shell or in the GUI choose the Setup button.

How Formality Matches Compare Points

As described in "Concept of Name-Based and Non Name-Based Matching" on page 1-8, compare point matching is either named-based or not.

The following matching techniques occur by default when you match compare points, and they are executed in this given order:

- 1. Exact-name matching (name-based matching)
- 2. Name filtering (name-based matching)
- 3. Topological equivalence (non name-based matching)
- 4. Signature analysis (non name-based matching)
- 5. Compare point matching based on net names (name-based matching)

After a technique succeeds in matching a compare point in one design to a compare point in the other design, that compare point becomes exempt from processing by other matching techniques.

The following sections describe each default compare point matching technique.

Formality[®] User Guide Version O-2018.06

Table 8-2 lists variables that control matching. Some are described in the following sections.

Table 8-2 Variables for Compare Point Matching

Variable Name	Default
name_match	all
name_match_allow_subset_match	strict
name_match_based_on_nets	true
name_match_filter_chars	`~!@#\$%^&*()_+= \{}[]":;<>?,./
name_match_flattened_hierarchy_separator_style	/
name_match_multibit_register_reverse_order	false
name_match_use_filter	true
signature_analysis_match_primary_input	true
signature_analysis_match_primary_output	false
signature_analysis_match_compare_points	true
verification_blackbox_match_mode	any

Exact-Name Matching

Formality matches unmatched compare points by exact case-sensitive name matching, and then by exact case-insensitive name matching. The exact-name matching technique is used by default in every verification. With this algorithm, Formality matches all compare points that have the same name both in reference and implementation designs.

For example, the following design objects are matched automatically by the Formality exact-name matching technique:

```
Reference: /WORK/top/memreg(56)
Implementation: /WORK/top/MemReg(56)
```

To control whether compare point matching uses object names or relies solely on function and topology to match compare points, specify the name_match variable as shown:

fm_shell	GUI
Specify:	1. Click Match.
<pre>set_app_var name_match [all none port cell]</pre>	 Choose Edit > Formality Tcl Variables. The Formality Tcl Variables dialog box appears.
	3. From Matching, select the name_match variable.
	4. In the Choose a value list, select all, none, port, or cell.
	5. Choose File > Close.

The default all, performs all types of name-based matching. Use none, to disable all name-based matching except for the primary inputs. Use port, to enable name-based matching of top-level output ports. Use cell, to enable name-based matching of registers and other cells, including black box input and output pins.

Name Filtering

After exact-name matching, Formality attempts filtered case-insensitive name matching. Compare points are matched by filtering out some characters in the object names.

To turn off the default filtered-name matching behavior, use the Formality shell or the GUI as shown:

1. Click Match.
Choose Edit > Formality Tcl Variables.The Formality Tcl Variables dialog box appears.
From Matching, select the name_match_use_filter variable.
4. Deselect Use name matching filter.
5. Choose File > Close.

The name_match_use_filter variable is supported by the name_match_filter_chars variable that lists all the characters that are replaced by an underscore (_) character during the name-matching process.

Filtered name matching requires that any nonterminating sequence of one or more filtered characters in a name must be matched by a sequence of one or more filtered characters in the matched name.

For example, the following design object pairs are matched automatically by the Formality name-filtering algorithms:

```
Reference: /WORK/top/memreg__[56][1]
Implementation: /WORK/top/MemReg_56_1
Reference: /WORK/top/BUS/A[0]
Implementation: /WORK/top/bus__a_0
```

The following design objects are not matched by the Formality name-filtering algorithms:

```
Reference: /WORK/top/BUS/A[0]
Implementation: /WORK/top/busa_0
```

You can remove or append characters in the <code>name_match_filter_chars</code> variable. The default character list is:

```
`~!@#$%^&*()_-+=|\[]{}"":;<>?,./
```

For example, the following command resets the filter characters list to include v:

Reversing the Bit Order in Multibit Registers

You can use the name_match_multibit_register_reverse_order variable to reverse the bit order of the bits of multibit registers during compare point matching. The default is false, meaning that the order of the bits of multibit registers is not reversed. Formality automatically matches multibit registers to their corresponding single-bit counterparts, based on their name and bit order. If the bit order has been changed after synthesis, you must set this variable to true, so that the order of the bits of multibit registers is reversed. For more information about Formality multibit support, see "Multibit Library Cells" on page 7-50. In the GUI, you can access this variable from the Formality Tcl Variable Editor dialog box by choosing Edit > Formality Tcl Variables, and then from Matching, select the variable.

Topological Equivalence

Formality attempts to match the remaining unmatched compare points by topological equivalence — that is, if the cones of logic driving two unmatched compare points are topologically equivalent, those compare points are matched.

Signature Analysis

Signature analysis is an iterative analysis of the compare points' functional and topological signatures. Functional signatures are derived from random pattern simulation; topological signatures are derived from fanin cone topology.

The signature analysis algorithm uses simulation to produce output data patterns, or signatures, of output values at registers. The simulation process in signature analysis is used to identify uniquely a controlled node.

For example, if a vector makes a register pair go to a 1 and all other controlled registers go to a 0 in both designs, signature analysis has completed one match.

For signature analysis to work, the primary input ports from both designs must have matching names or you must have manually matched them by using the set_user_match, set_compare_rule, or rename_object commands.

During signature analysis, Formality also automatically attempts to match previously unmatched datapath and hierarchical blocks and their pins. To turn off automatic matching

of datapath blocks and pins, set the signature_analysis_match_datapath variable to false. To turn off automatic matching of hierarchical blocks and pins, set the signature_analysis_match_hierarchy variable to false. For the latter case, if you notice a performance decrease when running hierarchical verification, you can change the setting of signature_analysis_match_hierarchy to false.

Signature analysis in Formality works well if the number of unmatched objects is limited, but the algorithm is less likely to work if there are thousands of compare point mismatches. To save time in such a case, you can turn off the algorithm in the Formality shell or the GUI, as shown in the following table.

fm_shell	GUI
Specify:	1. Click Match.
<pre>set_app_var signature_analysis_match_ compare_points false</pre>	 Choose Edit > Formality Tcl Variables. The Formality Tcl Variables dialog box appears.
	From Matching, select the signature_analysis_match_compare_points variable.
	4. Deselect Use signature analysis.
	5. Choose File > Close.

By default, signature analysis does not try to match primary output ports. However, you can specify the matching of primary outputs by setting the

 $\verb|signature_analysis_match_primary_output| \ensuremath{\textit{Variable to}} \ensuremath{\textit{to}} \ensuremath{\textit{true}}.$

It is possible to reduce matching runtimes by writing a compare rule rather than disabling signature analysis. For example, compare rules work well if there are extra registers in both the reference and implementation designs. For more information, see "Matching With Compare Rules" on page 10-14.

Note:

The tool uses signature analysis to match black boxes with different names. After the black boxes are matched, the tool first attempts to match the black box pins by name. If the black box pin names are similar, the pins are matched. If the pin names are different, then the tool uses signature analysis again to match the pins functionally.

Compare Point Matching Based on Net Names

Formality matches any remaining unmatched compare points by exact and filtered matching on their attached nets. Matches can be made through either directly attached driven or driving nets.

To turn off net name-based compare point matching, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	1. Click Match.
set_app_var name_match_based_on_nets false	Choose Edit > Formality Tcl Variables.The Formality Tcl Variables dialog box appears.
	 From Matching, select the name_match_based_on_nets variable.
	4. Deselect Use net names.
	5. Choose File > Close.

For example, the following design objects have different names.

Reference: /WORK/top/memreg(56)
Implementation: /WORK/top/MR(56)

Formality cannot match them by using the exact-name matching technique. If nets driven by output of these registers have the same name, Formality matches the registers successfully.

Commands and Variables That Cannot be Changed in Match Mode

The following commands and variables cannot be changed in the matched state:

set_cutpoint	set_fsm_encoding
remove_black_box	set_fsm_state_vector
remove_constant	set_inv_push
remove_cutpoint	set_parameters -resolution -retimed
remove_design	ungroup
remove_inv_push	uniquify
remove_object	verification_assume_reg_init
remove_parameters -resolution -retimed -all_parameters	verification_auto_loop_break
remove_resistive_drivers	verification_clock_gate_hold_mode
rename_object	verification_constant_prop_mode

set_cutpoint set_fsm_encoding

set_black_box verification_inversion_push

set_constant verification_merge_duplicated_registers

set_direction verification_set_undriven_signals

9

Verifying the Design and Interpreting Results

After you have matched your compare points, you are ready to verify the design and interpret the results. This chapter describes how to verify one design against another. It also offers some tips for batch verifications, interpreting results, and saving data.

This chapter includes the following sections:

- Introduction
- Basic Usage
- Advanced Usage

Figure 9-1 outlines the placing of run verification and interpretation of results in the Formality design verification process flow. This chapter focuses on running the verification and interpreting the results in Formality.

Start Formality Load Guidance Load Reference Load Implementation Perform Setup Match Compare Points Run Verify Interpret Results No Success? Debug Yes Done

Figure 9-1 Run Verify and Interpret Results in the Design Verification Process Flow

Basic Usage

At its most basic, the steps involved in running verification and interpreting the results are as follows:

- Verifying a Design
- · Reporting and Interpreting Results
- Interrupting Verification

Verifying a Design

To verify the implementation design against the reference design, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Verify.
verify	
[reference_designID]	
[implementation_designID]	

If you omit the reference and implementation design IDs from the command, Formality uses the reference and implementation designs that you specified when you read in your designs. For more information, see "Reading Designs" on page 6-8.

If you did not match compare points before verification as described in Chapter 8, "Matching Compare Points," the verify command first matches compare points and then checks equivalence. If all compare points are matched and no setup changes have been made, verification moves directly to equivalence checking without rematching.

If matching was performed but there are unmatched points or the setup was altered, Formality attempts to match remaining unmatched points before equivalence checking. The verify command does not rematch already matched compare points.

To force the verify command to rematch everything, specify the undo_match -all command beforehand.

Formality makes an initial low-effort verification attempt on all compare points before proceeding to the remaining compare points with matching hierarchy by signature analysis and high-effort verification. This initial attempt can significantly improve performance by quickly verifying the easy-to-solve compare points located throughout your designs. It also quickly finds most points that are not equivalent. Afterwards, Formality proceeds with

verifying the remaining compare points by partitioning (grouping) related points and verifying each partition in turn.

Verification automatically runs in incremental mode, controlled by the verification_incremental_mode variable (true by default). Each verify command attempts to verify only compare points in the unverified state. This means that after the verification is completed or has stopped, upon reissue of verify, the status of previously passing and failing points is retained and verification continues for unverified points. If matching setup has changed through the use of set_user_match or set_compare_rule, Formality determines which compare points are affected, moves them to the unverified state, and reverifies them. In addition, if the verification effort level has been raised, points that were aborted due to complexity are also verified again. To force verify to reverify all compare points, use the command's -restart option.

The following is an example of a verification results summary:

Matched Compare Points	BBPin	Loop	BBNet	Cut	Port	DFF	LAT	TOTAL
Passing (equivalent)	336	0	144	0	1946	43832	390	46648
Failing (not equivalent)	0	0	0	0	15	0	0	15
Aborted								
Hard (too complex)	0	0	0	0	0	2	0	2
Not Compared								
Constant reg						1113	212	1325
Don't verify	0	0	0	0	29	0	0	29
Unread	1	0	0	0	0	899	0	900
*******	*****	*****	*****	*****	*****	*****	*****	*****

Reporting and Interpreting Results

As part of your troubleshooting efforts, you can report passing, failing, unverified, and aborted compare points as shown:

fm_shell	GUI
Specify any of the following commands:	1. Click Debug.
report_passing_points	2. Click the Passing Points, Failing Points,
[-point_type point_type]	Aborted Points, Unverified Points, Probe Points, Analyzes, or Loops tab.
report_failing_points	
<pre>[-point_type point_type]</pre>	
<pre>report_aborted_points [-point_type point_type]</pre>	
report_failing_unverified	
<pre>[-point_type point_type]</pre>	
<pre>report_not_verified [-point_type point_type]</pre>	

Use the -point_type option to filter the reports for specific object types, such as ports and black box cells. For a complete list of objects that you can specify, see the man pages.

In the GUI, by clicking the display name, you can display compare points with either their original names or the names that they were mapped to due to the compare rules.

From the command line, this can be achieved by using the report_* -mapped command.

From the Formality shell, Formality displays information to standard output. This information is updated as the verification proceeds. From the transcript, you can see which design is being processed and observe the results of the verification. In the GUI, the transcript is displayed in the transcript area. In addition, a progress bar shows the status of verification.

During verification, Formality assigns one of five types of status messages for each compare point it identifies:

Status Message	Description
Passing	A passing point represents a compare point match that passes verification. Passing verification means that Formality determined that the functions that define the values of the two compare point design objects are functionally equivalent.
Failing	A failing point represents a compare point match that does not pass verification or does not consist of two design objects. Failing verification means that Formality determined that the two design objects that constitute the compare point are not functionally equivalent.
Aborted	An aborted point represents a compare point that Formality did not determine to be either passing or failing. The cause can be either a combinational loop that Formality cannot break automatically or a compare point that is too difficult to verify.
Unverified	An unverified point represents a compare point that has not yet been verified. Unverified points occur during the verification process when the failing point limit has been reached or a wall clock time limit is exceeded. Formality normally stops verification after 20 failing points have been found.
Not Verified	A Not Verified, or Not Run, point appears if there was some error that prevented verification from running.

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

Classification	Description
Succeeded	The implementation design was determined to be functionally equivalent to the reference design. All compare points passed verification.
Failed	The implementation design was determined to be not functionally equivalent to the reference design. Formality found at least one compare point object in the implementation design that was determined as being nonequivalent to its comparable object in the reference design. These points are called failing compare points.
	If verification is interrupted, either because you press Ctrl+C or a user-defined time-out occurs, such as the verification_timeout_limit variable, and if at least one failing point was detected before the interrupt, Formality reports a verification result of failed.
Inconclusive	Formality could not determine whether the reference and implementation designs are equivalent. This situation occurs in the following cases:
	- A matched pair of compare points was too difficult to verify, causing an "aborted" compare point, and no failing points were found elsewhere in the design.
	- The verification was interrupted, either because you pressed Ctrl+C or a user-defined time-out occurred, and no failing compare points were detected before the interrupt. This results in "unverified" compare points.

For information about failing or inconclusive verification due to aborted points, see "Determining Failure Causes" on page 10-7, and for information about how to handle aborted points due to loops, see "Asynchronous State-Holding Loops" on page 7-37.

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

Interrupting Verification

To interrupt verification, press Ctrl+C. Formality preserves the state of the verification at the point you interrupted processing, and you can report the results. You also can interrupt Formality during automatic compare-point matching.

Introduction

When you issue the <code>verify</code> command, Formality attempts to prove design equivalence between an implementation design and a reference design. This section describes how to verify a design or a single compare point, as well as how to perform traditional hierarchical verification and batch verifications.

Advanced Usage

The advanced steps involved in running verification and interpreting the results are as follows:

- Saving the Session Information
- Verifying a Single Compare Point
- Controlling Verification Runtimes
- Verification Using Multicore Processing
- Performing Hierarchical Verification
- Verifying ECO Designs
- Using Batch Jobs
- Verifying Blocks Under a Certain Level Independently
- Removing Compare Points From the Verification Set
- Verification Using Checkpoint Guidance
- Identifying Inferred Register Names With Register Mapping

Saving the Session Information

You can save the session information at various intermediate states of verification and restore it later. When the session is restored, verification resumes from the state at which the session file was saved.

To save the session information, use the <code>verification_auto_session</code> variable. The syntax of the variable is

```
set verification_auto_session on | off | timeout | verify | match
```

The default is on. You can set different values to specify when the tool saves session files.

on

The tool saves session files when the verification terminates with a result other than succeeded and when it reaches the verification timeout threshold that is set using the verification_timeout_limit variable.

off

The tool does not save session files.

timeout

The tool saves session files only when it reaches the verification timeout threshold that is set using the <code>verification_timeout_limit</code> variable.

verify

In addition to the session files that are saved when the variable is set to on, the tool saves a session file after each effort level of the verify command.

match

In addition to the session files that are saved when the variable is set to verify, the tool saves a session file after running the match command.

The Formality tool saves the session information in the formalityn_auto.fss file, where n is an incremental integer, in the directory where the generated files are stored. To restore the session, use the restore session command.

Setting a Threshold to Save Session Files

To specify a time threshold after which the <code>verification_auto_session</code> variable saves session files automatically, use the <code>verification_auto_session_threshold</code> variable. After the specified time, the tool saves session files automatically when there is a user-specified interrupt or if verification is not successful. The syntax to specify the threshold is

```
set verification_auto_session_threshold hh:mm:ss
```

Where hh is an integer that specifies the duration in hours, mm is an integer that specifies the duration in minutes, and ss is an integer that specifies the duration in seconds. The default is 12:00:00, which specifies 12 hours.

Verifying a Single Compare Point

Single compare point verification is useful when you have trouble verifying a complete design and you want to debug an isolated compare point in the implementation design.

To verify a single compare point, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	1. Click Verify.
verify [-type type]	2. Select a compare point in the list.
objectID_1 objectID_2	3. Click Verify Selected Point.
-inverted	
[-constant0 -constant1]	

When design objects of different types have the same name, change the -type option to the unique object type.

Besides verifying single compare points between two designs, you can also verify two points in the same design or verify an inverted relationship between two points. To verify that a certain output port has the same value as a certain input port in the same design, use the command

```
verify $impl/input_port $impl/output_port
```

To verify an inverted relationship between two points, use the -inverted switch with the verify command.

In addition, you can verify a single compare point with a constant 1 or 0. Using either the <code>-constant0</code> or <code>-constant1</code> option of the <code>verify</code> command causes Formality to treat a point that evaluates to a constant as a special single compare point during verification. You can access this functionality through the GUI when you are in the Match or Verify steps by using the Run menu from the main window's menu bar.

To verify a subset of compare points, see "Removing Compare Points From the Verification Set" on page 9-22. For information about interpreting results, see "Reporting and Interpreting Results" on page 9-5.

Controlling Verification Runtimes

To control the total verification runtime, you can specify how long Formality is allowed to run the verification process by doing the following:

fm_shell	GUI
Specify:	Click Verify.
<pre>set_app_var verification_timeout_limit value</pre>	 Choose Edit > Formality Tcl Variables. The Formality Tcl Variables dialog box appears.
	 From Verification, select the verification_timeout_limit variable.
	 In the Enter a time (hh:mm:ss) box, enter none for no limit or specify a time in the hh:mm:ss format.
	5. Choose File > Close.

The verification_timeout_limit variable sets a maximum wall clock time (not CPU time) limit on the verification run. Be careful when using this variable, because Formality halts the verification when it reaches the limit regardless of the state of the verification.

Verification Using Multicore Processing

Multicore processing during verification improves the runtime by dividing large tasks into smaller tasks for processing.

To enable multicore processing, use the set_host_options command. For example, to enable the use of four cores to run your processes,

```
fm_shell> set_host_options -max_cores 4
```

The maximum number of cores you can specify is eight. However, each Formality license supports only four cores. If you specify five or more cores, the tool checks out a second Formality license.

Use the report_host_options command to identify the number of cores specified.

For more information about the set_host_options and report_host_options commands, see the command man pages.

Performing Hierarchical Verification

By default, Formality incorporates a hybrid verification methodology that combines the setup associated with flat verification with the benefits of hierarchical verification.

The write_hierarchical_verification_script command generates a Tcl script that you can edit and run to perform hierarchical verification. The script uses accurate block-level port constraints to reduce the number of blocks that fail verification and reduce the incidence of false failures. The blocks that fail verification are reverified during the verification of higher-level hierarchical blocks.

The script performs verification on comparable lower hierarchical blocks, one at a time, regardless of the number of instantiations. Verification starts at the lowest levels of the hierarchy and works upward. Explicit setup commands are generated to capture the top-level context.

By default, for each matched block of the current top-level implementation and reference designs, the Tcl script:

- Generates black boxes for subdesigns that are successfully verified. If the
 -dont_resolve_failures option is used, black boxes of subdesigns are created
 irrespective of the verification results.
- Removes unused compare points.
- Sets port matches for ports matched by means other than their names.
- Sets input port constants.
 - To override this behavior, use the -noconstant option.
- Sets input port equivalences for unmatched input ports known to be equivalent to other matched ports.
 - To override this behavior, use the -noequivalence option.
- Ignores inconsistent setup information for port matches, constants, and equivalencies.
 The generated script contains a comment to indicate that inconsistent setup information is ignored.

The script runs in the current session. If you run the hierarchical verification script in a different session, you must insert commands that read and link the reference and implementation designs.

GUI

To generate a script to perform hierarchical verification,

Specify: • (

fm shell

write_hierarchical_verification_script • [-replace] [-noconstant] [-noequivalence] [-match type] [-save_mode mode] [-save_directory path] [-save_file_limit integer] [-save_time_limit integer] [-level integer] [-path instance_specific_pathnames] [-block instance_specific_pathnames] [-dont_resolve_failures] [-top_level_only] filename

- Choose File > Write Hierarchical Script...
- Select the level at which to verify blocks in isolation
- Select the appropriate Setup Preferences.
- · Selected the type of Matching.
- Enter the directory in which to save the session files.
- Enter the file name in which to write the script.
- Select how many failing verification session files to save.
- Select the minimum amount of CPU seconds for a verification to use to save the session file.

You can customize this script to verify specific blocks and to constrain context information about the instantiated blocks.

The script reports the verification result for each block in a text file that is concatenated to the transcript. To save the verification session files specific to a verification status,

fm_shell> write_hierarchical_verification_script -save_mode mode

Specify one of the following modes:

- auto: Saves the session files for all verification results, except for those that fail because it attempts to resolve failing subblock. This is the default.
- not_passed: Saves the session files for the blocks that did not pass verification.
- failed: Saves the session files for those blocks that fail verification.
- inconclusive: Saves session files for inconclusive verifications. If the
 -dont_resolve_failures option is specified, the command saves session files for both
 failing and inconclusive verifications.

To view the verification result for each block in the GUI, if you run hierarchical verification in the GUI, select Open Hierarchical Results from the File menu.

Traditional hierarchical verification, by creating black boxes of subdesigns irrespective of the verification results, is useful when you want to verify and view explicit, block-by-block hierarchical results. To generate a script to perform the traditional hierarchical verification, without eliminating false failures, use the <code>-dont_resolve_failures</code> option.

For more information about the write_hierarchical_verification_script command, see the man page.

Verifying Feedthroughs in Hierarchical Subdesigns

Feedthroughs help to reduce congestion in hierarchical designs. Feedthrough can be a buffer, an inverter, or a constant. You need to create a Tcl script that contains set_feedthrough_point commands to specify feedthrough behaviors for each

- hierarchical subdesign
- created black box of hierarchical subdesigns

The set_feedthrough_point command can be used only in the setup mode after specifying the reference and implementation design. The tool issues an error message

- If the specified reference or implementation design does not exist
- If the specified input or output port name does not belong to the specified designs
- If the output port name is used previously with the set_feedthrough_point command for the same design

The feedthrough verification result is displayed as part of the Formality tool verification result in the Formality log file. To report the status of feedthrough points, use the report_feedthrough_status command.

Example 9-1 Status of Feedthrough Points

```
fm_shell (verify)> report_feedthrough_status
**********
Report : feedthrough_status
Reference : r:/WORK/dp
Implementation : i:/WORK/dp
Version : 0-2018.06
         : Thu May 17 14:07:09 2018
Output
                                   Status
PASS
                                    PASS
                                   FAIL
[1]
                i:/WORK/dp/dp/m1/bb1/bbout
                                    UNKN
```

For more information about feedthrough verification and report of verification result, see the set_feedthrough_point and report_feedthrough_status man pages.

Subdesigns

The tool checks whether each feedthrough specification is present by traversing the design's hierarchy to confirm topological equivalence. While verifying feedthrough specifications in hierarchical subdesigns,

- The tool verifies that the output port specified with the -output option is driven by the
 input port specified with the -input option, and the ports are present in the feedthrough
 specification. The feedthrough paths from input ports to output ports can only have
 buffers or inverters.
- Ports appearing in the feedthrough specification are not considered during matching and verifying because they are verified as a compare point to avoid verification failure.
- The same port name that exists in both the reference and implementation designs must be specified in the feedthrough specification for both designs whether they have the same functions or not.

The Tcl script shown in Example 9-2 includes feedthrough specifications for verification. While verifying, the tool applies the feedthrough specification found in reference design A. However, the tool does not apply the feedthrough specification in design B because it cannot find implementation design B. The verification result is shown in Example 9-3.

Example 9-2 Feedthrough Specifications

```
set_feedthrough -design r:/WORK/A -output o2 -input i1
set_feedthrough -design i:/WORK/B -output o1 -tie_low
set_feedthrough -design i:/WORK/B -output o3 -input i1 -invert
```

Example 9-3 Verifying the Feedthrough Specification for a Subdesgin

```
read... <RTL source for "A">
set_top r:/WORK/A
set_reference r:/WORK/A
read... <netlist for "A">
set top i:/WORK/A
set_implementation i:/WORK/A
source <tcl file with feedthrough specification>
Warning (FM-XXX): ignoring feedthrough specification for design "i:/WORK/
<other setup>
verify
******************** Verification Results *****************
Verification SUCCEEDED
Reference design: r:/WORK/A
 Implementation design: i:/WORK/A
 115 Passing compare points
 0 Failing compare points
 0 Aborted compare points
 0 Unverified compare points
 O Failing feedthrough points
 1 Passing feedthrough points
```

Black Box of Subdesign

While verifying top-level designs, the tool creates a black box for each of the lower-level hierarchical subdesigns (incomplete subdesigns or any hierarchical subdesigns are black boxed)

While verifying the feedthrough specification for the black box of hierarchical subdesigns,

- The tool links each black box of hierarchical subdesigns with a feedthrough specification
 to a new design. The new design forms a wrapper around the black box that converts the
 feedthrough specification of the black box into functional constraints using explicit
 connections.
- The output ports are explicitly driven by the input ports or by constants.
- The tool ignores the feedthrough specification for the subdesign that is not black-boxed and issues a warning message.

Assuming subdesigns A and B are black-boxed, the tool verifies the design TOP with the Tcl script shown in Example 9-2. The Tcl script shown in Example 9-2 does not have feedthrough specification for the black box of subdesigns. The verification result is shown in Example 9-4.

Example 9-4 Verifying Feedthrough Specification for Black Box of Subdesgins

```
read... <RTL source for "Top">
set_top r:/WORK/Top
set_reference r:/WORK/Top
read... <netlist for "Top">
set top i:/WORK/Top
set_implementation i:/WORK/Top
source <tcl file with feedthrough specification>
Info: Converted feedthrough specification for design "r:/WORK/A" into
functional constraints.
Info: Converted feedthrough specification for design "i:/WORK/B" into
functional constraints.
<other setup>
verify
********************** Verification Results *****************
Verification SUCCEEDED
Reference design: r:/WORK/top
 Implementation design: i:/WORK/top
 167 Passing compare points
 0 Failing compare points
 0 Aborted compare points
 0 Unverified compare points
```

Verifying ECO Designs

To verify an ECO design, you need

- An SVF file that describes the changes that were made to the design's RTL source to accomplish an ECO.
- A setup file that maps the datapath operator name changes between the original design and the design for ECO.

Modifying the SVF File

The SVF file is generated when the RTL is synthesized. When the RTL is modified for ECO, the corresponding SVF file is no longer compatible. Using the Formality tool, you can modify the file automatically to ensure compatibility with the modified RTL.

