

Formality[®]

User Guide

Version Z-2007.06, June 2007

Comments?

Send comments on the documentation by going to <http://solvnetsynopsys.com>, then clicking "Enter a Call to the Support Center."

SYNOPSYS[®]

Copyright Notice and Proprietary Information

Copyright © 2007 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____.”

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.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules,

Hierarchical Optimization Technology, HSIM^{plus}, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Contents

What's New in This Release	xviii
About This User Guide	xviii
Customer Support	xxi
1. Introduction to Formality	
What Is Formality?	1-2
How Does Formality Fit Into My Design Methodology?	1-4
What Designs Can I Verify?	1-8
Design Requirements	1-8
Design Types	1-8
Verification of Two RTL Designs	1-9
Verification of an RTL Design and a Gate-Level Design	1-9
Verification of Two Gate-Level Designs	1-10
What Pieces Make Up Formality?	1-11
General Process Flow	1-13
Input and Output File Types	1-15

Input	1-15
Libraries	1-19
Output	1-20
Controlling File Names Generated by Formality	1-23
Synopsys Setup File	1-24
Concepts	1-24
Compare Points	1-25
Compare Rules	1-28
Containers	1-29
Design Equivalence	1-31
Logic Cones	1-33
Reference Design and Implementation Design	1-34
Solvers	1-35
 2. Quick Start With Formality	
Before You Start	2-2
Creating Tutorial Directories	2-2
Tutorial Directory Contents	2-3
Invoking the Formality Shell and GUI	2-3
Graphical User Interface	2-4
Verifying fifo.vg Against fifo.v	2-6
Loading the Automated Setup File.	2-6
Specifying the Reference Design.	2-7
Specifying the Implementation Design.	2-9
Setting Up the Design	2-11

Compare Point Matching	2-11
Verifying the Designs	2-12
Debugging	2-13
Verifying fifo_with_scan.v Against fifo_mod.vg	2-18
Verifying fifo_jtag.v Against fifo_with_scan.v	2-23
Debugging Using Diagnosis	2-26
For More Information	2-29
 3. Starting Formality	
Invoking Formality	3-3
Starting the Shell Interface	3-3
Invoking the Formality GUI.	3-5
Formality Shell Environment	3-6
Entering Commands	3-7
Supplying Lists of Arguments	3-8
Editing From the Command Line	3-9
Listing Previously Entered Commands	3-10
Recalling Commands.	3-12
Redirecting Output.	3-13
Using Command Aliases	3-14
Using the alias Command	3-15
Using the unalias Command	3-15
Listing Design Objects	3-16
Using the Formality GUI Environment.	3-17
Managing Formality Windows	3-17

Using the Formality Prompt	3-18
Saving the Transcript	3-18
Copying Text From the Transcript Area	3-19
Copying Text to the Formality Prompt	3-19
General Formality Usage Options	3-20
Getting Help	3-20
Interrupting Formality	3-22
Controlling Messages	3-23
Setting Message Thresholds	3-25
Working With Script Files	3-26
Using the Command Log File	3-27
Controlling the File Search Path	3-28
Tcl Source Command	3-28
Examining the File Search Path	3-29
 4. Setting Basic Elements for Design Verification	
Reading in Libraries and Designs	4-3
Setup-free Flow	4-5
Technology Libraries	4-7
Designs	4-8
Changing Bus Naming and Dimension Separator Styles	4-9
Supporting DesignWare Components	4-11
Setting Variables for VHDL and Verilog Directives	4-12
Top-Level Design	4-14
Loading the Reference Design	4-15
Reading Technology Libraries	4-15

Synopsys (.db) Format	4-15
SystemVerilog, Verilog, and VHDL RTL Format	4-16
Verilog Simulation Data	4-18
Reading Design Libraries	4-20
Reading Milkyway and .ddc Databases	4-22
Milkyway Databases	4-23
.ddc Databases	4-24
Setting the Top-Level Design	4-24
Loading the Implementation Design	4-28
Setting Up and Managing Containers	4-29
 5. Preparing the Design for Verification	
Using Don't Care Cells	5-3
Setting Up Designs	5-4
Supporting Multibit Library Cells	5-5
Resolving Nets With Multiple Drivers	5-6
Eliminating Asynchronous State-Holding Loops	5-10
Working With Cutpoints	5-12
Creating a Cutpoint	5-13
Reporting Information About Cutpoints	5-14
Removing a Cutpoint	5-14
Working With Black Boxes	5-15
Loading Design Interfaces	5-17
Marking a Design as a Black Box for Verification	5-18
Reporting Black Boxes	5-19
Performing Identity Checks	5-20

Setting Pin and Port Directions for Unresolved Black Boxes	5-21
Working With Constants	5-22
Defining Constants	5-23
Removing User-Defined Constants	5-24
Listing User-Defined Constants	5-24
Working With Equivalences	5-25
Defining an Equivalence	5-26
Removing User-Defined Equivalences	5-26
Listing User-Defined Equivalences	5-27
Working With External Constraints	5-30
Defining an External Constraint	5-32
Creating a Constraint Type	5-33
Removing an External Constraint	5-34
Removing a Constraint Type	5-34
Reporting Constraint Information	5-35
Reporting Information About Constraint Types	5-35
Working With Hierarchical Designs	5-35
Setting the Flattened Hierarchy Separator Character	5-36
Propagating Constants	5-37
Working With Combinational Design Changes	5-38
Disabling Scan Logic	5-38
Disabling Boundary Scan in Your Designs	5-39
Managing Clock Tree Buffering	5-41
Working With Sequential Design Changes	5-42
Managing Asynchronous Bypass Logic	5-43
Setting Clock Gating	5-45
Enabling an Inversion Push	5-50
Instance-Based Inversion Push	5-51

Environmental Inversion Push	5-52
Working With Reencoded Finite State Machines.	5-53
Using the Automated Setup File for FSM Reencoding.	5-54
Reading a User-Supplied FSM State File	5-55
Defining FSM States Individually	5-56
Multiple Reencoded FSMs in a Single Module.	5-56
Listing State Encoding Information.	5-57
Working With FSMs Reencoded in Design Compiler.	5-58
Working With Retimed Designs	5-59
Retiming Using Design Compiler	5-59
Retiming Using Other Tools	5-62
Working With Single-State Holding Elements	5-62
Working With System Functions for Low-Power Design	5-63
Working with Retention Registers without RTL System Functions for Low Power Design	5-66
Working With Multiplier Architectures	5-67
Setting the Multiplier Architecture.	5-68
Reporting Your Multiplier Architecture	5-71
Working With Automated Setup Files	5-72
Creating an Automated Setup File	5-72
Reading an Automated Setup File Into Formality.	5-73
Reading In Multiple Automated Setup Files	5-74
Automated Setup File Messaging	5-75
Automated Setup File Diagnostic Messages	5-81
Automated Setup File Conversion to Text	5-81
Removing Information	5-82