Object names are derived from the RTL line number and the position in the line where they appear. The fm_eco_to_svf command accounts for the changes to the modified RTL that affect object names.

- Inserting, replacing, or deleting lines in the RTL changes the line numbers and affects the names of the operators on the lines.
- Adding, changing, or moving operators affects the naming of otherwise unedited operators on the same line.

For example, if a modification removes the first of two adders on line 123 in the RTL, the name of the second adder changes from add_123_2 to add_123.

Generating the SVF File for ECO

Use the fm_eco_to_svf script to automatically modify the SVF file for the modified RTL.

The script is located in the following directory:

```
install_dir/PLATFORM/fm/bin/fm_eco_to_svf
```

When you use the script, specify the original RTL file and then the modified RTL file. Alternatively, specify the directories that contain the original and the modified RTL files. The script finds matching file names and compares the contents to generate the guidance commands that indicate line changes.

You must run this script for each modified RTL and compile the changes in an SVF file. In this example, the name of the file is eco_change.svf. The first command creates the file and the consecutive command appends to the file.

For example, run the script using the following syntax:

```
fm_eco_to_svf original/my_design.v eco/my_design.v > eco_change.svf
fm_eco_to_svf original/my_design_2.v eco/my_design_2.v >> eco_change.svf
```

The generated SVF file contains the <code>guide_eco_change</code> commands that describe the location of each modification to the RTL. Single lines are represented by a single line number and multiple lines are represented by two line numbers that indicate the first line and the last line of the modified region.

The following examples show how the line numbers are indicated. The commands identify the changes to the mydsgn.v design.

The following example indicates that lines 4 and 5 in the modified RTL are inserted.

```
guide_eco_change -file {mydsgn.v} -type {insert} -original {4} -eco {4 5}
```

The following example indicates that line 7 in the original RTL is deleted.

```
guide_eco_change -file {mydsgn.v} -type {delete} -original {7} -eco {8}
```

The following example indicates that lines 12 through 14 in the original RTL are replaced by lines 13 and 14 in the modified RTL.

```
guide_eco_change -file {mydsgn.v} -type {replace} -original {12 14} -eco {13 14}
```

Generating the Automated Setup Mapping File

An automated setup mapping file maps datapath operator and general operators from the original SVF file to the modified SVF file. The mapping is based on the ECO SVF file that is generated using the fm_eco_to_svf script.

Example 9-5 shows how to generate the automated setup mapping file using the generate_eco_map_file command.

Example 9-5 Generating the Automated Setup Mapping File

```
fm_shell > set_svf eco_change.svf original.svf
fm_shell > read_container -r design_original.fsc
fm_shell > read_container -i design_eco.fsc
fm_shell > generate_eco_map_file -replace eco_map.svf
```

The mapping file lists the <code>guide_eco_map</code> commands that specify the design name, the original operator name, and the ECO operator name. The file also contains the general operator name changes that are mapped using the <code>guide_eco_map</code> command.

Example 9-6 shows the contents of an automated setup mapping file.

Example 9-6 Automated Setup Mapping File

```
quide
### IMPORTANT: Inspect and change the following guide_eco_map commands.
### Each "from" operator can be matched to at most one "to" operator,
### and vice versa.
### Uncomment the correct matches.
### INSPECT AND CHANGE THESE LINES
# guide_eco_map -design { my_design } -from { add_5 } -to { add_6 }
\# guide_eco_map -design \{ my_design \} -from \{ add_5 \} -to \{ add_6_2 \}
# guide_eco_map -design { my_design } -from { add_5 } -to { add_6_3 }
# guide_eco_map -design { my_design } -from {
                                              add_5_2 } -to { add_6 }
# guide_eco_map -design { my_design } -from {
                                              add_5_2 } -to { add_6_2
# guide_eco_map -design { my_design } -from {
                                              add_5_2 } -to { add_6_3 }
# guide_eco_map -design { my_design } -from { mult_5 } -to { mult_6 }
setup
```

Uncomment the required mapping.

Verifying a Design Modified for an ECO

To verify an ECO design after generating the ECO SVF files and the mapping file,

1. Read in the SVF files:

set_svf eco_change.svf eco_map.svf original.svf

- 2. Read in the design files for ECO.
- 3. Read in the ECO netlist.
- 4. Run verification using the verify command.

Uninstantiated Designs in Verilog Libraries

In a Verilog library that is read using the <code>read_verilog</code> command, only the cells that are specified using the <code>set_top</code> command are elaborated. The other library cells are not elaborated and are empty shells without pins, ports, or content. These cells cannot be edited and are not available for an ECO implementation.

Using the read_verilog -extra_library_cells command, you can specifically elaborate the cells that are not elaborated by the set_top command. The syntax is

```
read_verilog -extra_library_cells cell_list
```

You can use the <code>read_verilog -extra_library_cells</code> command either before or after the <code>set_top</code> command. When you run the command before running the <code>set_top</code> command, the specified cells are elaborated during the <code>set_top</code> command. Note that cells that are elaborated using the <code>read_verilog -extra_library_cells</code> command overwrite the cells that are already elaborated by the <code>set_top</code> command.

When you run the read_verilog -extra_library_cells command after running the set_top command, only the specified cells are elaborated. You can only elaborate cells that are not elaborated and existing cells are not overwritten. If elaborated cells are specified in the cell_list, the tool issues an error message.

For more information about the read_verilog command, see the command man page.

Using Batch Jobs

Running Formality shell commands in a batch job can save you time in situations where you have to verify the same design more than one time. You can assemble a stream of commands, or script, that sets up the environment, loads the appropriate designs and libraries, performs the verification, and tests for a successful verification. Any time you want to control verification through automatic processing, you can run a batch job.

Starting Verification Using Batch Jobs

For a sequence of fm_shell commands, you can start the batch job in several different ways:

 Enter fm_shell commands one at a time as redirected input. For example, from the shell, use commands in the following form:

```
% fm_shell << !
? shell_command
? shell_command
...
? shell_command
? !</pre>
```

 Store the sequence of commands in a file and source the file using the Tcl source command. For example, from the shell, use a command in the following form and supply a .csh file that contains your sequence of fm shell commands:

```
% source file
```

Note:

Be sure your .csh file starts by invoking Formality and includes the appropriate controls to redirect input.

• Submit the file as an argument to the <code>-file</code> option when you invoke Formality from the shell. For example, from the shell, use a command in the following form and supply a text file that contains your sequence of fm_shell commands:

```
% fm_shell -file my_commands.fms
```

The output Formality produces during a batch job is identical to that of a verification performed from the shell or GUI. For information about interpreting results, see "Reporting and Interpreting Results" on page 9-5.

Controlling Verification During Batch Jobs

In your script, you can provide control statements that are useful in concluding verification. In particular, you can take advantage of the fact that fm_shell commands return a 1 for success and a 0 for failure. Given this, the following set of commands at the end of your script can direct Formality to perform diagnosis, report the failing compare points, and save the session, should verification fail:

```
if {[verify]!=1} {
    diagnose
    report_failing_points
    cd ..
    save_session ./saved_state
}
```

Verification Progress Reporting for Batch Jobs

You can specify how much time is allowed to elapse between each progress report by using the <code>verification_progress_report_interval</code> variable. During long verifications, Formality issues a progress report every 30 minutes, by default. For updates at different intervals, you can set the value of this variable to *n* minutes.

Verifying Blocks Under a Certain Level Independently

If you do not want Formality to verify blocks under a certain level independently, use the <code>-level</code> option. This option causes Formality to ignore hierarchical boundaries below the level you set. You should use this option only if you have a reason to know that certain hierarchical boundaries below the level you specified have not been preserved. Use this option with caution because if you use it incorrectly, it can negatively affect verification performance.

Removing Compare Points From the Verification Set

You can elect to remove any matched compare points from the verification set. This is useful when you know that certain compare points are not equivalent, but want the rest of the verification to proceed and ignore those points.

To prevent Formality from checking for design equivalence between two objects that constitute a matched compare point, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
set_dont_verify_points	set_dont_verify_points
[-type ID_type]	[-type ID_type]
[object_1 [object_2]]	[object_1 [object_2]]

When you specify an object belonging to a matched compare point set, the second object is automatically disabled. Sometimes design objects of different types share the same name. If this is the case, change the <code>-type</code> option to the unique object type.

Specify instance-based path names or object IDs for compare points in the reference and implementation designs. Although black boxes and hierarchical blocks are not compare points, black box input pins are compare points.

Specify the remove_dont_verify_points command to undo the effect of set_dont_verify_points on specified objects; that is, to add them to the verification set again.

Specify the report_dont_verify_points command to view a list of points disabled by set_dont_verify_points.

These commands accept instance-based path names or object IDs.

Verification Using Checkpoint Guidance

During the matching step, the Formality tool verifies a checkpoint netlist against the RTL when a <code>guide_checkpoint</code> command is found in the SVF file. If the verification succeeds, the tool replaces the reference design with the verified checkpoint netlist for the subsequent verification of either the next checkpoint netlist or the final implementation netlist. However, if the checkpoint verification is not successful, the reference design is retained for the verification of the subsequent netlist.

When you use checkpoint guidance, the match and verification stages show additional information related to the checkpoints.

Controlling the Checkpoint Verification Flow

To prevent the Formality tool from verifying the checkpoint netlists even when a <code>guide_checkpoint</code> command is found in the SVF file, set the <code>svf_checkpoint</code> variable to <code>false</code>. By default, the variable is set to <code>true</code>. If the <code>match</code> command is not explicitly used, ensure that the <code>svf_checkpoint</code> variable is set before running the <code>verify</code> command.

By default, if the checkpoint verification is not successful, the tool continues the verification. To stop the overall verification if the checkpoint verification not successful, set the svf_checkpoint_stop_when_rejected variable to true. The default is false.

Investigating a Checkpoint Verification

You can investigate checkpoint verification by restoring a checkpoint session file.

By default, the Formality tool saves sessions of checkpoint verifications only when they fail. To save the session files of each verification regardless of the result, set the svf_checkpoint_save_session variable to all. The default is not_passed. The allowed values are all, none, not_passed, failed, and inconclusive.

The tool saves the session files in a directory named fm_checkpoint_sessions and displays a message:

```
Info: Checkpoint session file saved at 'fm_checkpoint_sessions/
checkpoint_1234.fss'
```

To prevent saving session files regardless of the verification status, set the variable to none.

Any interrupt, such as a verification timeout limit, also affects the checkpoint verification. In this case, a session file is only written out if specified by the svf checkpoint save session variable.

Known Limitations

The Formality tool does not support verification using checkpoint guidance in some cases. You still need to use the two-step verification flow using user-generated intermediate netlists in the following cases:

- When you run the create_register_bank command on retimed registers, the tool does not support the flow that has SVF commands, except guide_environment, between the guide_retiming_finished and guide_checkpoint commands. In the Design Compiler tool, if you run an incremental compile or any command that generates verification guidance between the compile_ultra command and the create_register_bank command,
 - a. Verify RTL to the pre-multibit mapping netlist.
 - b. Verify the pre-multibit mapping netlist to the post multibit mapping netlist.
- When retiming is performed multiple times on a design using the following commands:
 - o compile_ultra -retime followed by compile_ultra -incremental -retime
 - o set_optimize_registers and compile_ultra followed by compile ultra -retime

Identifying Inferred Register Names With Register Mapping

The inferred register names in the RTL might change in SVF guidance during register optimizations, such as merge, duplication, inversion push, constant optimization, or using the <code>change_name</code> command. To identify inferred register names in post-SVF guidance, use the <code>write_register_mapping</code> command to generate a report that maps registers in the reference design with the matching registers in the implementation design. Use the <code>-replace</code> option to overwrite the existing file.

If the reference design is an RTL, the report is based on the original RTL names as the inferred register names in the RTL might change because of SVF guidance.

Note:

You must use this command only after using the match or verify commands.

For examples of reports generated by the write_register_mapping command, see the command man page.

The following rules apply when generating reports with the write_register_mapping command:

• For register technology cells with multiple output pins, the command reports information only for the first output pin. The remaining output pins are not reported.

 Unsupported register mappings are reported with the comment (#) symbol. See Example 9-7.

- The command cannot track the original name of RTL for
 - o Retimed registers, including pipeline registers in DesignWare parts.
 - Duplicate registers that undergo multiple optimizations, such as inversion push, using the change_names command, or banking.

The report shows only the post SVF names for these duplicate registers.

- RTL-instantiated multibit logic library cells; however, the original RTL names are tracked for multibit banking of inferred registers.
- Technology cell pin names for matched registers with complex output drivers, such as nonbuffer or inverter cells. However, polarity is determined from the compare point match reports.

Example 9-7 shows the report of technology cell pin names for the matched registers with complex output drivers and unsupported register mappings with the comment (#) symbol.

Example 9-7 Reports Technology Cell Pin Names and Unsupported Register Mappings With the Comment (#) Symbol

```
# Register Mapping File
# Created using Formality (R) Version L-2016.03 -- Feb 3, 2016
# Reference top design : r:/WORK/test
# Implementation top design : i:/WORK/test
# Timestamp : Wed Feb 3 04:20:19 2016
#
# ref pos pl
# impl pos pl
```

10

Debugging Verification

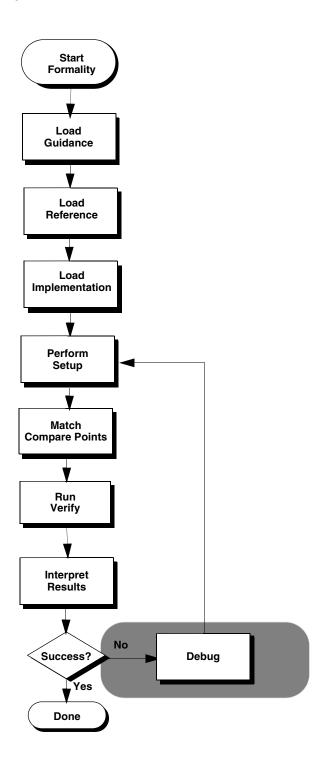
There are two main verification results that require debugging, specifically those with failing points and those verifications for which Formality did not come to a conclusive result because of the complexity of the design.

This chapter includes the following sections:

- Introduction
- Debugging a Failing Verification
- Debugging a Hard Verification
- Alternate Strategies to Resolve Hard Verifications

Figure 10-1 outlines the timing of the debugging step within the design verification process flow. This chapter focuses on how to debug failing designs in Formality.

Figure 10-1 Debug in the Design Verification Process Flow Overview



Introduction

Prior to debugging the specific instances of a failing verification or a hard verification, you should understand how the general debug process works and what information can be gleaned from it.

This chapter includes the following sections:

- Debug Process Flow
- Gathering Information

Debug Process Flow

Figure 10-2 shows an overview of the debugging process as described in this chapter. The A in the diagram symbolizes a wire connection. The debugging process for technology library verification is described in Chapter 12, "Library Verification Mode."

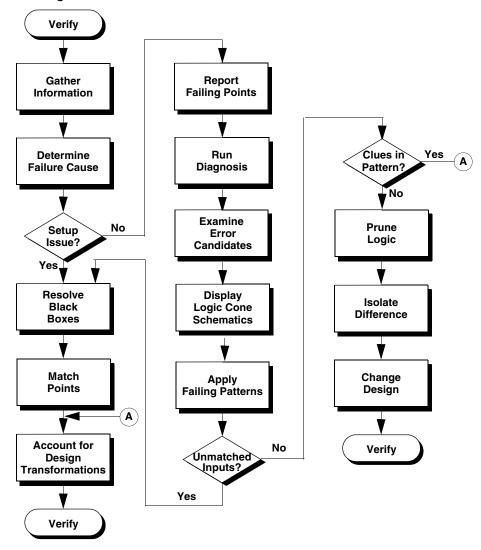


Figure 10-2 Debug Process Flow Overview

Gathering Information

When a verification run reports that the designs are not equivalent, failure is due either to an incorrect setup or to a logical design difference between the two designs. Formality provides information that can help you determine the cause of the verification failure. The following sources provide you with this information:

- The transcript window provides information about verification status, black box creation, and simulation or synthesis mismatches.
- The formality.log file provides a complete list of black boxes in the design, assumptions
 made about directions of black box pins, and a list of multiply driven nets.
- Reports contain data on every compare point that affects the verification output. These
 reports are named report_failing, report_passing, and report_aborted.

This chapter describes when and how to use the various information sources during the debugging process.

Debugging a Failing Verification

Occasionally, Formality encounters a design that cannot be verified because it is particularly complex. For example, asynchronous state-holding loops can cause Formality to terminate verification if you did not check for their existence before executing the <code>verify</code> command. For more information, see "Asynchronous State-Holding Loops" on page 7-37.

The following steps provide a strategy to apply when verification does not finish due to a design difficulty. Note that these steps are different from those presented in "Determining Failure Causes" on page 10-7, which describes what to do when verification finishes but fails.

Note:

Incomplete verifications can occur when Formality reaches a specified number of failing compare points. This limit causes Formality to stop processing. Use the verification_failing_point_limit variable to adjust the limit as needed.

- 1. If you have both aborted points and failing points, locate and fix the failing compare points. For strategies about debugging failed compare points, see "Debugging Unmatched Points" on page 8-6.
- 2. Verify the design again. Fixing the failing compare points can sometimes eliminate the aborted points.
- 3. After eliminating all failing compare points, isolate the problem in the design to the smallest possible block.

- 4. Declare the failing blocks as black boxes by using the set_black_box command. Use the set_black_box command to specify the designs that you want to black box.
 - Alternatively, you can insert cutpoint black boxes to simplify hard-to-verify designs, as described in "Retiming Using Design Compiler" on page 7-26.
- 5. Verify the implementation design again. This time the verification should finish. However, the problem block remains unverified.
- 6. Use an alternative method to prove the functionality of the isolated problem block. For example, in a multiplier example, use a conventional simulation tool to prove that the multiplier having the different architecture in the implementation design is functionally equivalent to the multiplier in the reference design.
 - At this point, you have proved the problem block to be equivalent and you have proved the rest of the implementation design equivalent. One proof is accomplished through a conventional simulation tool, and the other is accomplished through Formality. Both proofs combined are sufficient to verify the designs as equal.
 - Establish the existing implementation design as the new reference design. This substitution follows the incremental verification technique described in Figure 1-1 on page 1-3.
- 7. Prior to running verification a second time, manually match any equivalent multipliers that Formality has not automatically matched in the reference and implementation designs. Manually matching the multipliers aids the solver in successfully matching remaining multipliers. Use the report_unmatched_points -datapath command to identify the unmatched multipliers.
- 8. Preverification might have timed out due to the effort level set in the verification_datapath_effort_level variable. You can set this limit to a higher effort level to allow Formality more time to preverify any black box equivalent datapath blocks successfully.

Determining Unread Failing Compare Points

To determine if any failing compare points are unread, use the unread_analysis command. The command does not match the failing compare point and checks whether verification fails with any downstream points. If the downstream points are not affected by the unmatched point and verification succeeds, the tool marks the compare point as unread in the design.

The following example shows the analysis of an unread and a read compare point in the final summary:

```
fm_shell> unread_analysis

Analyzing unread compare point: r:/WORK/mychip/myblock/abc/def/keyval_reg
Level 0: UNREAD

Analyzing unread compare point: r:/WORK/mychip/instr_reg
Level 0: 16 potential readers
Level 1: READ, 66 readers

Unread analysis cleanup
New unread points:
r:/WORK/mychip/myblock/abc/def/keyval reg
```

Determining Failure Causes

To debug your design, you must first determine whether a failing verification is due to a setup problem or a logical difference between the designs.

Use the <code>analyze_points</code> <code>-failing</code> command to have Formality examine the failing points and to determine if there is a possible setup problem. After executing this command, Formality generates a report of possible setup issues. If it is the case that the verification failed due to a setup problem, you should start the debug process by looking for obvious problems, such as forgetting to disable scan.

Sometimes you can determine the failure cause by examining the number of failing, aborted, and unmatched points, as shown in Table 10-1.

Table 10-1 De	etermining Failure	Cause
---------------	--------------------	-------

Unmatched	Failing	Aborted	Possible cause		
Number of points in each category:					
Large	-	-	Compare point matching problem, or black boxes		
Very small	Some	Small	Logical difference		
Very small	Some	Large	Setup problem		
Very small	None	Some	Complex circuits, combinational loops, or limits reached		

Setup problems that can cause a failed verification include unmatched primary inputs and compare points, missing library models and design modules, and incorrect variable settings.

The following steps describe how to make sure design setup did not cause the verification failure.

If you determine that your design contains setup errors, skip to "Eliminating Setup Possibilities" on page 10-11 to help you fix them. You must fix setup problems and then verify the implementation design before debugging any problems caused by logical differences between the designs.

- 1. If you automatically matched compare points with the verify command, look at the unmatched points report by running the report_unmatched_points command in fm_shell, or choosing Match > Unmatched in the GUI. The report shows matched design objects, such as inputs, as well as matched compare points; use the filtering options included with the command to view only the unmatched compare points.
 - Use the iterative compare point matching technique described in "Matching Compare Points" on page 2-5 to resolve the unmatched points.
 - A likely consequence of an unmatched compare point, especially a register, is that downstream compare points fail due to their unmatched inputs.
- 2. Specify the report_black_boxes command in fm_shell or at the Formality GUI prompt to check for unmatched black boxes. During verification, Formality treats comparable black boxes as equivalent objects. However, to be considered equivalent, a black box in the implementation design must map one to one with a black box in the reference design. In general, use black box models for large macro cells, such as RAMs and microprocessor cores, or when you are running a bottom-up verification.

Note:

Black boxes that do not match one-to-one result in unmatched compare points.

For information about black boxes, see "Black Boxes" on page 7-3.

3. Check for incorrect variable settings, especially for the design transformations listed in "Design Transformations" on page 10-19. To view a list of current variable settings, use the printvar command.

Debugging Using Diagnosis

At this point, you have fixed all setup problems in your design or determined that no setup problems exist. Consequently, the failure occurred because Formality found functional differences between the implementation and reference designs. Use the following steps to isolate the problem. This section assumes you are working in the GUI. For more information about the Formality verification and debugging processes, see Chapter 2, "Formality Use Model."

After you have run verification, debug your design by taking the following steps:

- 1. From the Debug tab, click the Failing Points tab to view the failing points.
- 2. Run diagnosis on all of the failing points listed in this window by clicking Analyze.

After clicking Analyze, you might get a warning (FM-417) stating that too many distinct errors caused diagnosis to fail (if the number of distinct errors exceeds five). If this occurs and you have already verified that no setup problems exist, try selecting a group of failing points, such as a group of buses with common names, and click Diagnose Selected Points. If the group diagnosis also fails, select a single failing point and run selected diagnosis.

After the diagnosis is complete, the Error Candidate window displays a list of error candidates. An error candidate can have multiple distinct errors associated with it. For each error, the number of related failing points is reported. There can be alternate error candidates apart from those shown in this window.

- 3. Inspect the alternate candidates by using Next and Previous. You can reissue the error candidate report anytime after running diagnosis by using the report_error_candidates Tcl command.
- 4. Select an error with the maximum number of failing points. Right-click that error, and then choose View Logic Cones. If there are multiple failing points, a list appears from which you can select a particular failing point to view. Errors are the drivers in the design whose function can be changed to fix the failing compare point.

The schematic shows the error highlighted in the implementation design along with the associated matching region of the reference design.

Examine the logic cone for the driver causing the failure. The problem driver is highlighted in orange. You can select the Isolate Error Candidates Pruning Mode option to view the error region in isolation. You can also prune the associated matching region of the reference design. To undo the pruning mode, choose Edit > Undo. For more information about pruning, see "Pruning Logic" on page 10-36.

Debugging Using Logic Cones

You want to debug the failing point that shows the design difference as quickly and easily as possible. Start with the primary outputs. You know that the designs are equivalent at primary outputs, whereas internal points could have different logic cones due to changes such as boundary optimization or retiming. Pick the smallest cone to debug. Look for a point that is not part of a vector.

You can open a logic cone view of a failing compare point to help you debug design nonequivalencies. Use the following techniques to debug failing points in your design from the logic cone view:

1. To view the entire set of failing input patterns, choose View > Show Logic Cones > click the Show Patterns toolbar in the logic cone window.

A pattern view window appears. Click the number above a column to view the pattern in the logic cone view. For each pattern applied to the inputs, Formality displays logic values on each pin of every instance in the logic cone.

Check the logic cone for unmatched inputs. Look for unmatched inputs in the columns in both the reference and implementation designs. For example, two adjacent unmatched cone inputs (one in the references and one in the implementation design) have opposite values on all patterns, they should be matched.

Alternatively, you can also specify the report_unmatched_points compare_point command at the Formality prompt, or check the pattern view window for inputs that appear in one design but not the other.

There are two types of unmatched inputs:

Unmatched in cone

This input is not matched to any input in the corresponding cone for the other design. The logic for this cone might be functionally different. The point might have been matched incorrectly.

Globally unmatched

This input is not matched to any input anywhere in the other design. The point might need to be matched using name-matching techniques. The point might represent extra logic that is in one design but not in the other.

Unmatched inputs indicate a possible setup problem not previously fixed. For more information about fixing problems, see "Eliminating Setup Possibilities" on page 10-11. If you change the setup, you must reverify the implementation design before continuing the debugging process.

For more information about failing input patterns and the pattern view window, see "Viewing, Editing, and Simulating Patterns" on page 10-40.

2. Bring up a logic cone view of your design.

A pattern view window appears. Click the number above a column to view the pattern in the logic cone view. For each pattern applied to the inputs, Formality displays logic values on each pin of every instance in the logic cone.

For more information about displaying your design in a logic cone, see "Logic Cones" on page 10-31.

3. Look for clues in the input patterns. These clues can sometimes indicate that the implementation design has undergone a transformation of some kind.

For a list of design transformations that require setup before verification, see "Design Transformations" on page 10-19.

4. Prune the logic cones and subcones, as needed, better to isolate the problem.

For more information, see "Pruning Logic" on page 10-36.

After you have isolated the difference between the implementation and reference designs, change the original design using these procedures and reverify it.

If the problem is in the gate-level design, a one-to-one correspondence between the symbols in the logic cone and the instances in the gate netlist should help you pinpoint where to make changes in the netlist.

To help you further when debugging designs, click the Zoom Full toolbar option to view a failing point in the context of the entire design. Return to the previous view by pressing Shift-a.

Eliminating Setup Possibilities

As discussed in the "Determining Failure Causes" on page 10-7 section, you must resolve setup problems as part of the debugging process. If your design has setup problems, you should check the areas discussed in the following sections (listed in order of importance):

- 1. Black Boxes
- 2. Unmatched Points
- 3. Design Transformations

Black Boxes

If the evidence points to a setup problem, check for black boxes. You can do this by,

- · Viewing the transcript
- · Checking the formality.log file
- Running the report_unmatched_points -point_type bbox command
- Running the report_black_boxes command in the Formality shell or Formality prompt from within the GUI

For more information about black boxes, see "Black Boxes" on page 7-3.

Unmatched Points

As described in "Debugging Unmatched Points" on page 8-6, you might need to match compare points manually by using the techniques described in this section. Normally, you do this during the compare point matching process, before running verification.

Matching With User-Supplied Names

You can force Formality to verify two design objects by setting two compare points to match. For example, if your reference and implementation designs have comparable output ports with different names, creating a compare point match that consists of the two ports forces Formality to match the object names.

Note:

Use caution when matching compare points. Avoid creating a situation where two design objects not intended to form a match are used as compare points. Understanding the design and using the reporting feature in Formality can help you avoid this situation.

To force an object in the reference to match an object in the implementation design, use the Formality shell or the GUI as shown:

fm_shell GUI			
Specify:	1. Click Match > Unmatched Points.		
set_user_match	2. Select a point in the reference list.		
[-type ID_type]	3. Select a point in the implementation list.		
[-inverted]	4. Select +, -, or ?.		
[-noninverted] object_1 object_2[]	Click the User Match Setup tab to view the list of user-specified matches.		
02)000_1 02)000_1 []			

Sometimes design objects of different types share the same name. If this is the case, change the -type option to the unique object type.

You can set the <code>-inverted</code> or <code>-noninverted</code> option to handle cases of inverted polarities of state points. Inverted polarities of state registers can be caused by the style of design libraries, design optimizations by synthesis, or manually generated designs. The <code>-inverted</code> option matches the specified objects with inverted polarity; the <code>-noninverted</code> option matches the specified objects with non inverted polarity. Polarity is indicated in the GUI with a "+" for noninverted, "-" for inverted, and "?" for unspecified.

The set_user_match command accepts instance-based path names and object IDs. You can match objects such as black box cells and cell instances, pins on black boxes or cell instances, registers, and latches. The two objects should be comparable in type and location.

Along with matching individual points in comparable designs, you can use this command to match multiple implementation objects to a single reference object (1-to-n matching). You do this by issuing set_user_match, matching each implementation object to the reference object. You cannot, however, match multiple reference objects to one implementation object. Doing so would cause an error. For example, the following command sets several implementation objects to one reference object, datain[55]:

```
set_user_match $ref/CORE/RAMBLK/DRAM_64x16/I_TOP/datain[55] \
```

Formality[®] User Guide Version O-2018.06

```
$impl/CORE/RAMBLK/DRAM_64x16/I_TOP/datain[55] \
$impl/CORE/RAMBLK/DRAM_64x16/I_TOP/datain[55]_0 \
$impl/CORE/RAMBLK/DRAM_64x16/I_TOP/datain[56]_0 \
$impl/CORE/RAMBLK/DRAM_64x16/I_TOP/datain[59]_0 \
$impl/CORE/RAMBLK/DRAM_64x16/I_TOP/datain[60]_0
```

Use the set_user_match command to match an individual point in a design, a useful technique if you do not see multiple similar mismatches. Note that this command does not change the names in the database. For example, the following design objects are not matched by the Formality name-matching algorithms:

```
reference:/WORK/CORE/carry_in
implementation:/WORK/CORE/cin
```

To match these design objects, use the set_user_match command as follows.

```
fm_shell (verify)> set_user_match ref:/
WORK/CORE/carry_in \impl:/WORK/
CORE/cin
```

1. Removing User-Matched Compare Points

To unmatch objects previously matched by the set_user_match command, use the Formality shell or the GUI as shown:

fm_shell	GUI		
Specify:	At the Formality prompt, specify:		
remove_user_match	remove_user_match		
[-all] [-type type]	[-all] [-type type]		
instance_path	instance_path		

This command accepts instance-based path names and object IDs.

2. Listing User-Matched Compare Points

To generate a list of points matched by the set_user_match command, use the Formality shell or the GUI as shown:

fm_shell	GUI		
Specify:	At the Formality prompt, specify:		
report_user_matches [-inverted -noninverted -unknown]	report_user_matches [-inverted -noninverted -unknown]		

The -inverted option reports only user-specified inverted matches. The -noninverted option reports only user-specified noninverted matches. The -unknown option reports user

matches with unspecified polarity. The GUI displays polarity of these points using "-" to indicate inverted user match, "+" to indicate noninverted user match, and "?" to indicate unspecified user match.

Matching With Compare Rules

As described in "Performing Compare Point Matching" on page 8-4, compare rules are user-defined regular expressions that Formality uses to translate the names in one design before applying any name-matching methods. This approach is especially useful if names changed in a predictable way and many compare points are unmatched as a result.

Note:

Because a single compare rule can map several design object names between the implementation and reference designs, use caution when defining compare rules. Regular expressions with loose matching criteria can affect many design object names.

Defining a compare rule affects many design objects during compare point matching. For example, if the implementation design uses a register naming scheme where all registers end in the string $_{r_0}$, while the reference design uses a scheme where all registers end in $_{reg}$. One compare rule could successfully map all register names between the two designs.

Compare rules are applied during the compare point matching step of the verification process.

1. Defining Compare Rules

To create a compare rule, use the Formality shell or the GUI as shown:

fm_shell	GUI		
Specify:	1. Click the Match > Compare Rule Setup.		
set_compare_rule	2. Click Set and click the Reference or		
-from search_pattern	Implementation tab.		
-to replace_pattern	3. Select an Object Type: any, port, cell or net.		
designID	4. Type a Search pattern and a Replace pattern in the respective fields.		
	6. Click OK.		

Supply "from" and "to" patterns to define a single compare rule, and specify the design ID to be affected by the compare rule. For the patterns you can supply any regular expression or arithmetic operator. You need to use \((and \)) as delimiters for arithmetic expressions, and you can use +, -, *, /, and % for operators.

Formality[®] User Guide Version O-2018.06

The set_compare_rule command does not permanently rename objects; it "virtually" renames compare points for matching purposes. The report commands are available for use after compare point matching is completed.

Compare rules are additive in nature so they should be written in such a way that rules do not overlap. Overlap can cause unwanted changes to object names that can negatively affect subsequent compare rules. The rules are applied one at a time throughout the design.

For example, the following registers are unmatched when two designs are verified:

```
reference:/WORK/top_mod/cntr_reg0
.
reference:/WORK/top_mod/cntr_reg9
implementation:/WORK/top_mod/cntr0
.
implementation:/WORK/top_mod/cntr9
```

You can use a single set_compare_rule command to match up all these points, as follows:

```
fm_shell (verify)> set_compare_rule ref:/WORK/top_mod \
    -from {_reg\([0-9]*\)$} \
    -to {\1}
```

In this example, the rule is applied on the reference design. Therefore, all $_reg\#$ format object names in the reference design are transformed to # format during compare point matching.

In the following example, assume that the registers are unmatched when two designs are verified:

```
RTL:/WORK/P_SCHED/MC_CONTROL/FIFO_reg2[0][0]
RTL:/WORK/P_SCHED/MC_CONTROL/FIFO_reg2[0][1]
RTL:/WORK/P_SCHED/MC_CONTROL/FIFO_reg2[1][1]

GATE:/WORK/P_SCHED/MC_CONTROL/FIFO_reg20_0
GATE:/WORK/P_SCHED/MC_CONTROL/FIFO_reg20_1
GATE:/WORK/P_SCHED/MC_CONTROL/FIFO_reg21_1
```

A single set_compare_rule matches up all these points:

This rule transforms all objects in the reference design that follow the format $_{reg2[\#][\#]}$ to $_{reg2\#_\#}$, where # is restricted to only 0 and 1 values. This rule is applied on the reference design, but it also can be changed so that it can be applied on the implementation design.

You can use \(and \) as delimiters for arithmetic expressions and then use +, -, *, /, and % operators inside the delimiters to determine them unambiguously to be arithmetic operators.

Formality[®] User Guide Version O-2018.06

For example, to reverse a vector from the reference bus [15:0] to the implementation bus [0:15] using an arithmetic expression, use the following command:

The "-" operator in the replace pattern means arithmetic minus.

2. Testing Compare Rules

You can test name translation rules on unmatched points or arbitrary user-defined names by using the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Match > Compare Rule Setup.
test_compare_rule [-designID -r -i]	Click Set and click the Reference or Implementation tab.
-from search_pattern -to replace_pattern	Select an Object Type and enter values in the Search pattern and Replace pattern box.
<pre>[-substring string] [-type type] Or</pre>	 4. Click Test and select Test With Unmatched Points or Test With Specified Names. If you select the Test With Unmatched Points tab, you can optionally type a
test_compare_rule -from search_pattern	substring that restricts the test to those unmatched points with the specified substring.
<pre>-to replace_pattern -name list_of_names</pre>	 If you select the Test With Specified Names tab, you must add a name or list of names in the Enter a name to test against box and click Add.
	5. Click Test.

You can test a single compare rule on a specific design or arbitrary points. You can also use this command to check the syntactic correctness of your regular and arithmetic expressions. To do so, you supply "from" and "to" patterns, specify the name to be mapped, indicate the substring and the point type, and specify the design ID to be affected by the proposed compare rule. A string that shows the results from applying the compare point rule is displayed with 0 for failure and 1 for success.

3. Removing Compare Rules

To remove all compare rules from a design, use the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	1. Click the Match > Compare Rules Setup.
remove_compare_rules [designID]	2. Select an Object Type and click Remove.3. Select a design, and then click OK.

Currently it is not possible to remove a single compare rule.

4. Listing Compare Rules

To track compare rules, you can generate reports that list them by using the Formality shell or the GUI as shown:

fm_shell	GUI
Specify:	Click Match > Compare Rules Setup.
<pre>report_compare_rules [designID]</pre>	

Each line of output displays the search value followed by the replace value for the specified design.

Matching With Name Subset

During subset matching, each name is viewed as a series of tokens, separated by characters in the <code>name_match_filter_chars</code> variable. Formality performs a best-match analysis to match names containing shared tokens. If an object in either design has a name that is a subset of an object name in the other design, Formality can match those two objects by using subset-matching algorithms. If multiple potential matches are equally good, no matching occurs.

Digits are special cases, and mismatches involving digits lead to an immediate string mismatch. An exception is made if there is a hierarchy difference between the two strings and that hierarchy name contains digits.

Use the name_match_allow_subset_match variable to specify whether to use a subset of the token-based name matching method and to specify which particular name to use. By default, the variable value is set to strict. Strict subset matching should automatically match many of the uniform name changes that might otherwise require a compare rule. This is particularly helpful in designs that have extensive, albeit fairly uniform, name changes resulting in an unreasonably high number of unmatched points for signature analysis to handle. The strict value ignores the delimiter characters and alphabetic tokens that appear

Formality[®] User Guide Version O-2018.06

in at least 90 percent of all names of a given type of object (if doing so does not cause name collision issues).

If the value of the name_match_use_filter variable is false, subset matching is not performed regardless of the value of the name match allow subset match variable.

For example, the following design object pairs are matched by the subset-matching algorithms:

```
reference:/WORK/top/state
implementation:/WORK/top/state_reg
reference:/WORK/a/b/c
implementation:/WORK/a/c
reference:/WORK/cntr/state2/reg
implementation:/WORK/cntr/reg
```

The following design object pairs would not be matched by the subset-matching algorithms:

```
reference:/WORK/top/state_2
implementation:/WORK/top/statereg_2
reference:/WORK/cntr/state_2/reg_3
implementation:/WORK/cntr/state/reg[3]
```

The first pair fails because state is not separated from statereg with a "/" or "_". In the second pair, the presence of digit 2 in state2 causes the mismatch.

Renaming User-Supplied Names or Mapping File

Renaming design objects is generally used for matching primary input and outputs.

To rename design objects, use the Formality shell or the GUI as shown:

fm_shell	GUI		
Specify:	At the Formality prompt, specify:		
rename_object	rename_object		
-file file_name	-file file_name		
<pre>[-type object_type]</pre>	[-type object_type]		
[-shared_lib]	[-shared_lib]		
[-container container_name]	[-container container_name]		
[-reverse] objectID	[-reverse] objectID		
[new_name]	[new_name]		

Formality[®] User Guide Version O-2018.06

This command permanently renames any object in the database. The new name is used by all subsequent commands and operations, including all name-matching methods. Supply a file whose format matches that of the report_names command in Design Compiler.

Note:

To rename multiple design objects from a file, specify the -file option. The file format should match that of the report_names command in Design Compiler.

Use the rename_object command to rename design objects that are not verification compare points. For example, you can use this command to rename the input ports of a design so that they match the input port names in the other design. Input ports must be matched to obtain a successful verification. This command supplies exact name pairs so you know the exact change that is going to take place.

For example, the following rename_object command renames a port called clk_in to clockin to match the primary inputs:

```
fm_shell (verify)> rename_object impl:/
*/am2910/clk in clockin
```

You can use the rename_object command to change the name of a hierarchical cell, possibly benefiting the automatic compare point matching algorithms. In addition, you can use it on primary ports to make a verification succeed where the ports have been renamed (possibly inadvertently).

You can also use the <code>change_names</code> command in Design Compiler to change the names in the gate-level netlist. However, depending on the complexity of name changes, Formality might match the compare points successfully when verifying two designs (one before and one after the use of the <code>change_names</code> command). To work around this problem, obtain the changed-names report from Design Compiler and supply it to Formality with the <code>rename object</code> command for compare point matching.

For example, the following rename_object command uses a file to rename objects in a design:

```
fm_shell (verify)> rename_object -file names.rpt \
    -container impl
    -reverse
```

Design Transformations

Various combinational and sequential transformations can cause problems if you do not perform the proper setup before verification. Setup requirements are discussed in Chapter 7, "Performing Setup," for the following common design transformations:

- Internal scan insertion on page 7-14.
- Boundary scan on page 7-14.
- Clock tree buffering on page 7-16.

- Asynchronous bypass logic on page 7-35.
- Clock gating on page 7-18.
- Inversion push on page 7-23.
- Re-encoded finite state machines on page 7-38.
- Retimed designs on page 7-25.

Schematics

Viewing cells and other design objects in the context of the overall design can help you locate and understand failing areas of the design. This section describes how to use schematics to help you debug failing compare points. It pertains to the GUI only.

Viewing Schematics

In any type of report window, you can view a schematic for any object described in the report. This feature lets you quickly find the area in your design related to an item described in the report.

To generate a schematic view,

- In any of the following tabs,
 - Click Verify > click the desired tab.
 - Click Match > Unmatched Points or Matched Points
 - Click Debug > Failing Points, Passing Points, or Aborted Points
- Select either a object or all objects.
- Choose View > View Reference Object or View Implementation Object.

After you perform these steps, a schematic view window shows the selected object. The object is highlighted and centered in the schematic view window. To expand and view an additional level, double-click the cone symbol:



Double-click this symbol in the schematic to expand one level of the schematic. Hold down the Shift key and double-click the cone to expand the complete branch.

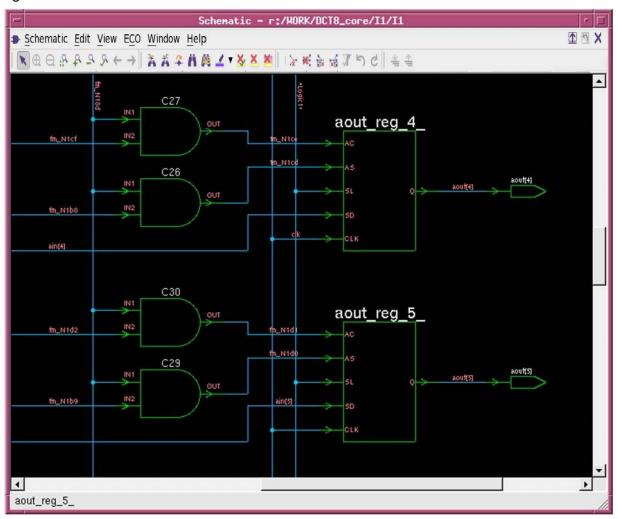
You can also choose Edit > Prune/Restore > Expand Schematic to view the complete schematic.

From the schematic view window, you can zoom in and out of a view, print schematics, and search for objects. You can also use the schematic view window menus to move up and down through the design hierarchy of the design.

To change the text size in a schematic, choose View > Increase Font Size or Decrease Font Size. Increasing or decreasing the font size changes the menu and window text but not the text in the schematic. Schematic text automatically increases or decreases as you zoom in or out.

Figure 10-3 shows a schematic view window.

Figure 10-3 Schematic View Window



The toolbar contains shortcuts to some menu selections. The schematic viewer supports the following tool options:



Set Select Mode (Esc): Click to select particular sections of the design.



Zoom In Tool: Click to increase the magnification applied to the schematic area by two times (2X).



Zoom Out Tool: Click to decrease the magnification applied to the schematic area by approximately 2X.



Zoom Full (F) - Click to redraw the displayed schematic sheet so that all logic is viewable.



Zoom In (I): Click to increase the magnification of the selected object.



Zoom Out (O): Click to decrease the magnification of the selected object.



Zoom Fit Selection (T): Click to fit the selected objects in the schematic area.



Back (Shift+A): Click to go to the previous view.



Forward (A): Click to go to the next view.



Push Design (P): Click to push into the selected level of hierarchy.



Pop Design (Shift+P): Click to pop out of the current level of hierarchy.



Find Net Driver (D): Click to find the driver for the selected net.



Find Net Load (L): Click to find the load on the selected net.



Skip Crossings: Click to skip hierarchical crossings of the selected net.



Find Initial Object (H): Click to find the initial object of the selected net.



Find By Name (F3): Click to display the object finder dialog box to find an object by name in the schematic.



Color Selected (Ctrl+=): Click to select the color to apply on the selected objects.



Clear Selected (Ctrl+-): Click to remove highlighting from the selected objects.



Clear Current Color: Click to clear highlighting from all objects that are highlighted with the current color.



Clear All (C): Click to clear highlighting from all objects.



Remove Fan-In (F6): Click to remove the fanin from the selected net.



Remove Fan-Out (Shift+F6): Click to remove the fanout from the selected net.



Isolate Fan-In (F7): Click to isolate the fanin from the selected net.



Isolate Fan-Out (Shift+F7): Click to isolate the fanout from the select net.



Return Fan-InOut (Ctrl+F6): Click to revert the fanin or fanout cone changes made on the selected net.



Undo Last (Z): Click to undo the last edit of fanin or fanout operation.



Revert (Shift+Z): Click to revert to the original fanin or fanout cone before editing operations.

The schematic area displays a digital logic schematic of the design. You can select an object in the design by clicking it. To select multiple objects, hold down the Shift key. Selected objects are highlighted in yellow.

Traversing Design Hierarchy

From a schematic view window, you can move freely through a design's hierarchy.

You can use either of these methods to traverse a design's hierarchy:

• To move down the hierarchy, select a cell and then click the Push Design toolbar option. Formality displays the schematic for the selected instance. This option is dimmed when there is nothing inside the selected cell.

• To move up the hierarchy, select a cell and then click the Pop Design toolbar option. Formality displays the design containing the instance of the current design, selects that instance in the new schematic, and zooms in on it.

To retain selection of a port, pin, or net when traversing hierarchy, use the following method:

- To move down the hierarchy, select both the desired pin or net and the corresponding cell, using Ctrl+click. Next, click the Push Design toolbar option.
- To move up the hierarchy, select a port or corresponding net, and then click the Pop Design toolbar option.

Finding an Object

To find an object in the displayed design,

- 1. In the schematic view window, choose Edit > Find > Find By Name. The Find By Name dialog box appears, which lists the objects in the design.
- 2. From the By Name list, select Cells, Ports, Nets, or Hier Crossings. Objects of the selected type are displayed in the list box.
- 3. Select an object from the list.
 - To choose multiple objects sequentially, press the Shift key and select multiple objects. To choose multiple objects individually, press the Ctrl key and click multiple object names.
- 4. To choose the color to highlight the selected objects, choose a color from the color palette.
- 5. Click Highlight to highlight the objects or click Select to select the objects for further operations.
- 6. Click Close.

Formality displays the object at the center of the view.

Figure 10-4 shows the multiple objects displayed in Find By Name Object Finder.

Figure 10-4 Find By Name Object Finder



Generating Lists

Using the object finder, you can interact with a schematic through dynamic lists of drivers, loads, nets, cells, and ports. Click Find Net Driver, Find Net Load, Find Initial Object, or click the Find By Name toolbar, or choose the corresponding item from the menu to open the dialog box that you use to generate your preferred list.

For example, to get a list of loads for a net, follow these steps:

- 1. Click to select the desired net in your schematic.
- 2. Choose Edit > Find > Find Net Load.

The Object Finder dialog box appears with a list of loads for the net you selected.

Note:

If the net has a single load and you click Find Net Load, the GUI takes you directly to the load without bringing up the dialog box. It is the same when you are using Find Net Driver.

3. Click one of the loads from the list.

Notice that the schematic has centered on and highlighted that cell.

You can also switch to a list of drivers from that cell by using Find Driver and selecting a driver from the list provided. Likewise, you can switch to a list of all cells, nets, or ports, and select one of those instead.

Zooming In and Out of a View

The schematic view window provides three tools that allow you quickly to size the logic in the window: Zoom In, Zoom Out, and Zoom Full.

Formality tracks each schematic view window's display history beginning with creation of the window. You can use Back and Forward to view the changes made in the same window.

To display the entire design, use the Zoom Full tool as follows:

- Choose View > Zoom Full.
- Right-click in the schematic window and choose Zoom Full.
- Click the Zoom Full toolbar.
- Press f on the keyboard.

Similarly, to zoom in or zoom out, choose View > Zoom In or Zoom Out, and click where you want the new view to be centered.

To zoom into a design repeatedly, do the following:

- 1. Place the pointer in the schematic area.
- 2. Press the equal sign key (=) to activate the Zoom In tool. The pointer changes to a magnifying glass icon with a plus symbol as if you had clicked the Zoom In Tool toolbar option.
- 3. Place the pointer where you want to zoom in and click.
- 4. Keep clicking as needed to zoom in further.

To zoom out of a design repeatedly, follow the same steps used to zoom into a design, except press the minus (–) key to activate the Zoom Out tool.

To zoom in on a small area quickly, invoke the Zoom In tool to display the magnifying glass pointer. Hold down the left mouse button and drag a box around the area of interest.

You can print the schematic from a schematic view window or a report from a report window.

To print a schematic, do the following:

- 1. In the schematic window, choose Schematic > Print. Make sure the schematic appears as you want it to print. You can use the Zoom In and Zoom Out toolbar buttons to get different views of the schematic.
- 2. In the Setup Printer dialog box, select the print options as desired, and then click OK. If you print to a file, you are asked to specify the file name.

After spooling the job to the printer, Formality restores the schematic view.

The procedure is the same for printing a schematic from a schematic window or a report from a report window. Use File > Print.

Viewing RTL Source Files in the Design Browser

From the hierarchical design browser, you can select an object and view its corresponding RTL or netlist source file.

To view the RTL source file:

- 1. Select a design object, such as a net.
- 2. Right-click and choose View > View Source.

Formality links and displays the RTL source file for the selected object. You can browse the selected object using the previous and next buttons.

In a report window, right-click a design object and choose View > View Reference Source or View Implementation Source.

The RTL source file display is not supported

- If the source is encrypted or created using DesignWare components
- If the cells are replaced during datapath optimization

Hierarchical Design Browser

This section explains how to view objects of hierarchical designs.

Listing Design Objects

By default, the design objects are hidden. To view the design objects from the hierarchical browser,

- Click the Designs menu.
 - The Designs Menu shows either Reference Design or Implementation Design based on the design used.
- Deselect the relevant boxes in the Hide Object panel to display the objects in the browser.

Reference, 1 - r:/WORK/fifo R File Edit View Designs Setup Window Help **↑** 🖔 X in i | A Y | @ 0 | ≥ B -Design Tab Reference Design | Design Libraries | Technology Libraries Hide Objects Ports: Input **②** Output ☑ Inout **Hide Objects** Primitives: **₹** Register ✓ Non-Register Panel Cells• FM Generated Tech Cells Black Boxes Search Panel Hide Search Design Name Onstance Name Reference: Instance Design Library - □- data_in[27] - □- data_in[28] - data_in[29] □ data_in[30] - **□** data_in[31] - 🗠 рор - □- pop_dk - □- push -□- push_dk -- □- reset_n Selection: r:/WORK/fifo/data_in[19] (fifo_mod.vg) Ready

Figure 10-5 Hierarchical Design Browser With Design Objects

Searching for Design Objects

To search for design objects from the search panel,

- Choose either design name or instance name.
- Enter the design object or the instance name.
- Click Search.

To hide the search results, click Hide.

Viewing Schematics

To view schematics from the browser,

- Select a design object or an instance from the browser.
- Choose View > View Instance, View Design, or View Object.
 You can also right-click the design object or instance name and choose View Object or View Design.

Viewing the RTL Source

To view the RTL source from the browser,

- Select a design object or an instance.
- From the View menu, choose View Instance Source, View Design Source, or View Source.

You can also right-click the design object or an instance and choose View Instance Source or View Design Source.

The RTL source is displayed in a text editor window. You can edit and save the source.

Performing Setup Tasks in the Design Browser

To setup the design from the browser,

- Select a design object or an instance.
- From the Setup menu, choose the setup operation that you want to perform. You can also right-click the design object or instance and choose the setup operation from the menu that appears.
- Click the Design Libraries or the Technology Libraries tabs to view the relevant libraries.

Queuing Setup Commands

When you are in the match or verify mode and issue setup commands, the tool queues them for execution later. These setup commands are displayed in the Command Queue window. The tool runs them when you revert to the setup mode.

You can also click the Execute Queue button in the Command Queue window to run the queued setup commands.

When you click the Execute Queue button, Formality

- Removes the results of match and verify.
- · Reverts to the Setup mode.
- Executes the queued commands.

Logic Cones

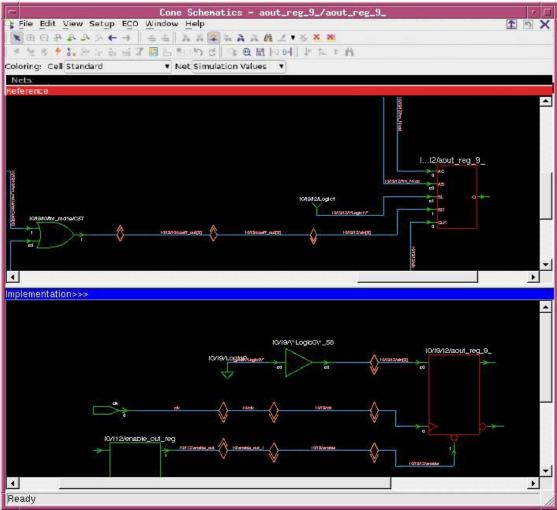
As described in step 2 of "Debugging Using Logic Cones" on page 10-9, you can view the logic cone of a failing compare point to help you debug design nonequivalencies.

To open a logic cone view,

- 1. Select a design object in a report window (passing points, failing points, aborted points, or verified points).
- 2. Right-click and click Show Logic Cones.

A logic cone window appears, as shown in Figure 10-6.

Figure 10-6 Logic Cone View Window



In the Logic Cone View window, the toolbar contains buttons that act as shortcuts to some menu selections. Some of the toolbar buttons that are common in Schematic and Logic view windows are shown in Figure 10-3 on page 10-21. The schematic viewer supports the following buttons:



Find X Sources (X): Click to find all net sources of the selected net with logic value X.



Find Compare Point (.): Click to find and zoom to the compare point in the schematic.



Find Matching Object (M): Click to find point in opposing window to match selected point in double-cone schematic.



Set Probe: Click to set probe points on objects.



Set Inverted Probe: Click to set inverted probe points on objects.



Set Verify Probe: Click to verify probe points.



Hide/Show Supply Nets: Click to hide or show supply nets.



Remove Non-Controlling (F5): Click to find point in opposing window to match selected point in double-cone schematic.



Remove Equal Valued CP Inputs (Ctrl+F5): Click to remove failing points from input cones with matching values for all patterns.



Remove Subcone(F6): Click to remove the subcone of the selected net.



Isolate Subcone(F7): Click to isolate the subcone of the selected net.



Isolate Error Candidates (F8): Click to prune logic and isolate error candidates.



Return Subcone (Ctrl+F6): Click to return the logic of the selected subcone.



Group All by Parent (Ctrl+G): Click to group all cells into their highest level of hierarchy.



Group Selected by Parent (G): Click to group the selected cell and its siblings into the next highest level of hierarchy.



Ungroup Selected (U): Click to ungroup the selected hierarchy.



Undo Last (Z): Click to undo the last edit cone operation.



Revert (Shift+Z): Click to revert to the original cone before editing operations.



Show Logic Cones: Click to view logic cone schematics.



Show Matching Tool: Click to bring up the matching analysis tool for this compare point.



Show Patterns: Click to open the patterns viewer window to show all patterns for current compare point.



Previous Pattern: Click to show the previous pattern values on the schematic.



Next Pattern: Click to show the next pattern values on the schematic.



Most Required Inputs: Click to sort rows by the most required inputs.



Least Required Inputs: Click to sort rows by the least required inputs.



Show/Hide Compare Point Info: Click to toggle for viewing compare point simulation results.



Find Report.. (Ctrl+F): Click to find values in the report.

In the Logic Cone View window, the two schematics display the logic cones, one each for the reference and implementation designs. The logic areas display object symbols, object names, object connections, applied states, and port names. To obtain information about an object in the logic area, place the pointer on it.

Formality displays the wire connections in different colors to represent the different coverage percentages of the error candidates. Nets and registers highlighted in magenta denote objects set with user-defined constants. The constant value is annotated next to the object. The following annotations are displayed next to failed registers:

Failure Cause Data: One register loads a 0 while the other loads a 1.

- Failure Cause Clock: One clock is triggered by a signal change, while the other is not.
- Failure Cause Asynch: One asynchronous reset line is high, while the other is low.

To view the logic cone schematic for any object,

- 1. Select an object from the cone schematic window.
- 2. Right-click and choose View Logic Cone from the context menu.

Viewing Combinational Feedback Loops

Highlighting combinational feedback loops in a design helps in debugging a failed verification or in resolving an inconclusive verification that might have occurred because of the loops. You can highlight the combinational feedback loops after match.

To highlight the combinational feedback loops,

- In the Debug tab, click the Get Loop Data button.
 The Get Loop Data button generates a report for the loop regions in the design.
- 2. In the loop report, select a compare point or an object.
- 3. Click the representative link.

The schematic of the design is displayed and the combinational loop is highlighted.

You can also highlight the combinational loops in the logic cone schematic window by choosing Hide/Show Loops from the Edit menu.

To display only the combinational loops instead of highlighting them in a design,

- 1. Choose Edit > Prune/Restore in the Cone Schematics window.
- 2. Choose Isolate Loop Regions.

Pruning Logic

Logic pruning reduces the complexity of a schematic so that you can better isolate circuitry pertinent to the failure. You generally prune logic toward the end of the debugging process, as noted in step 4 in "Debugging Using Diagnosis" on page 10-8.

To change the logic cone view to show only the logic that controls the output results, click the Remove Non-Controlling toolbar option. This command prunes away logic that does not have an effect on the output for the current input pattern, thus simplifying the schematic for analysis. Logic that has been pruned away is replaced with a cone symbol to indicate the change. To filter the pruned cone inputs, select the Filter pruned cone schematic inputs check box. You can see the filtered cone inputs in the Pattern window.

To aid in finding differences in the full schematic, remove the noncontrolling logic from the reference or implementation schematic and keep the full view in the other schematic.

To restore the full logic cone view, click the Undo last cone edit or Revert to original toolbar option, as applicable. The Undo button undoes the last change, while the Revert button restores the original logic cone view. It is also possible to restore a single subcone. Select the cone symbol of the subcone you want to restore, and click the "Return Selected Cone" toolbar option.

Sometimes looking at part of a logic cone is useful. Within Formality, a part of a cone is called a subcone. When you view logic in the logic area, you might be interested only in a particular subcone. You can remove and restore individual subcones in the display area.

To remove a subcone,

- 1. In the schematic window, click the net from which you want the subcone removed. The selected net is highlighted in yellow.
- 2. Click the "Remove Subcone of selected net or pin" toolbar option.

Formality redraws the logic without the subcone leading up to the selected net. The removed logic is replaced with a cone symbol.

To isolate a subcone,

- 1. Click the net whose logic cone you want to isolate. The selected net is highlighted in white.
- 2. Click the "Isolate subcone of selected pin or net" toolbar option.

Formality redraws the logic with only the subcone of the selected net visible. The logic for the subcones that are removed are replaced with cone symbols.

To return a subcone.

- 1. Click the cone symbol for the subcone you want to restore. The selected subcone is highlighted in white.
- 2. Click the "Return selected cone" toolbar option.

Grouping Hierarchy in a Logic Cone

Grouping hierarchy within a logic cone is another method to reduce complexity in a schematic to aid in debugging. To change the logic cone view to group all cells into their highest level of hierarchy, click the "Group All By Parent" toolbar option. This command examines the hierarchy of all cells in the logic cone and replaces cells in a common level of hierarchy with a block.

During debugging, you might find that it would be helpful to group a single level of hierarchy. To do this, select a cell and click the "Group Selected By Parent" toolbar option. This command examines the selected cell for its next highest level of hierarchy, find all other cells in the logic cone belonging to the same hierarchical level, and replace them with a single block.

To return logic that has been grouped, first select the hierarchical block you want to ungroup. Then click the "Ungroup Selected" toolbar option. This restores the logic that is replaced with the hierarchical block.

Setting Probe Points

The probing feature facilitates easier debugging of failing or hard verifications by allowing you to select two nets in a logic cone called probe points (one from the reference design and one from the implementation design) and determine if the logic is equivalent up to those probe points.

This probing feature introduces probe compare points that are kept separate from the existing set of compare points. Setting probe points can be done anytime in Formality after both designs have been read-in and linked. You can specify one or many probe points. However, the verification of probe points is available only in the Formality verify mode.

Probe verification is performed using the existing matching information and does not change the existing set of compare point matches. It is also important to note that probe compare points do not terminate the cones of their downstream compare points. They are compared like normal compare points but are never considered as input points of a logic cone.

In the GUI, click the Probe Points tab under the Debug tab to display all known probe point pairs and their verification status. Use this tab to verify probe points or to remove them selectively from the list.

You can create Probe Points while viewing the logic cones of either failing compare points, passing compare points, aborted compare point, unverified compare points, or even other probe points. Simply highlight the appropriate reference and implementation nets while viewing the logic cone of a compare point. Then, right click and choose Set Probe from the menu that appears. You can also create the probe points by clicking the Set Probe Point icon in the logic cone window. The selected nets appear in a list under the Probe Points tab. The appropriate command also appears in the transcript window. You can specify probe points that include one reference net that matches several implementation nets.

In shell mode, the applicable commands and options for this feature are as follows:

```
set_probe_points ref_net impl_net
report_probe_points
remove_probe_points net | -all
report_probe_status
verify -probe
```

To specify that a pair of probe points has an inverted relationship, use the set_probe_points -inverted command with the following syntax:

To filter the probe report based on probe verification status, use the report_probe_status -status command with the following syntax:

```
report_probe_status -status pass | fail | abort | notrun
```

Alternatively, you can issue the shell commands interactively in the command window of the GUI.

Multicolor Highlighting

To change the color of objects within the logic cone design, you can use the highlighting toolbar options. To make a highlight, first select the object(s) you want to highlight. Then click the "Highlight Selected" toolbar option. This changes the color of the selected object(s) to the current color shown in toolbar.

To change the current color, click the "Next Color" toolbar option. This changes the current color to the next color in the list. To choose a specific color from the list, click the pulldown menu next to the "Next Color" toolbar option and select one of the eleven colors. To cycle the colors automatically with each highlight, click the pulldown menu and choose "Auto Cycle Colors." Each time you click the "Highlight Selected" button, the current color automatically changes to the next color in the list.

To remove highlighting from selected objects, click the "Clear Selected" toolbar option.

To remove highlighting of objects that are highlighted the current color, click the "Clear Current Color" toolbar option.

To remove all highlighting, click the "Clear All" toolbar option.

Cell Coloring

There are two modes of cell coloring in a cone schematic. In the menu bar, select one of the following modes:

Mode	Description
Standard	Standard coloring colors all cells green, except the compare point which is colored red.
Power Domain	When Power Domain coloring is selected, all cells are colored according to their power domain, except the compare point which is colored red.

Viewing, Editing, and Simulating Patterns

The Pattern view window allows you to view, edit, and simulate patterns to perform what-if analysis. You can also see the matched cone inputs and their values and the next state value at the compare point in a concise table format.

The tool automatically applies the pattern to both the implementation and reference designs and displays the state values associated with the logic cones.

From the main window, open the Pattern view window for a matched compare point as follows.

- 1. Select a compare point from one of the reports.
- 2. Click View > Show Patterns or right-click on a compare point and click Show Patterns.

Or, in the logic cone view window, click the Show Patterns toolbar option as shown in Figure 10-6.

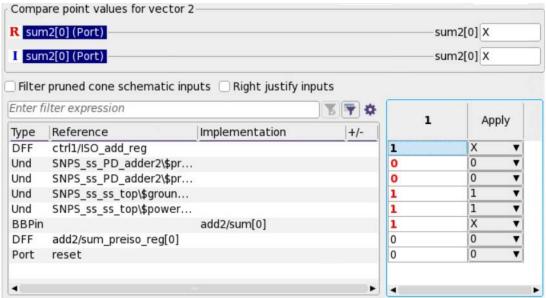
Figure 10-7 shows an example of the Pattern View window.

The reference and implementation columns list the inputs to the logic cone for each design. The patterns are shown to the right of the cone inputs. The top portion of the window shows the values at the reference and implementation compare points for the currently selected pattern.

For any given pattern, the values in red are required to make that pattern show the failure.

In the Pattern View window, the toolbar contains buttons that act as shortcuts to some menu selections that are listed in "Logic Cones" on page 10-31.

Figure 10-7 Pattern View Window



Note:

Names shown in red indicate that a constant 0 is applied to those inputs. Names shown in orange indicate that a constant 1 is applied to those inputs. You see the same color indicators in the cone schematic when you set the net coloring mode to view constants.

Patterns generated by the tool are indicated by an asterisk (*) in the column header.

In addition to viewing, if you have a Formality Ultra license you can also edit and simulate the input patterns.

To edit and simulate the patterns,

- Right-click a pattern and select Copy and Edit Pattern from the menu.
 A new column is created, and the values of the selected pattern are copied.
- 2. Select a new value for each input.
- 3. Click Apply.

The pattern is committed and simulated. The results of the simulation are displayed when you view the schematic of the cone. You can edit a tool-generated pattern repeatedly without having to copy it each time.

Debugging a Hard Verification

A verification is considered hard when,

- The Formality tool cannot completely verify all compare points
- The transcript shows no apparent progress for a long period of time
- The verification terminates due to design complexity

Usually this involves datapath intensive designs, but sometimes could involve nondatapath causes, such as cyclic redundancy check (CRC), parity generators, XOR trees, or simply very large cones of logic.

The following transcript example shows a hard verification due to the tool being hung or stuck:

```
***************** Matching Results ******************
32 Compare points matched by name
O Compare points matched by signature analysis
O Compare points matched by topology
96 Matched primary inputs, black box outputs
O(0) Unmatched reference(implementation) compare points
0(0) Unmatched reference(implementation) primary inputs, black box
outputs
15(0) Unmatched reference(implementation) unread points
******************
Status: Verifying
Status: Matching Hierarchy
Status: Verifying...
..... 0F/0A/3P/29U (9%) 04/14/09 04:17 417MB/1747.02sec
..... 0F/0A/3P/29U (9%) 04/14/09 04:47 417MB/3528.72sec
...... 0F/0A/3P/29U (9%) 04/14/09 05:17 417MB/5665.33sec
...... 0F/0A/3P/29U (9%) 04/14/09 05:47 417MB/7379.30sec
```

From reading the transcript example you can see that there are unverified compare points if the verification is interrupted.

The next transcript example extract shows a hard verification because the tool is terminating because of complexity:

```
Compare point mix[23] is aborted
   Compare point mix[24] is aborted
   Compare point mix[25] is aborted
   Compare point mix[26] is aborted
   Compare point mix[27] is aborted
   Compare point mix[28] is aborted
   Compare point mix[29] is aborted
   Compare point mix[30] is aborted
   Compare point mix[31] is aborted
********************** Verification Results *****************
Verification INCONCLUSIVE
(Equivalence checking aborted due to complexity)
Reference design: r:/WORK/test
 Implementation design: i:/WORK/test
 3 Passing compare points
 29 Aborted compare points
 0 Unverified compare points
______
Matched Compare Points BBPin Loop BBNet Cut Port DFF LAT TOTAL
______
Passing (equivalent) 0 0 0 0 0 0 3
Failing (not equivalent) 0 0 0 0 0 0
Aborted Hard (too complex)0 0 0 0 29 0 0
```

The basic hard verification debugging flow involves examining the transcript for obvious problems, creating a list of hard points, finding the cause of hard points, and finally attempting to resolve the hard points.

In the extract of the transcript, you can see that verification was attempted on all compare points and that there were 29 aborted compare points causing the hard verification failure due to complexity.

Checking the Guidance Summary

First look at the guidance summary, which is displayed in the transcript. Note that it is also available on demand using the report_guidance -summary command.

Initially, you should look for any high-level issue. For instance, the guidance summary that follows shows that all guide datapath commands were rejected, due to the architectural netlist command being rejected. This is a good indication of a global issue that needs to be addressed.

The guidance summary does not show a global issue, as illustrated in the following example extract:

Command		Accepted		Unsupported	Unprocessed	Total
architecture_	_netlist:	1	0	0	0	1
datapath	:	79	22	0	0	101
environment	:	1	0	0	0	1
instance_map	:	93	0	0	0	93
merge	:	87	14	0	0	73
multiplier	:	2	0	0	0	2
replace	:	758	0	0	0	758
scan_input	:	2	0	0	0	2
uniquify	:	2	0	0	0	2
ununiquify	:	2	0	0	0	2

Some or all of these rejections can contribute to the resulting hard verification. You need to investigate only those rejections that might have lead to the hard verification. To do this, you must first investigate the hard points.

Creating a List of Hard Points

There are two Formality report commands which are used to determine your current hard points in the verification.

- report unverified points
- report_aborted_points

An example of this is shown as follows:

```
fm_shell (verify) > report_unverified_points

21 unverified compare points:
        21 unverified because of interrupt or timeout
        0 unverified because failing point limit reached
        0 affected by matching changes

Ref DFF        r:/WORK/dp/angle_reg[10]

Impl DFF        i:/WORK/dp/angle_reg[10]

Ref DFF        r:/WORK/dp/angle_reg[11]

Impl DFF        i:/WORK/dp/angle_reg[11]
```

Determining the Cause of Hard Points

Use the <code>-aborted</code> and <code>-unverified</code> options of the <code>analyze_points</code> command to examine the aborted and unverified points, respectively. The command also identifies and diagnoses different sources of don't-cares for failing, hard, and unverified compare points. The command displays messages that might help you resolve this issue.

The analyze_points command identifies don't-care (X) sources on the fanin of the selected failing, hard, and unverified compare points. It displays the RTL source line number responsible for the X source in the compare point and the type of RTL X source. The following example shows an analysis report generated using the analyze_points command.

```
fm shell (verify)> analyze points -aborted
Found 1 RTL Source of X
An X Source is caused due to direct assignment by the following
line in the RTL code
Over-indexing in
       /u/test/test_a/test.v:37
reorder.v:147
Propagates 'X' to the ref compare point in the cones for 2 compare
point(s):
r:/WORK/top/out[2]
r:/WORK/top/out[9]
-----
Analysis Completed
Compare points that are aborted or unverified (due to X sources) may
potentially be resolved by re-writing the RTL constructs by eliminating
the X sources.
```

For additional information, use the report_svf_operation command to determine if there is any relevant rejected datapath guidance in the cones of the hard points. Always start with the -summary option to get a higher level view of what is contained in the cone.

Operation	Line	Command	Status
		1	
6	55	replace	accepted
7	72	replace	rejected
8	91	transformation_merge	rejected
10	164	boundary	accepted
11	440	constraints	accepted
13	453	datapath	accepted
15	462	replace	accepted
16	515	boundary	accepted
17	865	constraints	accepted
19	878	datapath	accepted

Often, guide commands are dependent upon the acceptance of previous guide commands; in this example, the rejection of the guide replace command is investigated first.

For more detailed report about the two guide commands, use the <code>-status</code> rejected option without the <code>-summary</code> option, or use the <code>report_svf_operation</code> command to report based on the specific guide command number.

Analyzing Fanin Logic Cones of a Hard Compare Point

Fanin logic cones can be analyzed by specifying objects using the analyze_cones command. This command can be used only in the match or verify mode.

An inconclusive verification can occur from a single or combination of factors. The analyze_cones command helps in debugging inconclusive compare points by identifying the following potential causes:

- Datapath instances
- Don't-care sources
- Large XOR trees
- Unisolated power domain crossing
- Ungrouped design instances and datapath instances

The analyze_cones command generates

• Statistical information for the specified fanin cone objects in the design

- Information about the logic that drives the compare point
- Multicone reports with information about shared logic between hard compare points

The analyze_cones command reports

- Size of the fanin cone and the distribution of logic across hierarchies
- Operators in the fanin cone
- Sources of don't-care cells and their location in the RTL files
- Power domains in the fanin cone and their fanin cone sizes
- Pins of a specified instance in the fanin cone (helps in identifying potential cut points)

The analyze_cones command performs the following tasks:

Analyzes specified objects

The objects can be specified using the following options.

 Use the -type option to specify objects from either the reference or implementation design. The objects can be cells, nets, pins, and ports. You can use wildcard (*) characters to specify these objects.

```
analyze_cones -filter logic -percent 1 -type <objects>
```

O Use the -unverified, -aborted, or -failing option with the -r or -i option to specify unverified, aborted, or failing objects respectively. The tool issues an error message if you use the -r and -i options together.

These options can be used only after verifying. The -unverified, -aborted, and -failing are mutually exclusive options.

Limits the number of objects to be analyzed

You can limit analyzing the number of objects

By restricting the maximum number of objects for analysis

The analyze_cones command stops analyzing the objects based on the value set for the -max_objects option. The following command analyzes the first 20 unverified objects and stops the analysis:

```
analyze_cones -filter logic -percent 1 -unverified -i -max_objects
20
```

By comparing only a few compare points

The analyze_cones command compares only one object out of the specified number of objects that is set with the -reduction_factor option. The following command

selects 1 out of every 100 (1, 1001, 2001,..., 9001) points from the list of failing objects:

analyze_cones -filter logic -percent 1 -failing -r -reduction_factor
1000

With different set of compare points

The command randomly analyzes a different set of compare points based on the value set for the <code>-start_object</code> option. The default is 0. This option avoids analyzing the same set of compare points that was used by the <code>-reduction_factor</code> option. The <code>-start_object</code> option must be used with the <code>-reduction_factor</code> option.

The following command selects 1 out of 100 (501, 1501, 2501,..., 9501) points from the list of failing objects. This is done to analyze a random set of objects when issues exist after analyzing objects using the -reduction_factor option.

```
analyze_cones -filter logic -percent 1 -failing -r -start_object 500
-reduction_factor 1000
```

Generates subtype, multicone, and summary reports

The -mutilcone option generates a combined report for all the specified objects (Example 10-3). The -summary option reports only the cone size of specified objects (Example 10-2). The -multicone and -summary options are mutual exclusive.

The -filter option (Example 10-1) generates subtype reports using the following objects:

- datapath
- dontcare
- o logic
- o pin
- o powerdomain
- o ungroup

Example 10-1 Report Using the -filter Option

```
fm_shell (match)> analyze_cones -filter logic -percent 20 r:/WORK/
st at/fit b/slice 3*/pit bit/pdout req
Report
            : cone_analysis
              -filter logic
              -percent 20
              r:/WORK/st_at/fit_b/slice_3*/pit_bit/pdout_reg
            : r:/WORK/st_at
Reference
Implementation : i:/WORK/st_at
Version
         : N-2017.09
            : Tue Jul 18 14:50:42 2017
Date
Analyzing r:/WORK/st_at/fit_b/slice_3*/pit_bit/pdout_reg
Logic
         XOR
                SEQ SIZE Percent Level Instance (Design)
     39
             0
                          44 100.00% 1 st_at (st_at)
                    5
                                       2 fit_b (st_top)
     35
             0
                           40
                              90.91%
                    5
                                       3 slice_0 (fit_slice)
     33
             0
                               86.36%
                           38
     29
             0
                    3
                           32
                               72.73%
                                        4 pit bit (pit)
             0
                    2
                           16
    14
                               36.36%
                                        5 pdout reg
(UPF_RET_SEQ_AACC_0_0_0_1...)
Analysis Completed
```

Limits the output of the report

You can determine what to view in the report by

- O Specifying the cutoff size with the -size option
- Specifying the cutoff percent with the -percent option
- Specifying the cutoff level with the -level option
- Generating the summary report

Example 10-2 Report Using the -summary Option

```
fm_shell (match)> analyze_cones -filter logic -summary r:/WORK/st_at/
fit_b/slice_3*/pit_bit/pdout_reg
44 r:/WORK/st_at/fit_b/slice_3/pit_bit/pdout_reg
44 r:/WORK/st_at/fit_b/slice_30/pit_bit/pdout_reg
44 r:/WORK/st_at/fit_b/slice_31/pit_bit/pdout_reg
44 r:/WORK/st_at/fit_b/slice_35/pit_bit/pdout_reg
44 r:/WORK/st_at/fit_b/slice_38/pit_bit/pdout_reg
```

Generating the multicone report

Example 10-3 Report Using the -multicone Option

```
fm_shell (match) > analyze_cones -filter logic -size 12 -multicone r:/
WORK/st at/fit b/slice 3*/pit bit/pdout reg
: cone_analysis
             -filter logic
              -size 12
              -multicone
              r:/WORK/st at/fit b/slice 3*/pit bit/pdout req
Reference : r:/WORK/st_at
Implementation : i:/WORK/st at
Version : N-2017.09
           : Tue Jul 18 14:53:17 2017
Analyzing r:/WORK/st_at/fit_b/slice_3*/pit_bit/pdout_reg
Analyzing r:/WORK/st_at/fit_b/slice_3*/pit_bit/pdout_reg
Analyzing r:/WORK/st_at/fit_b/slice_3*/pit_bit/pdout_reg
Analyzed 3 points
(UPF_RET_SEQ_AACC_0_0_0_1...)
     1 33.33% 3 slice_30 (st_slice)
1 33.33% 4 pit_bit (pit)
       33.33% 4
33.33% 5
     1
                             pdout req
(UPF_RET_SEQ_AACC_0_0_0_1...)
     1 33.33% 3 slice_31 (st_slice)
     1
        33.33%
                         pit_bit (pit)
        33.33%
     1
                            pdout_reg
(UPF_RET_SEQ_AACC_0_0_0_1...)
Analysis Completed
```

Alternate Strategies to Resolve Hard Verifications

In the Formality tool, when equivalence checking is inconclusive and results in one or more unverified or aborted compare points, run the following command to check for rejected datapath-related SVF guidance such as merge, datapath, and multiplier in the inconclusive logic cones, or any other recommendations that might be reported.

```
analyze_points -unverified | -aborted
```

When rejected SVF guidance is reported for datapath logic, debug the reason for the rejections, and possibly fix or work around them. When no pertinent SVF rejections are reported, the run is considered a solver-related hard verification. That is, verification of the inconclusive compare points cannot be completed using the default solver settings.

Verifying Designs Using Alternate Strategies

Alternate strategies modify solver parameters and enable solvers that are turned off by default. They provide alternate verification methodologies that might resolve hard verifications.

To use alternate strategies for verifying designs:

- 1. Run the normal verification.
- 2. Save the session file if the verification does not complete or runs for a long time without completing.
- 3. Use the session file as the starting point to use alternate strategies to incrementally verify the unverified points.

You might need to use the verification_timeout_limit variable to stop the tool if it makes no progress during verification.

Incremental verification with alternate strategies saves runtime as compared to rerunning the complete verification.

The verification_alternate_strategy_names variable lists the names of all alternate strategies and the recommended order of using an alternate strategy.

You can use the following methods to perform verifications using alternate strategies. The recommended method is the automated parallel deployment of alternate strategies.

- Verifying Designs Using an Alternate Strategy Manually
- Verifying Designs by Automated Parallel Deployment of Alternate Strategies

Verifying Designs Using an Alternate Strategy Manually

You can perform verifications

- Using an Alternate Strategy in the Existing Run
- Using Alternate Strategies With Linux Bourne Shell Scripts

The Linux Bourne shell scripts provide the capability of using alternate strategies to run verifications either in series or in parallel. Using alternate strategies to run in parallel is the quickest method when you have sufficient compute resources and multiple licenses.

Using an Alternate Strategy in the Existing Run

In the existing run, you can run alternate strategies only sequentially. Set an alternate strategy using the verification alternate strategy variable. The default is none.

prompt> set app var verification alternate strategy strategy name

Note:

The order of alternate strategies specified using the verification_alternate_strategy variable is the order followed by the Formality tool to run each alternate strategy sequentially. Some of the alternate strategies are mutually exclusive.

The point at which you enable the alternate strategies during verification affects the verification result. The tool deploys solvers during matching as well as during verification. Some alternate strategies affect the solvers deployed during matching, affecting preverification of data path blocks, these alternate strategies help resolving hard verifications. Enabling the alternate strategies during setup mode can affect both matching and verification. If enabled only during verification, the solver settings affect only the current set of unverified compare points. There are strategies that are only effective if you use them from setup mode, and the tool allows them to be specified only in the setup mode.

Note:

Avoid overusing an alternate strategy that results in a successful verification for one design. The nondefault solver settings can negatively affect the runtime on other designs. The default verification provides the optimum results. Use alternate strategies only when the default flow is not sufficient for a specific verification. Do not apply the successful strategy setting to other designs, unless those designs are also found to require a nondefault strategy.

Changes to the design might affect the ability of the <code>verification_alternate_strategy</code> settings used previously to complete the verification. It is recommended to try a default verification after making any significant design revisions, by removing the <code>verification_alternate_strategy variable setting</code>.

For more information, see the verification_alternate_strategy_names and verification_alternate_strategy variable man pages.

Using Alternate Strategies With Linux Bourne Shell Scripts

The examples of Linux Bourne shell scripts are provided for each strategy as follows:

```
path_to_fm/auxx/fm/strategy/strategy_name.sh
```

You can use the scripts as is or modify them for your environment. If you modify a script, specify either a session file or a Formality Tcl file as an argument as follows:

```
% strategy_name.sh -s post_verify.fss
```

Verifying Designs by Automated Parallel Deployment of Alternate Strategies

To use alternate strategies manually for verification, you must specify each strategy, monitor the progress of each alternate strategy, and terminate remaining verifications when one of the alternate strategies successfully completes the verification. You can automate this process by using the run_alternate_strategies command. This command

- Automatically uses all alternate strategies to run in parallel using your GRD or LSF environment
- Monitors and reports the progress of verification with each alternate strategy
- Automatically terminates all strategies when one of the alternate strategy successfully completes verification

To perform verification by automated parallel deployment of alternate strategies,

- 1. Run the normal verification.
- 2. Save the session file if the verification does not complete or runs for a long time without completing.

You might need to use the <code>verification_timeout_limit</code> variable to stop the tool if it makes no progress during verification.

3. Start the Formality shell (fm_shell) on the master shell, as follows.

```
% fm shell -f master.tcl | tee master.log
```

Note:

Before starting the Formality shell, you must edit the master.tcl file to configure the set_run_alternate_strategies_options command and the run_alternate_strategies command options to deploy automated alternate strategy worker processes.

When you use the master script, it performs the following flow. See Example 10-4.

It restores the saved session.

You can also use the -tclfile option with the run_alternate_strategies command to specify a Tcl file, instead of a session file.

The run is now in the blocked state, and you cannot execute any new commands at the fm-shell prompt.

- It writes out intermediate verification status for each log in separate directories. For example, run_alternate_strategies/m1/m1.log, run_alternate_strategies/s1/s1.log, and so on.
- By default, the master shell terminates all other strategies when one of the alternate strategy reports "Verification Succeeded".

To disable the default behavior, use the <code>-run_all</code> option with the <code>run_alternate_strategies</code> command.

It completes running all specified strategies and reports their results.

Example 10-4 A master.tcl File

```
set verification_timeout_limit 24:0:0
set_run_alternate_strategies_options -max_cores 4 -protocol SGE \
   -submit_command "qsub -P bnormal -V -l arch=glinux" \
   -num_processes 16
run_alternate_strategies -session mysession.fss
```

In Example 10-4,

- The master script deploys all 16 alternate strategies to run in parallel.
- o Each worker process times out in 24 hours.
- Each worker process uses 4 cores
- All worker processes restore the saved session from the session file, mysession.fss, before running the alternate strategies.

The set_run_alternate_strategies_options command specifies

- The following protocols with the -protocol option
 - Grid engine (SGE)
 - Load sharing Facility (LSF)
 - Portable Batch System (PBS)
 - Runtime Design Automation Network Computer (RTDA)
 - Netbatch Compute Farm (NetBatch)
 - Custom Farm (job scripts specified by users)
- The number of cores with the -max_cores option to use with each worker process
- The number of worker processes to be deployed with the -num_processes option
 Each worker process runs one alternate strategy and consumes one Formality
 license that can use up to 4 cores

In Example 10-5, the master script starts the verification using the saved session file, fm.tcl.session.verify.fss. Eight worker sessions run in parallel. Each worker session times out in 12 hours and uses 4 cores. This verification consumes 8 Formality licenses, one license for each worker session. Each license supports up to 4 cores.

Example 10-5 Verifying a Design by Parallel Deployment of Alternate Strategies

master.tcl:

set verification_timeout_limit 12:0:0
set_run_alternate_strategies_options -max_cores 4 -protocol SGE
-submit_command "qsub -V -P bnormal -cwd -l
arch=glinux,os_bit=64,mem_free=10G -pe mt 4" -num_processes 8
run_alternate_strategies -session fm.tcl.session.verify.fss
exit

dg:302 /users/testing_alternate_strategies/mytest> fm_shell -f
master.tcl |tee -i master.log

Formality (R)
Version N-2017.09 for linux64 - Aug 07, 2017
Copyright (c) 1988 - 2017 Synopsys, Inc.

This software and the associated documentation are proprietary to Synopsys, Inc. This software may only be used in accordance with the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, or distribution of this software is strictly prohibited.

Loading db file '/u/formal/nightly/synopsys/libraries/syn/gtech.db'

set verification_timeout_limit 12:0:0
12:0:0

set_run_alternate_strategies_options -protocol SGE -submit_command
"qsub -V -P bnormal -cwd -l arch=glinux,qsc=m,os_bit=64,mem_free=10G
-pe mt 4" -num_processes 8

 $run_alternate_strategies$ -session fm.tcl.session.verify.fss Job created, status 0

Current time: Tue Aug 8 11:31:26 2017

______ Strategy Job State Verification Status ______ k1 Not Run Not Available Not Run Not Available s2 Not Available s3 Not Run 13 Not Run Not Available s8 Not Run Not Available Current time: Tue Aug 8 11:33:26 2017

Strategy Job State Verification Status

kl Running Not Available
s2 Running Not Available
s3 Running Not Available
l3 Not Run Not Available
s8 Not Run Not Available

Current time: Tue Aug 8 11:37:26 2017

======================================	Job State	Verification Status		
=======================================	=======================================	=======================================		
k1 08/8/17 11:36 2084MB/726	Running .55sec	0F/0A/7304P/529U (93%)		
s2 08/8/17 11:36 2003MB/725	Running .77sec	OF/OA/7080P/753U (90%)		
s3 08/8/17 11:36 1502MB/636	Running .89sec	OF/OA/7491P/342U (95%)		
13	Not Run	Not Available		
s8	Not Run	Not Available		
Current time: Tue Aug 8	11:38:26 2017			
=======================================	=======================================	=======================================		
Strategy	Job State	Verification Status		
=======================================	=======================================	=======================================		
k1	Running	OF/OA/7609P/224U (97%)		
08/8/17 11:37 1299MB/854				
s2	Running	0F/0A/7108P/725U (90%)		
08/8/17 11:37 2058MB/974		0= /0= /50225 /0= /1000)		
S3	Completed	OF/OA/7833P/OU (100%)		
08/8/17 11:37 1299MB/692 13	Not Run	Not Available		
±3 s8	Not Run	Not Available		
Current time: Tue Aug 8		NOC AVAITABLE		
current time. Tue Aug o	11.30.20 2017			
Strategy	Job State	Verification Status		
======================================				
k1	Killed	Matched (OF/		
0A/7609P/224U (97%) 08/		· ·		
s2	Killed	Matched (OF/		
0A/7108P/725U (90%) 08/		· ·		
s3	Completed	Succeeded (OF/		
0A/7833P/0U (100%) 08/8/17 11:37 1299MB/692.62sec)				
13	Killed	Not Available		
s 8	Killed	Not Available		

Alternate strategies run completed successfully. exit

The run_alternate_strategies command will create a sub-directory "formality_alternate_strategy" where it creates and runs scripts using each alternate strategy.

```
dg:331 /users/testing_alternate_strategies/mytest/ formality_alternate_strategy> ls ./ ../ commlogs/ k1/ k2/ l1/ l2/ l3/ o2 s1/ s10/ s2/ s3/ s4/ s5/ s6/ s7/ s8/ s9/
```

11

Using Formality Ultra

The Formality Ultra tool is an extension to the Formality equivalence-checking solution. The tool provides interactive commands for analysis, modification, and verification of an implementation design during the ECO cycle. The Formality Ultra tool allows you to implement functional ECO changes with minimum effect on timing and layout.

After modifying the implementation design, you can verify only the modified parts of the design using the Formality Ultra tool, and rapidly iterate through the edit-reverify cycle.

Note:

To use Formality Ultra features in version M-2016.12 and later releases, you must have a Formality-Ultra license.

To use Formality Ultra features in previous releases, you must use the Formality-Advanced license.

This chapter describes how to use the Formality Ultra tool in the following sections:

- The Formality Ultra Flow
- Analyzing Differences Between the RTL and the Netlist
- Modifying the Implementation Design
- Verifying ECO Modifications
- Exporting ECO Modifications
- Integration With Verdi nECO

- Integration With the IC Compiler Tool
- RTL Cross-Probing

The Formality Ultra Flow

When an ECO is introduced into a design, the RTL is modified for the functional design changes. The modifications must be verified through simulation and then applied to the netlist and layout. To minimize the reprocessing effort, you do not want to resynthesize the RTL and rerun the entire design steps to re-create the layout. You want to be able to introduce changes directly into the existing netlist and bypass the processing steps.

The Formality Ultra tool enables you to analyze the differences in the existing netlist that were introduced by the ECO, to capture the modifications, and to verify the netlist changes directly in Formality. You can also create an ECO file that can be used by the IC Compiler tool to introduce the changes into the layout.

Note:

Before you use the Formality Ultra tool to verify the changes made to the RTL, modify the existing SVF file to account for the RTL changes introduced in the ECO. For information about modifying the SVF file, see "Verifying ECO Designs" on page 9-17.

Figure 11-1 illustrates the Formality Ultra flow. In the example, RTL' refers to a modified RTL and SVF' to an SVF file that is modified correspondingly.

SVF Netlist RTL' SVF' Мар Verify Verify RTL' and netlist to locate the modifications in the RTL' Failing Session Failing points indicate the modified areas in the RTL' Points File Find Find nets in the netlist that are Equivalent Nets equivalent to specified nets Also creates a backup of the Modify the unedited design Implementation Design Verifies only the specified compare points verify edits Undo Edits NO Success Removes compare points set using the verify_edits command YES Commit More YES Verifies the Edits Edits complete design NO Export remove_verify Modifications Verify points -all write edits

Figure 11-1 The Formality Ultra Flow

Analyzing Differences Between the RTL and the Netlist

To understand the effect of your RTL changes, inspect and analyze areas in your design using any of the following methods:

- Comparing versions of the RTL using the Linux diff command
- Using the Formality GUI to view the design schematics
- Analyzing the failing points from the ECO verification of the modified RTL against the original netlist

After locating areas in your design where RTL changes have introduced functional differences, use the find_equivalent_nets command to find the corresponding areas in the implementation netlist.

Note:

Automated setup, performed during match, modifies the RTL reference design based on optimizations and name changes made by the Design Compiler tool. The modifications to the RTL might change the name of the signal from the original RTL value.

You must manually translate names between original RTL files and the modified reference design.

Generating a List of Failing Points

You can use the failing points to identify the parts of the netlist to modify. To generate a list of failing points, verify the modified RTL against the original netlist.

Example 11-1 is a script that verifies the modified RTL against the original netlist. In this script, the verification_effort_level variable is set to low to find the expected failures due to ECO modifications.

In Example 11-1, the <code>verification_failing_point_limit</code> variable is set to 0 to find all failures. In most cases, the failing points are found using a low effort level. Inconclusive or unverified compare points are usually resolved when the variable is set to a higher effort level.

Example 11-1 Verification of a Design With an ECO

```
# ECO RTL vs original gates
# Use the modified SVF file
set_svf eco_change.svf timer.svf
# Read designs into Formality
read_db library.db
read_verilog -r rtl_new/timer.v
set_top timer
read_ddc -i timer.ddc
set_top timer
# Find all failures and use low effort to run faster
set verification failing point limit 0
set verification_effort_level low
verify
# Save information for later use
report_failing_points -list > failing_points.rpt
save session initial ECO
```

Finding Equivalent Nets

The find_equivalent_nets command identifies nets in the netlist that are logically equivalent to the specified nets in the RTL. This might be useful to find the location of ECO regions in the implementation netlist. The find_equivalent_nets command can be run only after the match command.

The find_equivalent_nets command finds equivalent nets of both polarities: noninverted and inverted equivalences.

Example 11-2 shows how to use the find_equivalent_nets command to report nets in the netlist that are equivalent to r:/WORK/core/u_crcval/ready[0]. The equivalent nets in the implementation design have inverted equivalences (-).

Example 11-2 Finding Equivalent Nets

```
fm_shell > find_equivalent_nets r:/WORK/core/u_crcval/ready[0]
--- Equivalent Nets:
    Ref    Net + r:/WORK/core/u_crcval/ready[0]
    Impl    Net - i:/WORK/core/u_crcval_crc_out_reg_1_/D
    Impl    Net - i:/WORK/core/U66/X
    Impl    Net - i:/WORK/core/n4
```

By default, the find_equivalent_nets command finds equivalent nets in all logic cones. To find equivalent nets in a specific logic cone.

```
find_equivalent_nets -nets [find_region_of_nets compare_point] nets
```

To search for equivalent nets in the fanin of a net, use the <code>-fanin</code> option. When an equivalent net is not found, this option can find other nets in the fanin that might help in creating an equivalent net.

Example 11-3 shows how to find equivalent nets in the fanin of the stall_date[3] net and limit the search to the nets of the r:/WORK/core/state[0] compare point.

Example 11-3 Finding Equivalent Nets in the Fanin of a Net

For more information about the find_equivalent_nets and find_region_of_nets commands, see the command man pages.

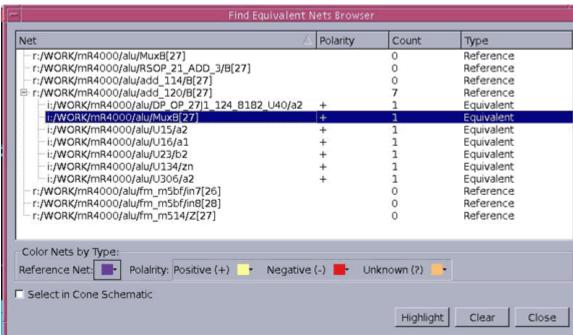
Using the GUI to Find Equivalent Nets

You can also use the Formality Ultra GUI to identify nets in the netlist that are logically equivalent to specific nets in the RTL.

To find an equivalent net in the netlist,

- In the logic cone view, select a reference signal.
- Choose ECO > Find Equivalent Nets. The Find Equivalent Nets Browser is displayed as shown in Figure 11-2.

Figure 11-2 Find Equivalent Nets Browser



You can also find equivalent nets in a specific logic cone and in the fanin of a net by choosing various options of the ECO menu.

Modifying the Implementation Design

The Formality Ultra tool enables you to modify the implementation design using Tcl based editing commands. Modify the design in the setup, match, or verify modes using the Formality Ultra edit commands. The Formality Ultra edit commands allow you to change the behavior of a design by creating, removing, connecting, or disconnecting design objects. The commands are compatible with the Design Compiler and IC Compiler commands of the same name. In Formality, these commands are extended to allow easier capture of ECO intent. See "Using High-Level Editing Commands" on page 11-11.

The Formality Ultra tool can also highlight the modifications in the implementation design schematics to enable you to check if they are modified as intended.

Use the following commands to edit the implementation design. The edit commands are available in the preverify, match, and verify modes only for designs created with the edit_design command.

- create_cell [cell_list] ref_name [-connections pin_connection_list]
- create_net [net_list] [-power | -ground] [-pins pin_list]
- create_port [port_list] [-direction in|out|inout]
- create_primitive [cell_list] type [-size size] [-connections pin_connection_list]
- remove_cell cell_list | -all
- remove_net [-hier] net_list | -all
- remove_port port_list
- connect net net pin list
- connect_pin -from pin_or_port -to pin_or_port_list
- disconnect_net net pin_list | -all
- change_link cell_list design_name [-force]

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

The Formality tool uses the following commands to define the design context:

- current_design [container:/library/design]
- current_instance [instanceID]

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

When you edit a design, the tool creates a backup of the unedited implementation design. You can view the backup design by selecting the View Backup Design GUI command in the menu of the edited design schematic window.

After the design is edited, match and verify the ECO modifications in the implementation design. If the verification is not successful or if the edits affect an unintended section of the design, you can undo the edits. If the verification is successful, export the edits to a Tcl file. If further modifications are required, commit the edits so that future undo_edits commands do not undo the desired edits.

For information about how to match and verify the modifications, see "Verifying ECO Modifications" on page 11-20. For information about how to export the edits to a Tcl file, see "Exporting ECO Modifications" on page 11-22.

Editing a Design in Match or Verify Modes

In the Formality Ultra tool, editing an implementation design for an ECO in match or verify modes enables you to view the logic cones of the failing points during editing. To edit an implementation design in verify or match modes,

- Verify the ECO RTL against the original netlist. This results in a failed verification, which
 is expected.
- In match or verify modes, use the edit_design command.

The command copies the original implementation design library to a design library named FM_EDIT_WORK and sets it as the current design. You can edit multiple designs in a session. The following example shows how to use the <code>edit_design</code> command.

```
(verify)> edit_design i:/WORK/timer
(verify)> current_design
i:/FM_EDIT_WORK/timer
```

Edit the design using the edit commands.

When editing the design in the FM_EDIT_WORK library, the tool creates a backup library named FM_BACKUP_FM_EDIT_WORK.

The ${\tt undo_edits}$ command copies the design in the FM_BACKUP_FM_EDIT_WORK library to the FM_EDIT_WORK library.

Note:

Edits to the FM_EDIT_WORK library are not exported using the write_edits or report_edits commands. Only the edits that are committed in the FM_WORK library are exported.

For more information about editing commands, see "Using High-Level Editing Commands" on page 11-11.

• Perform multipoint verification using the verify_edits command.

Verify the edits using the <code>verify_edits [-all]</code> command. The command copies the edited designs created by the <code>edit_design</code> command into the original design library, returns to setup mode, and verifies the compare points that are affected by the edits. This verification checks that the previously passing points affected by the edits are still passing and are unaffected by the design edits. You can specify additional compare points to verify using the <code>set_verify_points</code> command.

If the verification is successful, return to setup mode and verify the complete design, as shown in the following example:

```
$ setup
$ remove_verify_points -all
$ verify
```

Applying Edits

The apply_edits command moves the contents of the FM_EDIT_WORK library to the FM_WORK library. The FM_EDIT_WORK library remains unchanged. This command is only used in setup mode.

The commit_edits command deletes the FM_BACKUP_FM_EDIT_WORK library and applies the edits to the design in the FM_EDIT_WORK library.

Discarding Edits

The discard_edits command removes all copies of the design created by the edit_design command. The edits that are not applied using the apply_edits command are also discarded. This command is available in setup, match, and verify modes.

Note:

Using the apply_edits or verify_edits commands copies the FM_EDIT_WORK library back to the WORK library but retains the FM_EDIT_WORK library.

Displaying Edits

You can verify the applied edits by highlighting them in the design schematics. To display the designs that are edited using the edit_design command, select the original design in the hierarchy browser and then choose ECO > View Edit Design. The tool displays the edited copy of the original design. Choose ECO > Color Edits to highlight the changes to the implementation design.

Using High-Level Editing Commands

When performing ECO changes, there are commonly-encountered scenarios that require repetitive, verbose connections using the basic commands. In the Formality tool, the basic commands are extended to ease implementation of these commonly performed edits. These extensions differ from the IC Compiler and Design Compiler edit commands. However, they are converted to compatible commands when post-processed by the Formality Ultra tool.

This section describes how to use the Formality Ultra high-level editing commands to insert modifications using the Formality Ultra tool.

Example 11-4 adds a new AND gate to the current design using basic commands. You can use the Formality Ultra high-level commands instead of the commands in Example 11-4 to achieve the same goal. The high-level commands are shown in the examples in the following sections.

Example 11-4 Adding an AND Gate to a Design

```
# Set the current design
current_design i:/WORK/mCntrl_test_1

# Create an AND gate
create_cell eco_andgate an02d2

# Disconnect original fanout pin from its driver
disconnect_net n10 U59/a2

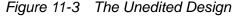
# Create a new net
create_net new_net

# Reconnect it to the output of the AND gate
connect_net new_net {eco_andgate/z U59/a2}

# Connect AND gate to original net driver
connect_net n10 eco_andgate/a1

# Connect AND gate to new control signal
connect net Op[5] eco andgate/a2
```

Figure 11-3 shows the original design and the net to be changed is highlighted.



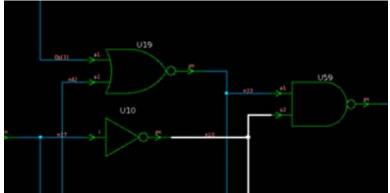
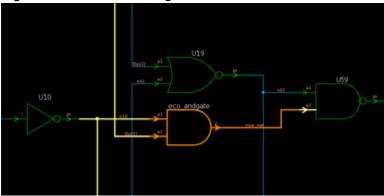


Figure 11-4 shows the modified design with a new AND gate.

Figure 11-4 Edited Design to Add an AND Gate



Disconnecting Pins Automatically

The Formality Ultra <code>connect_net</code> command automatically disconnects a pin that is connected to a net. This eliminates the need to use the <code>disconnect_net</code> command to specifically disconnect the net before reconnecting it. Example 11-5 shows the modified <code>Example 11-4</code> with the redundant <code>disconnect_net</code> command removed.

Example 11-5 Remove disconnect_net Command

```
current_design i:/WORK/mCntrl_test_1
create_cell eco_andgate an02d2
# disconnect_net n10 U59/a2
create_net new_net
connect_net new_net {eco_andgate/z U59/a2}
connect_net n10 eco_andgate/a1
connect_net Op[5] eco_andgate/a2
```

Connecting Pins When Creating Cells

The pins of a new cell have to be connected to a net. The Formality Ultra <code>create_cell</code> command connects pins when a cell is created. Example 11-6 shows the modified Example 11-5 with the <code>connect_net</code> commands replaced with the <code>create_cell-connections</code> command.

Example 11-6 Using the create_cell Command to Connect Nets

Note:

Do not use spaces between the pin name, the "=" delimiter, and the net name.

Using High-Level Commands with Hierarchical Designs

When working with objects that span a hierarchy, it is often necessary to reference an object that is at a higher hierarchical level. Actions occurring in the current design cannot refer to objects at a higher level of hierarchy because the hierarchical context is not specified.

The Formality Ultra <code>current_instance</code> command defines the current context as an instance path at a lower hierarchical level than the current design. Example 11-7 shows how to set the current context to the b design below the top design, and connect to a net above the b design.

Example 11-7 Setting the Current Context

Objects in the Formality editing commands can be referred to using the instance path name such as i:/WORK/top/m/b/myAndGate, or a relative path name such as myAndGate. When

the instance path name is not specified, the tool uses the <code>current_design</code> and <code>current_instance</code> commands to create the full path name:

```
[current_instance]/relative_object_pathname
```

Note:

Not all Formality commands support current instance.

When you set the <code>current_design</code> command to a design name, the current instance is set to the current design. For example, if the <code>current_design</code> command is set to i:/WORK/mid, the current instance is also i:/WORK/mid.

When you set the current_instance command to a full path name, the current design is set to the top-level design in the instance path name. For example, if the current_instance command is set to i:/WORK/top/m/b, the current design is i:/WORK/top.

The current instance can be set to a path name relative to the existing current instance. Setting the current_instance command to a relative path does not affect the existing current design. For example, if the current instance is i:/WORK/top/m/b, and the current_instance command is called with the relative path "..", the current instance is set to i:/WORK/top/m.

Port Punching Across Hierarchies

Often new connections span several levels of design hierarchy. To connect design objects across hierarchies using basic commands, you must create ports and pins, and connect them to nets.

The Formality Ultra connect_net and disconnect_net commands allow the specification of objects across hierarchies, as shown in Example 11-8.

In Example 11-8, the net net1 is in cell m, one level below the top design. The <code>connect_net</code> command connects the net1 net to the gate1/IN gate at the top level and to the gate2/OUT gate, one level below the m cell in the b cell by creating the necessary ports and net segments.

Example 11-8 Connecting Nets Across Hierarchical Levels

Formality Ultra autonaming ensures that the names of the created objects are unique.

Default Names for Nets, Cells, and Ports

If an instance name is not specified when you use the <code>create_cell</code>, <code>create_net</code>, and <code>create_port</code> commands, the commands generate an unique instance name. The Formality Ultra create commands return the names of the created objects, including the full path name.

The generated names are of the following form:

prefix_type_number

- prefix is controlled by the current_prefix command. The default is FM.
- type is CELL, PORT, or NET depending on the command used to create the name.
- number is the lowest integer to ensure that the name is unique.

Example 11-9 creates a net without a user-specified name. The message indicates that the a2 pin is disconnected from the original n10 net. The name of the new net is FM_NET_1.

Example 11-9 Generating a Name for a New Net

```
fm_shell (setup)> create_net -pins U59/a2
Info: Disconnecting pin 'a2' from net 'n10'.
i:/WORK/mCntrl test 1/FM NET 1
```

This is convenient because the return value of the create commands can be used where the object name is needed.

High-Level Commands to Add an AND Gate

Using the high-level editing commands, Example 11-6 is updated to the commands in Example 11-10.

In Example 11-10, the n10 net is automatically disconnected from U59/a2 when it is connected to the new net. The new net is not user-specified. The output of the <code>create_net</code> command is used directly for the connection of the new eco andgate/z pin to U59/a2.

Example 11-10 High-Level Editing Commands to Add an AND Gate

Using Edit Files

Tcl editing commands can be directly entered on the command line. However, it is convenient to save the commands that are used to edit a design in a file, and then source the file. To change the edits, you can then edit the file and re-source it. The undo_edits command reverts the changes made to the design. The commit_edits command commits the changes and you can proceed with a new set of edits. Future undo_edits commands do not undo commands saved by the previous commit_edits command.

This section describes how to create and load edit files. It also describes how you can undo, commit, and report the modifications to a netlist.

Creating an Edit File

To create an edit file, define a text file that contains Tcl editing commands.

Example 11-11 shows the contents of an edit file that creates and inserts an XOR cell into the design named core.

Example 11-11 Contents of an Edit File

Loading Edit Files

Use the source command to load an edit file. The command loads the edits in the file and creates a backup copy of the affected designs. The edits are temporary and can be undone unless the edits are committed to the design using the commit_edits command. The syntax is source filename. The filename argument specifies an edit file.

Note

In the Formality Ultra GUI, this is the load edits command.

Undoing Edits

Use the undo_edits command to undo the edits or modifications to the implementation design. The command reverts the edit commands used to modify the design. The design is reverted to the start of editing or to the previous commit edits command.

Note:

Individual edit commands cannot be undone; you can only undo the entire collection of edit commands performed since the last commit_edits command.

Committing the Edits to the Design

Use the <code>commit_edits</code> command to save modifications to a part of the implementation design before proceeding to edit another part. After running the <code>commit_edits</code> command, the previous edits cannot be undone.

After committing edits, you can start a new round of ECO editing. The future undo_edits commands do not undo the committed modifications.

Reporting the Edits

Use the report_edits command to report the edit commands that are used to modify the implementation design. The command reports the edit commands used in the current session. Use the record_edits command to enable and disable recording of the edit commands in a session. All edit commands are reported except those that are used when recording was disabled or those that are reverted using the undo edits command.

Displaying Modifications to the Design

Use the <code>compare_edits</code> command to list the parts of a netlist that are changed by the edit commands. The <code>compare_edits</code> command reports the pins, ports, nets, and cells that are added, removed, or changed using the edit commands.

Example 11-12 shows an example report.

Example 11-12 Comparing Edits

```
fm_shell (setup)> compare_edits
ADDED elements:
        NETS
                i:/WORK/bot/new
REMOVED elements:
        PINS
                i:/FM_BACKUP_WORK/mid/b1/bo1
                i:/FM_BACKUP_WORK/mid/b2/bo1
        PORTS
                i:/FM_BACKUP_WORK/bot/bo1
CHANGED elements:
        NETS
                i:/FM_BACKUP_WORK/bot/bo1
                i:/FM_BACKUP_WORK/mid/mo1
                i:/FM_BACKUP_WORK/mid/mo2
                i:/WORK/bot/bol
                i:/WORK/mid/mol
                i:/WORK/mid/mo2
        CELLS
                i:/FM_BACKUP_WORK/mid/b1
                i:/FM_BACKUP_WORK/mid/b2
                i:/WORK/mid/b1
                i:/WORK/mid/b2
```

Using the GUI to Display and Highlight Edits

Using the Formality Ultra GUI, you can highlight the modifications to a design to check that the design is edited as intended. Highlighting the GUI schematic pages is a visual representation of modifications to the implementation design.

You can view both the original and the modified designs. Choose the ECO > View Backup Design to display the backup design. If either design is displayed, choose the ECO > Color Edits to highlight the edits.

The modifications to nets whose connectivity has changed are highlighted in yellow, objects from the original design that are deleted are highlighted in red, and objects added to the modified design are highlighted in orange.

You can also access the <code>load_edits</code>, <code>commit_edits</code>, and <code>undo_edits</code> commands from the ECO menu. The GUI <code>load_edits</code> command is the <code>source</code> command in the script file. For more information about the commands, see the man pages.

Reporting Connectivity Errors

To check the connectivity of objects that are edited, use the report_electrical_checks command. The command reports connectivity errors that might have occurred during editing.

By default, the report_electrical_checks command reports the following errors:

- Unconnected ports
- Unconnected pins
- Output pins tied to constants
- Multiply driven pins
- Multiply driven ports
- Unread nets
- Undriven nets
- Unreachable cells
- · Uninstantiated designs

However, you can report specific errors using the report_electrical_checks command options. You can also specify a list of designs to be checked.

Example 11-13 shows how to create a flip-flop in the netlist. Note that the input D is not listed in the connections.

Example 11-13 Editing the Netlist to Add a Flip-Flop

```
create_cell all_stop_clocked_reg STN_FDPQ_1 -connections \
   [list CK=clock Q=[create_net -pins U32/B1]]
```

Example 11-14 shows how to use the report_electrical_checks command to report the unconnected input D.

Example 11-14 Reporting Connectivity Errors

For more information about the report_electrical_checks command, see the command man page.

Verifying ECO Modifications

To verify ECO Modifications,

- 1. Edit the implementation design.
- 2. Verify the edited implementation design against the modified RTL.

Note:

Make sure you use the modified SVF file for verification.

3. Use the verify_edits command to verify the compare points affected by the edits.

Note:

Verify only the compare points without verifying the complete design.

You can specify additional compare points using the set_verify_points command and then use the verify_edits command to verify all compare points.

To list the compare points that are relevant to the modifications, use the find_compare_points command. This command returns a list of downstream compare points affected by the modified design objects. Use this command also to generate a list of compare points that are affected by ECO modifications.

For more information about the verify_edits, set_verify_points, find_compare_points, and restore_session commands, see the command man pages.

Example 11-15 shows how to verify the ECO modifications.

Example 11-15 Verifying ECO Modifications

```
# Start from restored session of RTL with ECO to original design
restore_session initial_ECO

# Modify design and match compare points
setup
source edits.tcl

# Add additional compare points using the set_verify_points command
verify_edits

# If all failing points are fixed, and the verification is successful,
# remove all verify points and do a full verification
if [ string equal $verification_status "SUCCEEDED" ] {
    remove_verify_points -all
    verify
}
```

After verifying the specified compared points, the tool

Displays the following message:

ATTENTION: Only a subset of the compare points will be verified. Use remove verify points -all to do a full verification.

 Reports a summary of the verification, which includes the number of compare points that are specified for verification. Example 11-16 shows a summary report of a multiple point verification.

Example 11-16 Summary Report of Verification

Verification SUCCEEDED ATTENTION: Only a subset of the compare points were verified. Use remove_verify_points -all to do a full verification. ATTENTION: synopsys_auto_setup mode was enabled. See Synopsys Auto Setup Summary for details. ATTENTION: RTL interpretation messages were produced during link of reference design. Verification results may disagree with a logic simulator. _____ Reference design: r:/WORK/top Implementation design: i:/WORK/top 3 Passing compare points Matched Compare Points BBPin Loop BBNet Cut Port DFF LAT TOTAL Unverified Not Compared *******************

The tool does not verify compare points that are specified using the set_dont_verify_points command, even if they are specified using the set_verify_points command.

For information about the set_dont_verify_points and set_verify_points commands, see the command man pages.

Editing designs for ECO often requires multiple iterations of using the edit, match, and verify edits commands.

Verifying ECO Modifications to Designs With UPF

When you apply ECO modifications to designs with UPF, changes to the RTL and netlists require corresponding modifications to the UPF files.

The ECO edits that modify components of the circuit that are referenced, modified, or created by UPF might not be visible or might not have the same names in the IC Compiler tool view of the circuit.

Verifying ECO Modifications to Retimed Designs

Retiming is the only automated setup operation that modifies the implementation design. You can verify retimed designs, unless retiming has flattened part of the hierarchy.

Verification of retimed designs might result in some flattening of the hierarchy in the implementation design. Edit commands in these areas of the design might not translate to the unflattened design in IC Compiler. Edits to the netlist in the vicinity of retiming optimizations might not be compatible with the IC Compiler netlist.

Reporting Verify Points

Use the report_verify_points command to report the compare points specified using the set_verify_points command.

Removing Verify Points

After verifying the specified compare points in the ECO modification, use the remove_verify_points command to remove specific compare points in the list.

Use the remove_verify_points -all command and verify the complete design.

Exporting ECO Modifications

After verifying the ECO modifications to a netlist, you can export the edit commands to a Tcl file. Use the Tcl file to apply modifications to the design in the Design Compiler and IC Compiler tools.

To write a Tcl file containing the edit commands, use the write_edits command. The command writes the edits into a Tcl script that is compatible with the Design Compiler and IC Compiler tools. By default, the name of the Tcl script file is default_edits.tcl.

Note:

All edit commands are written to the file except those that are used while recording is turned off or those that are reverted using the undo edits command.

By default, the Formality Ultra tool records the edit commands that are used to modify the implementation design for an ECO. You can disable the recording using the record_edits command.

When you use the <code>-off</code> option with the <code>record_edits</code> command, the edit commands are not included in the output of the <code>write_edits</code> command.

For more information about these commands, see the command man page.

In the output edit script, you can use the following Tcl variables for controlling edit commands. The Tcl variables are grouped based on the tool running the edit script. The Tcl variables are defined and used only in the edit script to run in

- The Formality, Design Compiler, IC Compiler, and IC Compiler II tools
 - o fm_edit_enable_warnings: Controls warnings to be omitted or ignored.
- The Formality tool only
 - o fm_edit_substitute_container: Allows you to apply edits made in one container to another container and controls how the edits work. When you make edits in the container "i" and you run the tool, the edit script uses the variable to change the container (impl) for which the edit applies when reading the design.
 - o fm_edit_echo_commands: Prints the edit commands from the edit script instead of executing the commands.
 - o fm_edit_use_edit_libraries: Controls where to make edits, either in the design (designs created with the edit design command) or in an edit library.

Note:

You can make edits only in the setup mode. In designs created with the edit_design command, you can make edits in setup, preverify, match, and verify modes.

- The Design Compiler and IC Compiler tools
 - o fm_edit_substitute_library: Allows you to change the library name used for referencing library cells. The Formality tool names libraries differently from the other Synopsys, so use the variable to rename the library for the edit script to handle library names. For example, use an asterisk (*) to match any library.
- The IC Compiler and IC Compiler II tools
 - o fm_edit_root_path: Allows the edit script to adjust object names and other paths when the current_instance command in the IC Compiler or IC Compiler II tool is not relative to the top-level design but is relative to a path lower in the design hierarchy.

In Example 11-17, the Design Compiler tool uses the $fm_{edit_substitute_library}$ variable for changing the technology library used in the Formality tool. The IC Compiler and IC Compiler tools use the same variable ($fm_{edit_substitute_library}$) as the Design Compiler tool along with the $fm_{edit_root_path}$ variable to specify the root path.

Example 11-17 Edit Script

```
##
## You can 'source' this file in the Formality (R), Design Compiler (R),
## IC Compiler (TM), or IC Compiler (TM) II tool.
# Set fm_edit_enable_warnings to 1 to print warnings, or set it to 0 to
# hide warnings.
global fm_edit_enable_warnings
set fm_edit_enable_warnings 1
if { $synopsys_program_name eq "fm_shell" } {
    # Set fm edit substitute container to apply edit commands from
    # one container to another container. For example, set to "i" to
    # make all edit commands apply to the "i" container.
    global fm_edit_substitute_container
         set fm_edit_substitute_container "i"
    # Use fm_edit_echo_commands to echo ECO commands to stdout (if 1)
    # or execute ECO commands (if 0 or not set)
    global fm_edit_echo_commands
         set fm_edit_echo_commands 1
    # Use fm_edit_use_edit_libraries to do the edits in edit
    # libraries (if set to 1), or directly in the main (non-edit)
    # libraries (if 0 or not set)
    global fm_edit_use_edit_libraries
         set fm edit use edit libraries 1
}
if { $synopsys_program_name eq "dc_shell" } {
    # Set fm_edit_substitute_library to apply edit commands from
    # one library to another library. For example, set to "*" to
    # make all edit commands use a wildcard for the library name.
    global fm_edit_substitute_library
         set fm_edit_substitute_library "*"
if { $synopsys_program_name eq "icc_shell" } {
    # Set fm_edit_substitute_library to apply edit commands from
    # one library to another library. For example, set to "*" to
    # make all edit commands use a wildcard for the library name.
```

```
global fm_edit_substitute_library
        set fm edit substitute library "*"
   # Set fm_edit_root_path to the top of the design hierarchy that is
   # being worked on. Use this if you are working on a part of the
   # overall design. Leave unset if you are working on the entire
   # design.
   global fm_edit_root_path
        set fm_edit_root_path "top/cell1/cell2"
}
if { $synopsys_program_name eq "icc2_shell" } {
    # Set fm_edit_substitute_library to apply edit commands from
   # one library to another library. For example, set to "*" to
   # make all edit commands use a wildcard for the library name.
   global fm_edit_substitute_library
        set fm_edit_substitute_library "*"
   # Set fm_edit_root_path to the top of the design hierarchy that is
   # being worked on. Use this if you are working on a part of the
    # overall design. Leave unset if you are working on the entire
   # design.
   global fm_edit_root_path
        set fm_edit_root_path "top/cell1/cell2"
}
```

Integration With Verdi nECO

The Formality Ultra tool integrates the manual ECO implementation functionality with the schematic editing functionality of the Verdi nECO tool. The integration enables you to

- Start the Verdi nECO tool from the Formality Ultra GUI
- Pass design schematics from the Formality Ultra tool to the Verdi nECO tool
- Highlight schematic nets and gates across the tools
- Apply ECO edits in the Verdi nECO tool and import the edits to the Formality Ultra tool for verification

Starting the Verdi nECO Tool From the Formality Ultra GUI

To start the Verdi nECO tool from the Formality Ultra GUI,

Add the following line to the Formality script

```
set fm verdi executable path-to-executable
```

This command specifies the Verdi nECO executable to use and is not required if the Verdi nECO executable is defined in the user specified path.

Choose Verdi nECO from the ECO > nECO menu

The Verdi nECO tool starts, and the Formality Ultra tool establishes a socket connection to it. The default time to establish the socket connection is one minute. When the Formality Ultra tool cannot establish a socket connection with the Verdi nECO tool, choose Connect to nECO from the ECO > nECO menu to try and establish the socket connection.

Transferring Design Schematics From Formality Ultra to VerdinECO

After starting the Verdi nECO GUI from the Formality Ultra tool, you can specify the gates and nets to transfer from the Formality Ultra tool and open in the Verdi nECO tool.

- Select a gate or net in the Formality Ultra design schematics window.
 Shift-click to select multiple gates and nets.
- Choose Add to nECO from the ECO > nECO menu.

The Verdi nECO tool opens the schematic.

Highlighting Design Objects Across the Tools

You can highlight nets and gates across the Formality Ultra and Verdi nECO schematics. You can use cross-highlighting to find equivalent nets for an ECO or for general navigation of the schematics across the tools.

To cross-highlight a gate or net selected in the Formality Ultra schematics onto the Verdi nECO schematics, choose ECO > nECO > Highlight from FM.

To cross-highlight a gate or net selected in the Verdi nECO schematics onto the Formality Ultra schematics, choose ECO > nECO > Highlight from nECO. Note that the design schematics must be displayed in the Formality Ultra GUI before highlighting.

To retain cross-highlights, choose Accumulate Highlights from the ECO > nECO menu. When this option is disabled, previous cross-highlights are cleared with each cross-highlighting command. If this option is enabled, the tool retains the previous cross-highlights and displays the specified design objects.

For information about schematic editing in the Verdi nECO environment, see the Verdi nECO user documentation.

Importing Edits to the Formality Ultra Tool

To import the edits created in the Verdi nECO session back to the Formality Ultra tool for verification, choose Import Changes from the ECO > nECO menu. Formality schematic windows are closed to make the changes to the edited designs. This also returns the Formality Ultra tool to setup mode.

To display the edits in the modified design, choose Color Edits from the ECO menu.

After you import the edits to the Formality Ultra tool, use the <code>verify_edits</code> command to verify the compare points that are affected by the edits. This improves the performance of the verification of the complete design because the matching information is reused resulting in faster multiple-point verification.

Integration With the IC Compiler Tool

The Formality Ultra tool integrates the manual ECO implementation functionality with the IC Compiler tool. Obtaining the layout information of nets and cells from the IC Compiler tool helps in determining which net or cell to use in manual ECO modifications.

The Formality Ultra integration with the IC Compiler tool enables you to

- Connect the Formality Ultra tool to an existing IC Compiler session
- Highlight schematic nets, gates, and pin information across the tools

Connecting the Formality Ultra Tool to the IC Compiler Tool

The Formality Ultra tool generates a script which you can read into the IC Compiler tool and configure for communication with the Formality Ultra tool.

In the Formality Ultra GUI, choose ECO > ICC > Write ICC Script.
 The tool generates the fm_icc_script.tcl script in the current directory. The full path to the script is displayed in the Formality Console.

- In the IC Compiler tool, source the generated script. The script establishes the connection with the Formality Ultra tool.
- In the Formality Ultra GUI, choose ECO > ICC > Connect to ICC.

Highlighting Design Objects Across the Tools

You can highlight nets and gates across the Formality Ultra and IC Compiler layout. You can use cross-highlighting to find equivalent nets for an ECO or for general navigation of the schematics across the tools.

To cross-highlight a gate or net selected in the Formality Ultra schematics onto the IC Compiler layout, choose ECO > ICC > Highlight from FM.

To cross-highlight a gate or net selected in the IC Compiler layout onto the Formality Ultra schematics, choose ECO > ICC > Highlight from ICC.

All instances of a specified Formality Ultra design object are highlighted in the IC Compiler layout and in the visible Formality Ultra logic cone schematic.

If you specify an object in a Formality Ultra logic cone schematic or in the IC Compiler layout, the Formality Ultra tool highlights a specified object and its parents in all appropriate design schematics. This is useful for traversing the design hierarchy and identifying the design containing the specified object.

For example, if a specified object is a cell i:/WORK/a/b/c/d/e and design schematic i:/WORK/a is visible, the tool highlights cell b in the schematic. If the visible design schematic design is i:/WORK/a/b, the tool regenerates the highlighting and highlights cell c.

The objects are highlighted using the color specified in the highlight menu in Formality Ultra. The highlighted objects and their color are displayed in the Formality Console.

To retain cross-highlights, choose Accumulate Highlights from the ECO > ICC menu. When this is disabled, previous cross-highlights are cleared with each new cross-highlighting command. If this is enabled, the previous cross-highlights are retained.

Note:

The Formality Ultra tool does not support cross-highlighting of cell pins or design ports.

RTL Cross-Probing

The Formality tool offers cross-probing capabilities in the GUI to check your RTL design in the RTL browser. You can cross-probe cells, pins, or ports in a design from the design or cone schematic view and examine the corresponding RTL file. You can also open an RTL file and cross-probe to objects associated with a line in the RTL file. This allows you to determine the cause of a failed verification.

Note:

To use the RTL cross-probing capabilities, you must have an elaborated design from Formality version J-2014.09-SP2 or later.

To view the RTL source of designs, ports, and cells,

- 1. In the design or cone schematic view, right-click an object.
- 2. Choose View Source.

The tool displays the RTL file and highlights the corresponding lines.

To view the schematic of an object in the RTL file,

- 1. In the Formality Console, right-click an RTL file in the list of files that are currently loaded.
- 2. Choose View File.

The tool displays the RTL source file in the RTL Browser. Lines in the RTL file that correspond to objects are highlighted in green if the file is opened in the native browser.

3. Right-click a highlighted line and choose Select Objects of Highlighted Line.

The tool displays the list of objects, and instances of the objects, in the Global Object Finder.

4. Select and object and choose View > View Object.

The tool displays the schematic of the selected object. If the design or cone schematic window is already open, the object is highlighted in white.

The tool does not allow you to

- Cross-probe from various design views to the UPF file.
- Select nets as cross-probing objects.

For more information about the Global Object Finder, see "Finding a Design Object in a Collection" in Appendix A.

12

Library Verification Mode

You can use Formality to verify a reference design with an implementation design in the process described in Chapters 3 through 10. There are, however, some procedures in Formality which do not necessarily fall in the standard flow of operation.

This chapter includes the following sections:

- Introduction
- Library Verification Mode
- Loading the Reference Library
- · Loading the Implementation Library
- Listing the Cells
- Specifying a Customized Cell List
- Elaborating Library Cells
- Performing Library Verification
- · Reporting and Interpreting Verification Results
- Debugging Failed Library Cells

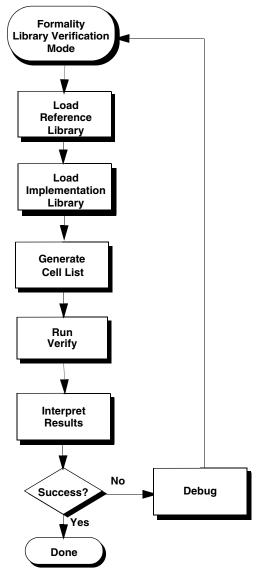
For additional information about verifying libraries, see the *Formality ESP User Guide* in the ESP documentation suite.

Introduction

This chapter assumes that you understand Formality concepts and the general process for Formality design verification. From here, it discusses compare technology (or cell) libraries.

The process flow for library verification mode is broadly similar to that of Formality itself. Figure 12-1 shows the general flow for verifying two technology libraries. This chapter describes all the steps in the library verification process.

Figure 12-1 Technology Library Verification Process Flow



During technology library verification, Formality compares all the cells in a reference library to all the cells in an implementation library. You can use the process to compare Verilog simulation and Synopsys (.db) synthesis libraries.

Library verification is similar to design verification in that you must load both a reference library and an implementation library. The principle difference is that, because the specified libraries contain multiple designs (cells), Formality must first match the cells to be verified from each library. This matching occurs when you load the reference and implementation designs. Formality then performs compare point matching and verification one cell-pair at a time.

The Formality add-on tool, Formality ESP, extends the functional equivalence checking provided by Formality. For more information about functional equivalence for full-custom transistor-level memory macros, datapath macros, and library cells, see the *Formality ESP User Guide*.

Library Verification Mode

To verify two libraries, the tool must be in the <code>library_verification</code> mode. If the tool is in one of the other modes (<code>setup</code>, <code>match</code>, <code>verify</code>, or <code>debug</code> mode), you must switch to the <code>library_verification</code> mode. Library verification is a command-line-driven process. Each time you enter (or leave) the library verification mode, Formality empties the contents of the <code>r</code> and <code>i</code> containers in readiness for a new library (or design) verification session.

To enter library verification mode, specify the library_verification command,

```
fm_shell (setup)> library_verification argument
```

You can specify one of the following options for *argument*:

- verilog verilog
- db_db
- verilog_db
- db_verilog
- verilog_pwrdb
- pwrdb_verilog
- none

The first design type in the preceding examples defines the reference library; the second type defines the implementation library. If you specify none, Formality returns to setup mode.

The fm_shell prompt changes to

```
fm_shell (library_setup)>
```

When you set this mode, Formality sets the following variable

```
set_app_var verification_passing_mode equality
```

When you exit library verification mode, Formality sets the variable back to its default, consistency.

Note:

Unsupported synthesis library formats must be translated by Library Compiler before being read into Formality.

For more information, see the library_verification command man page.

Loading the Reference Library

As with the design verification process described in Chapter 4, "Tutorial," you must specify the reference library before the implementation library.

To specify the reference library, use one of the following read commands, depending on the library format:

```
fm_shell (library_setup)> read_db -r file_list
fm_shell (library_setup)> read_verilog -r \
[-technology_library] file_list
```

The read_db and read_verilog commands have several options that do not apply to library verification. Use the read_verilog -technology_library command if you have a UDP file.

Formality loads the reference library into the r container. You cannot rename this container.

In the Formality shell, you represent the design hierarchy by using the <code>designID</code> argument. The <code>designID</code> argument is a path name whose elements indicate the container (r or i), library, and design name.

Unlike with the design verification process, you do not specify the set_top command because multiple top cells are available.

Loading the Implementation Library

Specify the implementation library as described in the previous section, with the exception of the -r argument. Instead, use the -i argument as follows:

```
fm_shell (library_setup)> read_db -i file_list
fm_shell (library_setup)> read_verilog -i [-technology_library]\
file_list
```

Formality loads the implementation library into the ${\tt i}$ container. You cannot rename this container.

After you read in the implementation library, Formality performs cell matching to generate the list of cells that are verified. Cells and ports must match by name. The cell list consists of single cell names, and each cell on it is expected to be found in the reference library. If not, it is a nonmatching cell and remains unverified.

Listing the Cells

By default, Formality verifies all library cells that match by name. You can query the default cell list before verification to confirm the matched and unmatched cells.

Specify the following command to print a list of library cells.

You must specify one of the following options:

Option	Description
-r	Prints the cells contained in the reference library.
-i	Prints the cells contained in the implementation library.
-verify	Prints the current list of cells to be verified, which could differ from the default cell list if you specified the select_cell_list command. For more information, see "Specifying a Customized Cell List" on page 12-6.
-matched	Prints a list of reference and implementation cells that match by name.

Option	Description
-unmatched	Prints the names of cells that did not match in the reference and implementation containers. This option is dynamic depending on the select_cell_list command specification.
-filter wildcard	Filters the report to include cells that match the specified wildcard. Always specify this option in conjunction with one of the preceding options.

In the rare case that the libraries contain no matching cells, follow these steps:

- 1. Return to setup mode by entering the library_verification none command.
- 2. Edit the cell names so they match.
- 3. Return to library verification mode by entering the library_verification mode command.
- 4. Reload the updated library by using the applicable read command.

Specifying a Customized Cell List

When you load the libraries by using read commands, Formality elaborates all matched cells in preparation for verification. After reporting the matched cells with the report_cell_list command, you can refine the default cell list as necessary.

To customize the default cell list, specify the following command:

You can use the following options as needed:

Option	Description
-file filename	Specifies a file that contains a list of cells to be verified.
-add cell_names	Adds the specified cells to the cell list.
-clear	Clears the cell list.
-remove cell_names	Removes the specified cells from the cell list.

This command supports wildcard characters for cell names. Enclose lists of cells in braces. For example,

```
fm_shell (library_setup)> select_cell_list {AND5 OR2 JFKLP}
fm_shell (library_setup)> select_cell_list ra*
```

As part of the debugging process, use this command to specify only those cells that previously failed verification.

Elaborating Library Cells

Formality automatically elaborates your library cells when running the <code>verify</code> command. You might want to elaborate your library cells before verification to apply constraints to specific cells. To elaborate these library cells, run the <code>elaborate_library_cells</code> command.

If you do not want to apply constraints to individual library cells, proceed directly to verification.

Performing Library Verification

Proceed to verification after refining your cell list. As with the design verification process, specify the <code>verify</code> command:

```
fm_shell (library_setup)> verify
```

Formality performs compare point matching and verification for each cell-pair as described in Chapter 8, "Matching Compare Points and Chapter 9, "Verifying the Design and Interpreting Results." However, because Formality assumes that all cell and ports match by name, compare point matching errors do not occur; for this reason, the optional match command does not apply to library verification.

As described in the man page, the \mathtt{verify} command has additional options that do not apply to library verification.

After verification, Formality outputs a summary transcript of the passing, failing, and aborted cell counts.

The following script performs library verification. This script sets hdlin_unresolved_modules to 'black box' as a precaution; generally technology libraries should not contain unresolved modules. These are not required settings. Remember that the verification_passing_mode and verification_inversion_push variables are set automatically.

```
#-----
# Sets the directories where Formality will search for files
#-----
set search_path "./db ./verilog/cells ./verilog_udp"
#-----
# Sets variables
set_app_var hdlin_unresolved_modules black_box
library_verification VERILOG_DB
#-----
# Reads into container 'r'
#-----
# Read UDP using -technology_library
read_verilog -r -technology_library {
UDP encodecod.v
UDP_mux2.v
UDP_mux2_1.v
UDP_mux2_1_I.v
UDP_mux2_2.v
}
# Read library cells
read_verilog -r {
and2A.v
and2B.v
and2C.v
aollA.v
aol1C.v
ao12A.v
bufferA.v
bufferAE.v
bufferAF.v
delay1.v
encode3A.v
xor1A.v
xor1B.v
xor1C.v
full_add1AA.v
half_add1A.v
mux21HA.v
mux31HA.v
mux41HA.v
mux61HA.v
mux81HA.v
mux21LA.v
notA.v
notAD.v
notAE.v
```

```
nand2A.v
nand2B.v
nand2C.v
nor2A.v
nor2B.v
nor2C.v
nxor3A.v
or_and21A.v
or2A.v
#-----
# Reads into container 'i'
#-----
read_db -i synth_lib.db
# Report which library cells will be verified
report_cell_list -verify
report_cell_list -matched
report_cell_list -unmatched
#-----
# Verifies libraries
#-----
verify
#-----
# Reports on passing and failing cells
report_status -pass
report_status -fail
report status -abort
#-----
# Exits
#-----
exit
```

Reporting and Interpreting Verification Results

Use the following command to report the verification results:

If you do not specify arguments, Formality reports the passing, failing, and aborted cell counts. The following table describes the use of the three options, along with an explanation of the type of status message assigned to each one during verification:

Option	Description	Status Message for Library Cell-Pairs
-pass	Returns a list of cells that passed verification.	A passing library cell-pair has all its compare points functionally equivalent.
-fail	Returns a list of cells that failed verification.	A failing library cell-pair has at least one compare point that is not functionally equivalent.
-abort	Returns a list of aborted cells.	Verification stops. This occurs when Formality reaches a user-defined failing limit. For example, Formality halts verification on a cell after 20 failing points have been found in the cell.
		In addition, any cells that fail elaboration are terminated, and a cell is terminated if Formality cannot determine whether one of its compare points passes or fails. Aborted points occur when Formality is interrupted during the verification process.

Debugging Failed Library Cells

Use the following procedure to debug failed library cells:

1. Choose a failing cell from the status report and specify the following command:

```
fm_shell (library_setup)> debug_library_cell cell_name
```

Formality reports the failing cells but retains the verification results from the last cell verified (which could be a passing cell). This command repopulates Formality with the verification data for the specified cell, which enables you to debug the cell in the current Formality session. You can specify the name of only one unique cell.

2. Specify the following command to view the failed cell's logic:

```
fm_shell (library_setup)> report_truth_table signal \
        [-fanin signal_list] [-constraint signal_list=[0|1]] \
        [-display_fanin] [-nb_lines number] \
        [-max_line length]
```

This command generates a Boolean logic truth table that you can use to check the failed cell's output signals. Often, this is sufficient information to fix the failed cell. Use the arguments as follows:

Argument	Description
-signal	Specifies the signal you want to check. For example, specify the path name as follows: r:/lib/NAND/z
-fanin signal_list	Filters the truth table for the specified fanin signals, where the list is enclosed in braces ({ }).
-constraint signal_list=[0 1]	Applies the specified constraint value (0 or 1) at the input and displays the output values on the truth table.
-display_fanin	Returns the fanin signals for the specified signal.
-nb_lines_number	Specifies the maximum number of lines allowed for the truth table.
-max_line length	Specifies the maximum length for each table line.

After fixing the cell, include only the fixed cells in the cell list and run the verify command again.

3. If further investigation is required to fix a failed cell, specify the following command:

This command generates individual Tcl scripts for each failed cell and places them in the DEBUG directory unless you specify the <code>-dir</code> option. The DEBUG directory is located in the current working directory.

4. If you attempt to view library cells in the Formality GUI, you see only a black box. As shown in the following example, the Tcl scripts direct Formality to treat the library cells as

designs and perform traditional verification. You can then investigate the failure results with the Formality GUI.

```
## --This is a run script generated by Formality library
verification mode --
set_app_var verification_passing_mode Equality
set_app_var verification_inversion_push true
set search_path "DEBUG"
read_container -r lib_ref.fsc
read_container -i lib_impl.fsc
set_reference_design r:/*/mux21
set_implementation_design i:/*/mux21
verify
```

- 5. Run one of the Tcl scripts and specify the start_gui command to view the results. When you have fixed the cell, go to each of the scripts until you have debugged them all. For information about using the GUI for debugging, see the following sections:
 - o "Debugging Using Diagnosis" on page 10-8
 - o "Schematics" on page 10-20
 - "Logic Cones" on page 10-31
 - "Viewing, Editing, and Simulating Patterns" on page 10-40
- Reverify cells that you fixed from within the GUI. You must begin a new session by reinitializing the library verification mode and reloading the reference and implementation libraries.



Querying Design Objects and Collections

Synopsys applications build an internal design database of objects and attributes that are applied to them. These databases consist of several classes of objects, including designs, libraries, ports, cells, nets, pins, clocks, and so on. Most commands operate on these objects.

By definition, a collection is a group of objects exported to the Tcl user interface. Collections have an internal representation (the objects) and, sometimes, a string representation. The string representation is generally used only for error messages.

The set of commands to create and manipulate collections is described in the following sections.

- Lifetime of a Collection
- Iteration
- Managing Collections Using Commands
- Filtering
- Sorting Collections
- Implicit Query of Collections
- The Collections Manager GUI

Lifetime of a Collection

Collections are active only if they are referenced. Typically, a collection is referenced when a variable is set to the result of a command that creates it or when it is used as an argument to a command or a procedure. For example, you can save a collection of design ports by setting a variable to the result of the get ports command:

```
fm shell> set ports [get ports *]
```

Either of the following commands deletes the collection referenced by the ports variable:

```
fm_shell> unset ports
fm_shell> set ports "value"
```

Collections can be implicitly deleted when they go out of scope. Collections go out of scope for various reasons. An example would be when the parent (or other antecedent) of the objects within the collection is deleted. For example, if a collection of ports is owned by a design, it is implicitly deleted when the design that owns the ports is deleted. When a collection is implicitly deleted, the variable that referenced the collection still holds a string representation of the collection. However, this value is useless because the collection is gone, as shown in the following example:

```
fm_shell> current_design r:/WORK/top
r:/WORK/top

fm_shell> set ports [get_ports in*]
{r:/WORK/top/in0 r:/WORK/top/in1}

fm_shell> remove_design r:/WORK/top
Removed design '/WORK/top' from container 'r'

fm_shell> query_objects $ports
Error: No such collection '_sel26' (SEL-001)
```

Iteration

To iterate over the objects in a collection, use the <code>foreach_in_collection</code> command. You cannot use the Tcl-supplied <code>foreach</code> command iterator to iterate over the objects in a collection, because the <code>foreach</code> command requires a list, and a collection is not a list. However, if you use the <code>foreach</code> command on a collection, it destroys the collection.

The arguments of the <code>foreach_in_collection</code> command are similar to those of foreach: an iterator variable, the collection over which to iterate, and the command body to apply at each iteration. Note that unlike the <code>foreach</code> command, the <code>foreach_in_collection</code> command does not accept a list of iterator variables.

The following example iterates through a collection to print the names of the objects it contains.

```
fm_shell> foreach_in_collection s1 $collection \
  {echo [get_attribute $s1 full_name]}
```

Managing Collections Using Commands

There are two categories of collection commands: those that create collections of objects for use by another command, and those that query objects for viewing. The result of a command that creates a collection is a Tcl object that can be passed along to another command. For a query command, although the visible output looks like a list of objects, the result is an empty string.

You can use the following commands to work with collections. In some cases, a command might not operate on a collection of a specific type.

- add_to_collection This command creates a new collection by adding a list of
 element names or collections to a base collection. The base collection can be the empty
 collection. The result is a new collection. In addition, the add_to_collection command
 allows you to remove duplicate objects from the resulting collection by using the -unique
 option.
- append_to_collection This command appends a set of objects (specified by name or collection) to an existing collection. The base collection is passed as a variable name, and the base collection is modified directly. It is similar in function to the add_to_collection command, except that it modifies the collection in place; therefore, it is much faster than the add_to_collection command when appending.
- remove_from_collection This command removes a list of element names or
 collections from an existing collection. The first argument is the collection to process and
 the second argument is the specification of the objects to remove. The result of the
 command is a new collection. For example,

```
fm_shell> set dports \
[remove_from_collection [all_inputs] CLK]
{r:/WORK/top/in0 r:/WORK/top/in1 r:/WORK/top/in3}
```

- compare_collections This command verifies that two collections contain the same objects (optionally, in the same order). The command returns "0" if the comparison succeeds.
- copy_collection This command creates a new collection containing the same objects in the same order as a given collection. Not all collections can be copied.

 index_collection - This command extracts a single object from a collection and creates a new collection containing that object. The index operation is done in constant time - it is independent of the size of the collection or the specified index value. Not all collections can be indexed.

• sizeof collection - This command returns the number of objects in a collection.

Filtering

You can filter any collection by using the filter_collection command. This command takes a base collection and creates a new collection that includes only those objects that match an expression.

Many commands that return collections have a <code>-filter</code> option that filters objects on the fly before adding them to the collection result. This is more efficient than obtaining the entire set of objects, then applying the <code>filter_collection</code> command. The following examples filters out all leaf cells using both methods:

```
fm_shell> filter_collection \
[get_cells *] "is_hierarchical == true"]
{r:/WORK/top/i1 r:/WORK/top/i2}

fm_shell> get_cells * -filter "is_hierarchical == true"
{r:/WORK/top/i1 r:/WORK/top/i2}
```

The basic form of a filter expression is a series of relations joined together with AND and OR operators. Parentheses are also supported. The basic relation compares an attribute name with a value through a relational operator. In the previous example, is_hierarchical is the attribute, == is the relational operator, and true is the value.

The relational operators are

- == Equal
- != Not equal
- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to
- =~ Matches pattern
- !~ Does not match pattern

The basic relational rules are

- String attributes can be compared with any operator.
- Numeric attributes cannot be compared with pattern match operators.
- Boolean attributes can be compared only with == and !=. The value can be only true or false.

Additionally, existence relations determine if an attribute is defined or not defined for the object. For example,

```
(direction == in) and defined(is_pi)
```

The existence operators are: defined and undefined.

Existence operators apply to any attribute if it is valid for the object class. See the collection man page for more information.

Sorting Collections

You can sort a collection by using the <code>sort_collection</code> command. It takes a base collection and a list of attributes as sort keys. The result is a copy of the base collection sorted by the given keys. Sorting is ascending, by default, or descending when you specify the <code>-descending</code> option. In the following example, the command sorts the ports by direction and then by full name.

Implicit Query of Collections

Commands that create collections implicitly query the collection to display the results when the command is used at the command line. The query commands are available in setup, preverify, match, and verify modes.

In setup mode after setting the top-level design, the commands enable you to explore a single design without having to load both a reference and an implementation design.

- In setup mode, the Formality design database has a uniquified hierarchy there is only one copy for each hierarchical design. Multiple instances of the design can reference the same parent design.
- In preverify, match, or verify modes, the database has a ununiquified hierarchy there is a unique copy for each instance of a hierarchical design object.

The design object query commands return lib, lib_cell, lib_pin, design, port, cell, pin, and net classes of objects in collections.

For library and design classes, a collection object represents the library or design. For other classes, a collection object represents a path to the circuit object.

The query commands create collections of multiple objects with paths starting from any design:

```
fm_shell> current_design mid
fm_shell> get_nets */n3
{r:/WORK/mid/B1/n3 r:/WORK/mid/B2/n3}
```

To find all instances of a specific object in the hierarchy, search from the top-level design as shown in the following example:

```
fm_shell> current_design top
fm_shell> get_nets -hierarchical n3
{r:/WORK/top/M1/n3
   r:/WORK/top/M2/n3
   r:/WORK/top/M1/B1/n3
   r:/WORK/top/M1/B2/n3
   r:/WORK/top/M2/B1/n3
   r:/WORK/top/M2/B2/n3}
```

In the following example, the <code>get_ports</code> command creates a collection of ports that is passed to the <code>set_constant</code> command. This collection is not the result of the primary command (<code>set_constant</code>), and when the primary command completes, the collection is destroyed.

```
fm_shell> set_constant [get_ports se*] 0
1
```

The following example shows how a command that creates a collection automatically queries the collection when it is used as interactively on the command line.

```
fm_shell> get_ports in*
{r:/WORK/top/in0 r:/WORK/top/in1 r:/WORK/top/in2}
```

The following example shows the verbose feature of the <code>query_objects</code> command, which is not available with an implicit query.

```
fm_shell> query_objects -verbose [get_ports in*]
{{port r:/WORK/top/in0} {port r:/WORK/top/in1}} {port r:/WORK/top/in1}}
```

The following example sets the iports variable to the result of the <code>get_ports</code> command. The collection persists to future commands until the iports variable is overwritten, unset, or goes out of scope.

```
fm_shell> set iports [get_ports in*]
{r:/WORK/top/in0 r:/WORK/top/in1 r:/WORK/top/in2}
```

Use the following commands to access information about attributes of design objects:

- get_attribute
- list_attributes
- help_attributes

The following attributes are supported for all classes:

Attribute	Туре	Example
object_class	string	cell
type	string	synonym for object_class
name	string	B1
full_name	string	r:/WORK/top/M1/B1
container_name	string	r
library_name	string	WORK
path_name	string	{top M1 B1} {top M1 B2}

The following attributes are supported for all classes except lib, design, and lib_cell:

Attribute	Type	Example
parent_name	string	r:/WORK/mid

The following attributes are currently defined for specific classes:

Attribute	Туре	Example
cell:		
ref_name	string	bot
cell_type		
is_techlib	string	true
is_register	string	true

Attribute	Type	Example
pin:		
direction	string	in
is_pi	string	true
is_inverted	string	true
port:		
direction	string	inout
is_pi	string	true
lib_pin:		
direction	string	inout
design:		
is_unique	string	true

The Collections Manager GUI

The Collections Manager GUI enables you to capture collections of objects, perform operations on these collections, and display them in the design schematics. The Collections Manager visualizes the schematic representation of the designs and helps in debugging. This section describes the following topics:

- Creating Collections
- Filtering Collections
- Operating on Collections
- Finding a Design Object in a Collection

Creating Collections

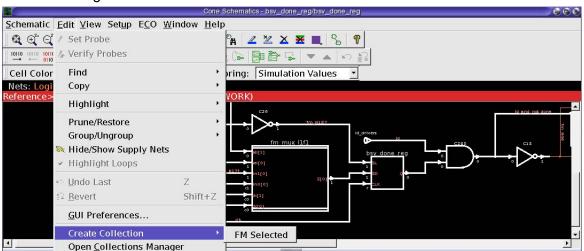
You can create collections by selecting design objects in the GUI or by using the get_* commands.

To create collections, select design objects in the Schematics window that you want to create a collection of and then do either of the following:

- In the Schematics window, choose Edit > Create Collection > FM Selected.
 Or
- 1. In the Formality console, choose Edit > Open Collections Manager.
- 2. Choose Create > FM Selected.

Figure A-1 shows how to create a collection in the Schematics window.

Figure A-1 Creating Collections in the Schematics Window



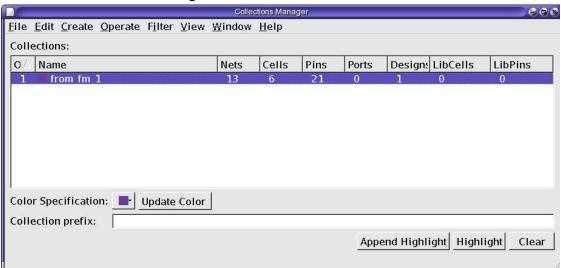
By default, the get_* commands return a collection. The collections are visible in the Collections Manager by setting them to a variable. For example, create and display a collection of design ports by setting a variable to the result of the get ports command:

```
fm_shell> set from_fm_1 [get_ports *]
```

The collection from_fm_1 is displayed in the Collections Manager.

Figure A-2 shows the Collections Manager.

Figure A-2 The Collections Manager



To display the contents of a collection,

- 1. In the Collections Manager, select a collection.
- 2. Choose View > View Collection Contents.

The tool displays the contents of the selected collection in the Global Object Finder window.

You can also right-click on a collection and choose View Contents.

For more information about the Global Object Finder, see "Finding a Design Object in a Collection" on page A-14.

You can specify a prefix to apply to newly generated collection name by entering a prefix in the Collection Prefix box. The name of a new collection is prefixed with the name. The prefix you enter is valid for the current Formality session. However, the name of the collection with the prefix is retained.

You can specify the color of a collection displayed in the schematic. To specify a color,

- 1. In the Collections Manager window, select a collection.
- 2. Choose a color from the color palette.
- 3. Click Update Color.

Figure A-3 shows how to choose a color from the color palette.

Collections Manage Purple <u>File Edit Create Or</u> <u>W</u>indow <u>H</u>elp Blue Collections: Green Name Nets Cells Pins Ports Designs LibCells LibPins Yellow from fm 1 13 21 Orange cells 1 0 6 0 0 ō difference 1 Red Light purple Light blue Light green Light orange Light red Color Specification: Update Color Collection prefix: Append Highlight Highlight

Figure A-3 Updating the Color Representing a Collection

You can also control the highlighting in the schematic by using the Append Highlight, Highlight, and Clear buttons. Appending a highlight adds to what is already highlighted. Highlighting replaces the color of the highlight with the new selection. Clearing removes highlights in the schematic.

Filtering Collections

The Collections Manager displays the number of nets, cells, pins, ports, designs, library cells, and library pins in each collection. Filters create a new collection of the following types of objects, which are a subset of the selected collection:

Choose:	To filter the following type of objects from the selected collection:
Cells > All Cells	All cells
Cells > Primitives >All	Primitive cells
Cells > Primitives > And/Or/Inv/Buf	And/or/inv/buf primitive cells
Cells > Primitives > Xor	XOR primitive cells
Cells > Primitives > DC	Don't-care primitive cells
Cells > Primitives > TRI	Tristate primitive cells
Cells > Primitives > SEQ	SEQ register primitive cells

Choose: To filter the following type of objects from the

selected collection:

Cells > Hier Hierarchical cells

Cells > TechCell Technology cells

Nets Nets

Ports Ports

Pins Pins

Designs Designs

Lib Cells Library cells

Lib Pins Library pins

Name Objects with names that match the specified

string. When you select this filter, the tool displays a dialog box where you can enter the text string. A text string can include glob characters, for

example "OUT[*]" to match "OUT[0]". You can also perform an inverse match to create a collection containing objects that do not match the

specified string.

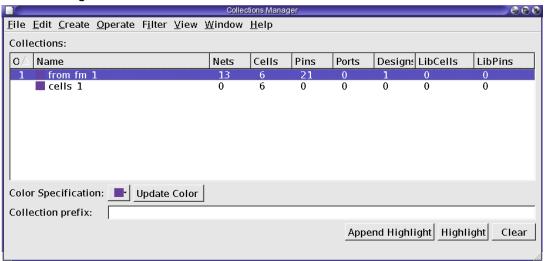
For example, to create a new collection of only the cells in an existing collection,

- 1. In the Collections Manager, select a collection.
- 2. Choose Filter > Cells > type of cell.

The Collections Manager creates a new collection consisting of only the specified type of cells.

Figure A-4 shows how to use the filters.

Figure A-4 Using Filters



Operating on Collections

There are several Operate commands, which allow for actions to be taken directly on one or more collections.

You can perform the following operations on collections:

- Union: Creates a new collection with objects that are in at least one of the selected collections ("OR")
- Intersection: Creates a new collection with objects that are in all the selected collections ("AND")
- Difference: Creates a new collection with objects that are in the first collection but are not in the subsequent collections ("AND NOT")
- Fanin: Creates a new collection with objects that are in the fanin of objects in the selected collections
- Fanout: Creates a new collection with objects that are in the fanout of objects in the selected collections
- Parents: Creates a new collection with objects that are the hierarchical parents of objects in the selected collections
- Instances: Creates a new collection with objects that are instances of the non-instance objects in the selected collections
- Translations: Creates a new collection with objects that are non-instance objects of the instance-based objects in the selected collections

 Net Segments: Creates a new collection with nets that are directly connected, via hierarchical crossings, to nets in the selected collections

For example, to find the difference between two collections, from_fm_1 and cells_1,

1. Select both collections using Shift+Click.

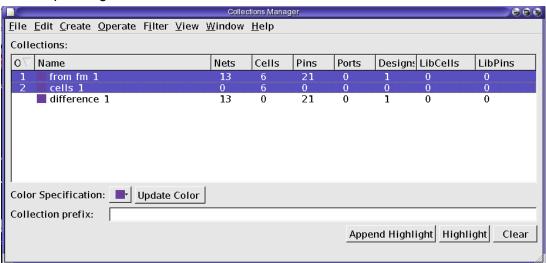
The order of your selection is represented in the first column using numbers.

2. Choose Operate > Differences.

The Collections Manager creates a new collection that consists of the differences between the two selected collections. The second collection you select is removed from the first collection to create a new collection.

Figure A-5 shows how to list the differences between two collections.

Figure A-5 Operating on a Collection



The default name generated for a new collection is based on the action taken to create it.

Finding a Design Object in a Collection

To find an object in the schematic view,

- 1. In the Collections Manager, select a collection.
- 2. Choose View > View Collection Contents. The Global Object Finder dialog box is displayed, which lists the objects in the collection.

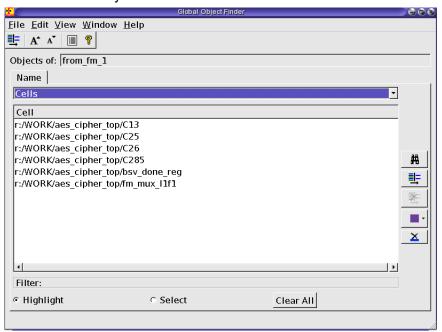
You can also right-click a collection and choose View Contents.

3. In the Name drop-down list, choose Cells, Nets, Pins, or Designs. Objects of the selected type are displayed. The path of the design objects are also displayed.

- 4. Select an object from the list.
 - To choose multiple objects sequentially, press Shift and select multiple objects. To choose multiple objects, press Ctrl and click the object names.
- 5. To choose the color used to highlight the selected objects, choose a color from the color palette.
- 6. Click Highlight to highlight the objects or click Select to select the objects for further operations.

Figure A-6 shows the Global Object Finder displaying a list of design objects in a collection.

Figure A-6 Global Object Finder



B

Tcl Syntax as Applied to Formality Shell Commands

This appendix describes the characteristics of Tcl syntax as applied to Formality shell commands. For instructions about using the Formality shell, see "Invoke the Formality Shell" on page 3-4.

Tcl has a straightforward language syntax. Every Tcl script is a series of commands separated by a new-line character or semicolon. Each command consists of a command name and a series of arguments.

This appendix includes the following sections:

- Using Application Commands
- Quoting Values
- Using Built-In Commands
- Using Procedures
- Using Lists
- Using Other Tcl Utilities
- Using Environment Variables
- Nesting Commands
- Evaluating Expressions

- Using Control Flow Commands
- Creating Procedures

Using Application Commands

Application commands are specific to Formality. You can abbreviate all application command names, but you cannot abbreviate most built-in commands or procedures. Formality commands have the following syntax:

```
command_name -option1 arg1 -option2 arg2 parg1 parg2
command_name
```

Names the application command.

```
-option1 arg1 -option2 arg2
```

Specifies options and their respective arguments.

parg1 parg2

Specifies positional arguments.

Summary of the Command Syntax

Table B-1 summarizes the components of the syntax.

Table B-1 Command Components

Component	Description
Command name	If you enter an ambiguous command, Formality attempts to find the correct command.
	<pre>fm_shell> report_p</pre>
	<pre>Error: ambiguous command "report_p" matched two commands: (report_parameters, report_passing_points) (CMD-006)</pre>
	Formality lists up to three ambiguous commands in its warning.
	To list the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern.
	<pre>fm_shell> help report_p*</pre>

Table B-1 Command Components

Component	Description
Options	Many Formality commands use options. A hyphen (-) precedes an option. Some options require a value argument. For example, in the following command <i>my_lib</i> is a value argument of the <code>-libname</code> option.
	<pre>fm_shell> read_db -libname my_lib</pre>
	Other options, such as <code>-help</code> , are Boolean options without arguments. You can abbreviate an option name to the shortest unambiguous (unique) string. For example, you can abbreviate <code>-libname</code> to <code>-lib</code> .
Positional arguments	Some Formality commands have positional (or unswitched) arguments. For example, in the set_user_match command, the object1 and object2 arguments are positional.
	<pre>fm_shell> set_user_match object1 object2</pre>

Using Special Characters

The characters in Table B-2 have special meaning for Tcl in certain contexts.

Table B-2 Special Characters

Character	Meaning
\$	Dereferences a Tcl variable.
()	Groups expressions.
[]	Denotes a nested command.
\	Indicates escape quoting.
<i>""</i>	Denotes weak quoting. Nested commands and variable substitutions occur.
{}	Denotes rigid quoting. There are no substitutions.
;	Ends a command.
#	Begins a comment.

Using Return Types

Formality commands have a single return type that is a string. Commands return a result. With nested commands, the result can be used as any of the following:

- Conditional statement in a control structure
- Argument to a procedure
- Value to which a variable is set

Here is an example of a return type:

```
if {[verify -nolink]!=1} {
      diagnose
      report_failing_points
      save_session ./failed_run
}
```

Quoting Values

You can surround values in quotation marks in several ways:

- Escape individual special characters by using the backslash character (\) so that the characters are interpreted literally.
- Group a set of words separated by spaces by using double quotation marks (""). This syntax is referred to as weak quoting because variable, command, and backslash substitutions can occur.
- Enclose a set of words that are separated by spaces by using braces ({ }). This
 technique is called rigid quoting. Variable, command, and backslash substitutions do not
 occur within rigid quoting.

The following commands are valid but yield different results. Assuming that variable a is set to 5, Formality yields the following:

```
fm_shell> set s "temp = data[$a]"
    temp = data[5]

fm_shell> set s {temp = data[$a]}
    temp = data[$a]
```

Using Built-In Commands

Most built-in commands are intrinsic to Tcl. Their arguments do not necessarily conform to the Formality argument syntax. For example, many Tcl commands have options that do not begin with a hyphen, yet the commands use a value argument.

Formality adds semantics to certain Tcl built-in commands and imposes restrictions on some elements of the language. Generally, Formality implements all of the Tcl intrinsic commands and is compatible with them.

The Tcl string command has a compare option that is used like this:

```
string compare string1 string2
```

Using Procedures

Formality comes with several procedures that are created through the /usr/synopsys/admin/setup/.synopsys_fm.setup file during installation. You can see what procedures are included with Formality by entering the help command:

The help command returns a list of procedures, built-in commands, and application commands.

Procedures are user-defined commands that work like built-in commands. You can create your own procedures for Formality by following the instructions in "Create Procedures" on page A-12.

Procedures follow the same general syntax as application commands

```
command_name -option1 arg1 -option2 arg2 parg1 parg2
```

For a description of the syntax, see "Using Application Commands" on page B-3.

Using Lists

Lists are an important part of Tcl. Lists represent collections of items and provide a convenient way to distribute the collections. Tcl list elements can consist of strings or other lists.

The Tcl commands you can use with lists are:

- list
- concat
- join
- lappend
- lindex
- linsert
- llength
- lrange
- lreplace
- lsearch
- lsort
- split

While most publications about Tcl contain extensive discussions about lists and the commands that operate on lists, these Tcl commands highlight two important concepts:

- Because command arguments and results are represented as strings, lists are also represented as strings, but with a specific structure.
- · Lists are typically entered by enclosing a string in braces, as follows

```
{a b c d}
```

In this example, however, the string inside the braces is equivalent to the command [list a b c d].

Note:

Do not use commas to separate list items, as you do in Design Compiler.

If you are attempting to perform command or variable substitution, the form with braces does not work. For example, this command sets the variable a to 5.

```
fm_shell> set a 5
5
```

These next two commands yield different results because the command surrounded by braces does not expand the variable, whereas the command surrounded by square brackets (the second command) does.

```
fm_shell> set b {c d $a [list $a z]}
        c d $a [list $a z]

fm_shell> set b [list c d $a [list $a z]]
        c d 5 {5 z}
```

Lists can be nested, as shown in the previous example. You can use the concat command (or other Tcl commands) to concatenate lists.

Using Other Tcl Utilities

Tcl contains several other commands that handle

- Strings and regular expressions (such as format, regexp, regsub, scan, and string)
- File operations (such as file, open, and close)
- Launching system subprocesses (such as exec)

Using Environment Variables

Formality supports any number of user-defined variables. Variables are either scalar or arrays. The syntax of an array reference is

```
array_name (element_name)
```

Table B-3 summarizes several ways for using variables.

Table B-3 Examples of Using Variables

Task	Description
Setting variables	Use the set command to set variables. For compatibility with dc_shell and pt_shell, fm_shell also supports a limited version of the a = b syntax. For example,
	set x 27 or x = 27
	set y \$x or y = \$x
Removing variables	Use the unset command to remove variables.
Referencing variables	Substitute the value of a variable into a command by dereferencing it with the dollar sign (\$), as in echo \$flag. In some cases, however, you must use the name of a value, such as unset flag, instead of the dollar sign.

The following commands show how variables are set and referenced:

```
fm_shell> set search_path ". /usr/synopsys/libraries"
. /usr/synopsys/libraries
fm_shell> adir = "/usr/local/lib"
/usr/local/lib
fm_shell> set my_path "$adir $search_path"
/usr/local/lib . /usr/synopsys/libraries
fm_shell> unset adir
fm_shell> unset my_path
```

Note:

You can also set and unset environment variables in the GUI by entering them into the command bar or selecting File > Environment from the console window.

Nesting Commands

You can nest commands within other commands (also known as command substitution) by enclosing the nested commands within square brackets ([]). Tcl imposes a depth limit of 1,000 for command nesting.

The following examples show different ways of nesting a command.

```
fm_shell> set index [lsearch [set aSort \ [lsort $11]] $aValue]
fm_shell> set title "Gone With The Wind" Gone With The Wind
fm_shell> set lc_title [string tolower $title] gone with the wind
```

Formality makes one exception to the use of command nesting with square brackets so that it can recognize netlist objects with bus references. Formality accepts a string, such as data[63], as a name rather than as the word *data* followed by the result of command *63*. Without this exception, data[63] must either be rigidly quoted with the use of braces, as in {data[63]}, or the square brackets have to be escaped, as in data\[63\].

Evaluating Expressions

Tcl supports expressions. However, the base Tcl language syntax does not support arithmetic operators. Instead, the expr command evaluates expressions.

The following examples show the right and wrong ways to use expressions:

```
set a (12 * $p) ;# Wrong.
set a [expr (12*$p)] ;# Right!
```

The expr command can also evaluate logical and relational expressions.

Using Control Flow Commands

Control flow commands (if, while, for, foreach, break, continue, and switch) determine the order of other commands. You can use fm_shell commands in a control flow command, including other control flow commands.

The following sections briefly describe the use of the control flow commands.

Using the if Command

An if command has a minimum of two arguments:

- An expression to evaluate
- · A script to start conditionally based on the result of the expression

You can extend the if command to contain an unlimited number of elseif clauses and one else clause. An elseif argument to the if command requires two additional arguments: an expression and a script. An else argument requires only a script.

The following example shows the correct way to specify <code>elseif</code> and <code>else</code> clauses:

```
if {$x == 0} {
   echo "Equal"
} elseif {$x > 0} {
   echo "Greater"
} else {
   echo "Less"
}
```

In this example, notice that the <code>else</code> and <code>elseif</code> clauses appear on the same line with the closing brace ()). This syntax is required because a new line indicates a new command. Thus, if the <code>elseif</code> clause is on a separate line, it is treated as a command, although it is not one.

Using while and for Loops

The while and for commands are similar to the same constructs in the C language.

Using while Loops

The while command has two arguments:

- An expression
- A script

The following while command prints squared values from 0 to 10:

```
set p 0
while {$p <= 10} {
    echo "$p squared is: [expr $p * $p]"
    incr p
}</pre>
```

Using for Loops

The for command uses four arguments:

- An initialization script
- A loop-termination expression
- An iterator script
- · An actual working script

The following example shows how the while loop in the previous section is rewritten as a for loop:

```
for {set p 0} {$p <= 10} {incr p} {
     echo "$p squared is: [expr $p * $p]"
}</pre>
```

Iterating Over a List: foreach

The foreach command is similar to the same construct in the C language. This command iterates over the elements in a list. The foreach command has three arguments:

- An iterator variable
- A list
- A script to start (the script references the iterator's variable)

To print an array, enter

```
foreach el [lsort [array names a]] {
  echo "a\($el\) = $a($el)"
}
```

To search in the search path for several files and then report whether or not the files are directories, enter

```
foreach f [which {t1 t2 t3}] {
  echo -n "File $f is "
  if { [file isdirectory $f] == 0 } {
    echo -n "NOT "
  }
  echo "a directory"
}
```

Terminating a Loop: break and continue

The break and continue commands terminate a loop before the termination condition has been reached. The break command causes the innermost loop to terminate. The continue command causes the current iteration of the innermost loop to terminate.

Using the switch Command

The switch command is equivalent to an if tree that compares a variable with a number of values. One of a number of scripts is run, based on the value of the variable:

```
switch $x {
a {incr t1}
b {incr t2}
c {incr t3}
}
```

The switch command has other forms and several complex options. For more examples of the switch command, consult your Tcl documentation.

Creating Procedures

A powerful Formality function is the ability to write reusable Tcl procedures. With this function, you can extend the command language. You can write new commands that can use an unlimited number of arguments. The arguments can contain the default values, and you can also use a varying number of arguments.

For example, the following procedure prints the contents of an array:

```
proc array_print {arr} {
     upvar $arr a
     foreach el [lsort [array names a]] {
        echo "$arr\($el) = $a($el)"
     }
}
```

Procedures can use any of the commands supported by Formality or other procedures you have defined. Procedures can even be recursive. Procedures can contain local variables and reference variables outside of their scope. Arguments to procedures can be passed by value or by reference.

The following sections provide some examples of procedures. Books on the Tcl language offer additional information about writing procedures.

Setting Defaults for Arguments

To set up a default for an argument, you must locate the argument in a sublist that contains two elements: the name of the argument and the default.

For example, the following procedure reads a favorite library by default, but reads a specific library if given:

```
proc read_lib { {lname favorite.db} } {
     read_db $lname
}
```

Specifying a Varying Number of Arguments

You can specify a varying number of arguments by using the args argument. You can enforce that at least some arguments are passed into a procedure, and then handle the remaining arguments as you see fit.

For example, to report the square of at least one number, use the following procedure:

```
proc squares {num args} {
  set nlist $num
  append nlist " "
  append nlist $args
  foreach n $nlist {
    echo "Square of $n is [expr $n*$n]"
  }
}
```

Index

Symbols > operator 3-15 >> operator 3-15	automated setup file reading 5-6 reading multiple 5-13 automatic creating compare points 1-7 creating containers 6-6
aborted compare points defined 9-6 during verification 9-7 alias command 3-14 aliases 3-14 analyze_points command 10-7, 10-45 analyzing differences 11-5 AND resolution function 7-43, 7-44 application commands B-3 arguments designID 6-8 positional B-3 programming default values B-13 varying the number of B-13 arithmetic generator creating multiplier architectures 7-47 ASIC verification flow diagram 1-3, 1-4 asynchronous bypass logic 7-35 asynchronous state-holding loops 7-37 attributes, black box 7-7 automated 5-5	batch mode controlling verification 9-21 overview 9-20 preparing for 9-20 scripts 9-20 black box attributes 7-7 controlling 7-4 debugging 10-11 identity check 3-21, 7-4, 7-7 loading design interfaces 7-5 locating 10-8 marking designs 7-6 overview 7-3 redefining 7-8 reporting 7-6 resolution function 7-43, 7-44 setting pins and port directions 7-8 unresolved 7-8 verifying 7-3, 10-8

block abstraction	flow control, Tcl B-10
reading 6-11	fm shell 3-4
boundary scan 7-15	getting syntax information 3-8, 3-9
built-in commands B-5	help 3-8, B-6
bus holders 7-4	-help option 3-8, 3-9
bus fioliders 7-4	history 3-12
	interrupting 3-7, 3-19
C	library_verification 12-3
cell	line breaks 3-10
defined 6-5	man 3-8, 3-9
library verification 12-1	match 8-4
change_link 11-8	multiline shell commands 3-10
change_names command 10-19	nesting B-4
clock tree buffering 7-16	positional arguments B-3
collapse_all_cg_cells value 7-20	procedures B-6
color-coding	puts 3-16
error candidates 10-24	read_db 6-7, 12-4, 12-5
schematic area 10-24	read_ddc 6-10
combinational design changes 7-14	read_fsm_states 7-39
boundary scan 7-15	read_milkyway 6-8, 6-9, 6-10 read_verilog 6-13, 12-4
clock tree buffering 7-16	read_vhdl 6-13, 6-14
command alias 3-14	recalling 3-13
command log file 3-16	redirect 3-15
command names, syntax B-3	remove_compare_rules 8-4, 10-17
command results, returning B-4	remove constraint 7-13
command shortcuts 3-13	remove_constraint_type 7-13
commands	remove_dont_verify_points 9-22
alias 3-14	remove_user_match 8-4, 10-13
application, using B-3	rename_object 10-18, 10-19
built-in B-5	report_aborted_points 9-5
case-sensitivity 3-10	report_architecture 7-50
change_names 10-19	report_black_boxes 7-6
commenting B-4	report_cell_list 12-5
cputime 3-9	report_compare_rules 10-17
create_constraint_type 7-12	report_constants 7-10
create_container 3-21, 6-17, 6-18, 7-7, 7-8,	report_constraint 7-14
7-43	report_constraint_type 7-14
debug_library_cell 12-10	report_dont_verify_points 9-22
echo 3-16	report_failing_points 9-5
elaborate_library_cells 12-7	report_loops 7.38
entering commands 3-10	report_loops 7-38

report_status 12-9 report_truth_table 12-10 report_unmatched_inputs 10-8 report_unmatched_points 8-5 report_user_matches 10-13 returning results 3-10 select_cell_list 12-6 set_architecture 7-48 set_black_box 7-6 set_compare_rule 10-14 set_constant 7-9 set_constraint 7-12 set_dont_verify_points 9-22 set_fsm_encoding 7-40 set_fsm_state_vector 7-40 set_inv_push 7-24 set_parameters 7-27, 7-28 set_svf 5-4 set_user_match 7-17, 10-12 setup 8-4 source 3-18 special characters B-4 Tcl syntax B-1 test_compare_rule 10-16 unalias 3-15 undo_match 8-8 verify 9-3, 9-5, 9-10, 12-7	flow 8-3 techniques 1-8, 8-8 name filtering 8-10 name-based matching 8-8 net-name matching 8-12 non-name-based matching 8-8 not verified 9-6 objects used to create 1-8 overview 1-8 passing 9-6 removing 10-13 signature analysis 8-11 status messages 9-6 topological equivalence 8-11 undoing match command 8-8 unmatched, reporting 8-5 unverified 9-6 verifying single 9-8, 9-10 compare rules 10-14 defining 10-17 removing 10-17 testing 10-16 complete verification 1-7 concepts
verify 9-3, 9-5, 9-10, 12-7 write_hierarchical_verification_script 9-13	black boxes 7-3 compare points 1-8
write_library_debug_scripts 12-11	constants 7-8
commenting commands B-4	containers 6-5
compare points aborted 9-6, 9-7	current container 6-19 design equivalence 1-10
automatic creation of 1-7	design objects 6-5
debugging 10-11	external constraints 7-11
debugging unmatched 8-6	FSMs 7-38
defining your own 1-8 exact-name matching 8-9	implementation design 1-5 logic cones 1-6
example 1-8	LSSD cell 7-46
failing 9-6	reference design 1-5
listing user-matched points 10-13	resolution functions 7-43
mapping names between 1-8	signature analysis 8-11
matched state 8-3	verification 9-8

connect_net 11-8, 11-12	creating
connectivity errors	constraint types 7-12
reporting 11-18	containers 6-6, 6-18
consensus, resolution function 7-43	multiplier architectures 7-47
consistency, defined 1-10	procedures B-12
console window	tutorial directories 4-2
toolbar 10-21	.cshrc 4-2
constants	current
defining 7-8	container 6-6, 6-18
listing 7-10	design 6-18
overview 7-8	current_design 11-9
propagating 7-8, 7-42	current_instance 11-9
removing 7-9	customer support iii-xxi
types 7-8	
user-defined 7-8	D
constraint module 7-12	D
containers	data
automatic creation 6-6	containers 6-5
contents 6-5	output file types 3-21
creating 6-6, 6-18	.ddc databases
current 6-6, 6-18	designs from 6-10
listing 6-18	reading designs 6-10
naming 6-18	.ddc files 6-10
overview 6-5	debug_library_cell command 12-10
reading data into 6-6	debugging
control flow commands, Tcl B-10	black boxes 10-11
control statements 9-21	compare points 10-11
Control-c interrupt 3-7, 3-19, 9-7	determining failure causes 10-7
controlling	eliminating setup possibilities 10-11
black boxes 7-4	gathering information 10-5
conventions for documentation iii-xx	library cells 12-10
coverage percentage 10-35	matching with compare rules 10-14
cputime command 3-9	renaming objects 10-19
create_cell 11-8	setting compare points to match 10-12
create_constraint_type command 7-12	strategies 10-5
create_container command 3-21, 6-17, 6-18,	subset matching 10-17
7-7, 7-8, 7-43	unmatched compare points 8-6
create_net 11-8	using diagnosis 10-8 using logic cones 10-9
create_port 11-8	working with subcones 10-36
create_primitive 11-8	defining
ordate_primitive ri-o	demining

compare points 1-8	E
constants 7-8	echo command 3-16
FSM states 7-38	ECO modifications
dereferencing variables B-4	exporting 11-22
design equivalence	verifying 11-20
overview 1-10	edit files
design objects	creating 11-16
finding 10-25	loading 11-16
generating lists 10-26	using 11-16
overview 6-5	edits
renaming 10-18	committing 11-17
unmatched 1-8	reporting 11-17
used in compare point creation 1-8	undoing 11-16
designID, argument 6-8	elaborate_library_cells command 12-7
designs	enable_multiplier_generation variable 7-48
constants 7-8	equivalent nets
current 6-18	finding 11-6
designID 6-8	error candidates
flattened 7-42	color-coding 10-24
hierarchy separator style 7-42	coverage percentage 10-35
implementation 1-5	error messages 3-19, 3-20
linking (with set_top) 6-5	-
locating problem areas 10-5	escape quoting B-4
marking as black boxes 7-6	exact-name compare point matching 8-9
propagating constants 7-8	examples
reference 1-5	bus holder 7-4
retiming 7-25	compare point, creation 1-8
retiming using Design Compiler 7-26	logic cone 1-6
retiming with other tools 7-27	multiply-driven nets 7-45
setting up 7-1	resolution function 7-45
diagnosis	schematic view window 10-21
debugging with 10-8	expressions
interrupting 3-7, 3-19	evaluation, Tcl B-9
directory	grouping B-4
creating for tutorial 4-2	external constraints
disconnect_net 11-8, 11-12	creating constraint types 7-12
don't care information	overview 7-11
verification modes 1-10	removing 7-13
	removing constraint type 7-13
	reporting 7-14
	reporting constraint types 7-14

setting 7-12	starting 3-4
types 7-11	fm_shell command 3-4
user-defined 7-12	-f option 9-21
	syntax B-3
_	within GUI 3-17
F	for loops, Tcl flow control B-11
failing compare points, defined 9-6	foreach command, Tcl flow control B-11
failing patterns	Formality
applying in the logic cone view window 10-31	log files 10-5
coverage percentage 10-35	Formality Ultra 11-1
failing points	flow 11-3
generating 11-5	formality.log file 10-5
files	FSM re-encoding 7-39
batch script 9-20	1 OW To chooding 7 Oo
command log 3-16	
.ddc 6-10	G
formality.log 10-5	golden design 1-5
output 3-21	grouping expressions B-4
reading automated setup 5-6	grouping words, Tcl commands B-5
session log 3-16 state files for FSMs 7-39	GUI
	current container 6-19
find_equivalent_nets 11-6	highlighting toolbar 10-39
find_svf_operation command 5-12	logic cone view window 10-31, 10-32
finding	overview 3-16
design objects 10-25	Probe Points tab 10-38
equivalent nets 11-6 lists of design objects 10-26	schematic view window 10-20
unmatched black boxes 10-8	starting 3-6
finite state machines (FSMs)	guidance summary 5-10, 10-44
defining states individually 7-39, 7-40	
listing state encoding information 7-41	П
overview 7-38	Н
preparing for verification 7-38	hdlin_infer_multibit variable 7-50
state files 7-39	hdlin_interface_only variables 3-4, 3-5, 3-8,
flattening designs	3-12, 3-13, 3-14, 3-15, 3-18, 3-19, 3-20
constant propagation 7-42	7-5
during verification 7-42	hdlin_library_directory variable 6-13
separator style 7-42	hdlin_library_file variable 6-13
fm_shell	hdlin_multiplier_architecture variable 7-48
getting help 3-8	help
listing commands 3-8	command 3-8, B-6

fm_shell commands 3-8	GUI 3-6
-help, options 3-8	isolating
hierarchical designs 7-42	subcones 10-37
GUI representation 6-8	
hierarchy separator style 7-42	1
propagating constants 7-42	L
storing 6-6	libraries
traversing 10-21, 10-24	Verilog simulation 6-14
verifying 7-32	library verification 12-1
hierarchical separator character, defining 7-42	debugging process 12-7, 12-10
hierarchical verification 9-12	example Tcl script 12-7
high-level editing commands	implementation library 12-5
hierarchical designs 11-13	initializing 12-3
using 11-11	process flow 12-2
history command 3-12	reference library 12-4
	reporting library cells 12-5 reporting status 12-9
I	specifying the cells to verify 12-6
1	supported formats 6-12, 6-19, 6-20, 12-3
identity check, black boxes 3-21, 7-4, 7-7	truth tables 12-10
if command, Tcl flow control B-10	verifying the libraries 12-7
implementation design	versus design verification 12-3, 12-4, 12-7
establishing 1-5	library_verification command 12-3
modifying 11-8	limiting
overview 1-5	messages 3-21
implementation libraries 12-5	linking designs 6-5
install directory 4-2	listing
installation 2-3, 4-2	constants 7-10
interfaces	design objects 10-26
GUI 3-16	fm_shell commands 3-8
interpreting verification results 9-21	previously entered commands 3-12
interrupting	lists in Tcl B-6
diagnosis 3-7, 3-19	loading design interfaces, black boxes 7-5
fm_shell commands 3-7, 3-19 verification 9-7	locating
	design objects 10-25
introduction to Formality 1-1	design problems 10-5
inversion push 7-23	lists of design objects 10-26
environmental 7-25	problems 7-8
instance-based 7-24	unmatched black boxes 10-8
invoking fm_shell 3-4	log file 3-16
1111_311611 3 -4	log files

formality.log 10-5	implementation design 11-8
Logic Cone	multibit support 7-50
grouping hierarchy	mw_logic0_net variable 6-11
logic cone 10-37	mw_logic1_net variable 6-11
logic cone view window	_ 🗸 _
applying failed patterns 10-31	N.I.
overview 10-31	N
subcones 10-36	name filtering compare point matching 8-10
logic cone, diagnose 10-9	name_match variable 8-9
logic cones originating point 1-6	name_match_allow_subset_match variable 8-9, 10-17
overview 1-6 termination points 1-6	name_match_based_on_nets variable 8-9, 8-13
viewing 10-31 loops, Tcl B-10	name_match_filter_chars variable 8-9, 8-10, 10-17
LSSD cell, defined 7-46	name_match_flattened_hierarchy_separator_ style variable 8-9
M	name_match_multibit_register_reverse_order variable 8-9, 8-11
man command 3-8, 3-9	name_match_use_filter variable 8-10, 10-18
man page overview 3-8	name_matched_flattened_hierarchy_separato
managing, black boxes 7-4	r_style variables 7-42
mapping names	nesting commands B-4, B-9
compare points 1-8	net-name compare point matching 8-12
design objects 10-18	nets
marking designs as black boxes 7-6	constant value 7-8
match command 8-4	setting to a constant 7-9
matched verification status 8-3	with multiple drivers 7-43
matching compare points 8-4	not verified compare points 9-6
message thresholds, setting 3-20	
messages	0
error 3-19	entions syntax D 4
limiting 3-21	options, syntax B-4
setting threshold 3-21	OR resolution function 7-43, 7-44
syntax 3-19	output file types 3-21
types 3-21	redirecting 3-15
Milkyway databases reading 6-10	overview
modifications	black boxes 7-3
displaying 11-17	compare points 1-8
modifying	constants 7-8
··· , ···- g	

containers 6-5 design equivalence 1-10 design objects 6-5 external constraints 7-11 finite state machines (FSMs) 7-38 implementation design 1-5 library verification 12-2 logic cones 1-6 man pages 3-8 reference design 1-5 resolution functions 7-43 verification 9-8	problem areas, see troubleshooting procedures creating B-12 default B-6 propagating constants 7-8, 7-42 puts built-in command 3-16 Q quick-start tutorial 2-1, 4-1, 4-7 quotation marks, using B-5
P	R
parameters automatically flattening designs 7-42 constant propagation 7-42 design 7-1 hierarchical separator style 7-42 identity check, black boxes 3-21, 7-7 message threshold 3-21 multiply-driven net resolution 7-44	read_db command 6-7, 12-4, 12-5 read_ddc command 6-10 read_fsm_states command 7-39 read_milkyway command 6-8, 6-9, 6-10 read_verilog command 6-13, 12-4 read_vhdl command 6-13, 6-14 reading block abstraction 6-11
passing compare points 9-6 path, setting 4-2 pins	.ddc databases 6-10 FSM states 7-39 multiple automated setup files 5-13
defining direction 7-8 port punching 11-14	record_edits 11-23 redefining
ports constant value 7-8 defining direction 7-8 direction, black boxes 7-4 setting to a constant 7-9	black boxes 7-8 redirect command 3-15 redirecting output 3-15 reference design
positional arguments with, commands B-3 power domain, cell coloring 10-39	establishing 1-5 overview 1-5
power models verifying hierarchical designs 7-32	reference libraries 12-4 regression testing 1-5
previous session, sourcing 3-16 printing	remove_cell 11-8 remove_compare_rules command 8-4, 10-17
schematics 10-27 Probe points 10-38	remove_constant command -type option 7-9

remove_constraint command 7-13	library cells 12-5
remove_constraint_type command 7-13	library verification results 12-9
remove_dont_verify_points command 9-22	truth tables 12-10
remove_net 11-8	resolution functions
remove_port 11-8	multiply-driven nets 7-43
remove_user_match command 8-4, 10-13	overview 7-43
remove_verify_points 11-22	resolving
removing	multiply-driven nets 7-43 nets with multiple drivers 7-43
subcones 10-37	results 9-7
rename_object command 10-18, 10-19	
renaming design objects 10-18	retimed designs, working with 7-25 retiming designs
report_aborted_points command 9-5	using Design Compiler 7-26
report_architecture command 7-50	using other tools 7-27
report_black_boxes command 7-6	verifying with Formality 7-27
report_cell_list command 12-5	returning
report_compare_rules command 10-17	subcones 10-37
report_constants command 7-10	returning shell command results B-4
report_constraint command 7-14	rigid quoting B-4
report_constraint_type command 7-14	
report_dont_verify_points command 9-22	S
report_electrical_checks 11-18	
report_failing_points command 9-5	schematic view window
report_fsm command 7-41	example 10-21 zooming 10-27
report_guidance command 5-12	schematics
report_loops command 7-38	printing 10-27
report_status command 12-9	viewing logic cones 10-31
report_svf_operation command 10-46	script file
report_truth_table command 12-10	batch jobs 9-20
report_unmatched_points command 8-5	sourcing 3-18, 9-21
report_user_matches command 10-13	tasks 3-18
report_verify_points 11-22	select_cell_list command 12-6
reporting	separating list items, Tcl commands B-7
black boxes 10-8	sequential design changes 7-18
reporting black boxes 7-6	asynchronous bypass logic 7-35
reports 10-5	inversion push 7-23
compare rules 10-17	set_architecture command 7-48
constants, user-defined 7-10	set_black_box command 7-6
containers 6-18	set_compare_rule command 10-14
finite state machine (FSMs) information 7-41	

set_constant command 7-9	source command
-type option 7-9	batch jobs 9-21
set_constraint command 7-12	syntax 3-18
set_dont_verify_points 11-21	sourcing
set_dont_verify_points command 9-22	previous sessions 3-16
set_fsm_encoding command 7-40	script files 3-18, 9-21
set_fsm_state_vector command 7-40	special characters
set_inv_push command 7-24	Tcl B-4
set_parameters command 7-27, 7-28	starting fm_shell 3-4
set_probe_points command 10-38	GUI 3-6
set_svf command 5-4	state files for FSMs 7-39
set_top command	subcones 10-36, 10-37
conditions 6-12	succeeded verification 9-7
set_user_match command 7-17, 10-12 setting	svf_ignore_unqualified_fsm_information variable 7-39
implementation design 1-5	switch command, Tcl flow control B-12
message thresholds 3-21	syntax
pins and port directions 7-8	fm_shell commands B-3
reference design 1-5	procedures B-6
setup 5-5	
automated 5-5, 5-6 reading 5-6	Т
setup command 8-4	Tcl
setup mode	arguments, varying the number of B-13
automated 5-5	break command B-12
reading 5-6	commands that support lists B-6
setup, automated 5-5	continue command B-12
severity rating for messages 3-21	control flow commands B-10
shell interface, starting 3-4	expression evaluation B-9
signature analysis 8-11	for loops B-11
signature_analysis_match_primary_input variable 8-9	foreach command B-11 grouping words B-5
signature_analysis_match_primary_output variable 8-9, 8-12	lists B-6 nesting commands B-9
signature_analysis_matching variable 8-9, 8-12	overview B-1 quoting values B-5
single-state holding elements 7-46	separating list items 3-11, B-7
SolvNet	special characters B-4 switch command B-12
accessing iii-xxi	user-defined variables B-8
documentation iii-xviii	user-ueimen vanables D-0

while command B-10 TECH_WORK library 6-7 technology libraries default name 6-7 shared 6-18, 6-19 unshared 6-18 verifying 12-1 terminating loops, Tcl B-12 test_compare_rule command 10-16 thresholds, message level 3-21 toolbar, console window 10-21	unalias command 3-15 undo_edits 11-9 undo_match command 8-8 unmatched compare points 8-5 unverified compare points, defined 9-6 UPF IEEE 1801 (UPF) 7-27 Low Power Designs 7-27 user-defined compare points 1-8
top-level designs 6-5	constants 7-8
topological equivalence 8-11 transcript window 10-5	removing 7-9
traversing hierarchical designs 10-21, 10-24	reporting 7-10 variables, Tcl B-8
troubleshooting asynchronous state-holding loops 7-37 black boxes 10-8, 10-11 determining failure cause 10-5, 10-7 eliminating setup possibilities 10-11 extraneous bus drivers 7-4 gathering information 10-5 incomplete verification 10-5 locating problems 7-8 logic cones, viewing 1-6 matching with compare rules 10-14 problem areas, locating 10-5 renaming objects 10-19 setting compare points to match 10-12 subset matching 10-17 unmatched compare points 8-6 using diagnosis 10-8 using logic cones 10-9 working with subcones 10-36 truth table 12-11 tutorial 2-1 tutorial directories creating 4-2	V values, quoting B-5 variables dereferencing B-4 enable_multiplier_generation 7-48 hdlin_infer_multibit 7-50 hdlin_interface_only 3-4, 3-5, 3-8, 3-12,
	name_match_use_filter 8-10, 10-18 name_match_use_filter variable 8-9

name_matched_flattened_hierarchy_separa tor_style 7-42 signature_analysis_match_primary_input 8-9 signature_analysis_match_primary_output 8-9, 8-12 signature_analysis_matching 8-9, 8-12 svf_ignore_unqualified_fsm_information 7-39 verification_asynch_bypass 7-37 verification_blackbox_match_mode 8-9 verification_clock_gate_hold_mode 7-20, 7-22 verification_constant_prop_mode 7-43 verification_incremental_mode 9-4 verification_inversion_push 7-25 verification_timeout_limit 9-11 /erification batch mode 9-20 boundary scan 7-15 cell libraries 12-1 clock tree buffering 7-16 compare point matching 8-8 complete 1-7 consistency 1-10 constant propagation 7-8, 7-42 controlling 9-21 CPU time 1-5 design equality 1-10 establishing environment 3-1, 5-1, 7-1 failed 10-7 finite state machines 7-38 flattened designs 7-42 hierarchical 9-12 hierarchical designs 7-42 inserting cutpoints 7-38 interrupting 9-7 LSSD cells 7-46 overview 1-10, 9-8	results 9-21 sequential design changes 7-18 setting external constraints 7-11 single compare point 9-8, 9-10 starting 9-3 status messages 9-6, 9-7 succeeded status 9-7 technology libraries 12-1 transformed designs 7-14 troubleshooting 10-5 using diagnosis 10-8 using logic cones 10-9 viewing results 9-6 verification_asynch_bypass variable 7-37 verification_blackbox_match_mode variable 8-9 verification_clock_gate_hold_mode variable 7-20, 7-22 verification_incremental_mode variable 9-4 verification_inversion_push variable 7-25 verification_timeout_limit variable 9-11 verification_verify_directly_undriven_output variable 5-5 verify command 9-3, 9-5, 9-10, 12-7 verify points removing 11-22 reporting 11-22 verifying black box behavior 7-3 black boxes 7-3 designs 9-3 marking design as black box 7-6 single compare point 9-10 Verilog simulation, library files 6-14 viewing logic cones 10-31
performing 8-1, 9-1 problem areas, locating 10-5 reporting progress 9-21	W work quoting R 4
1 31 -3	weak quoting B-4

wildcard characters 3-8
write_hierarchical_verification_script
command 9-13
write_library_debug_scripts command 12-11

Z zooming in and out 10-27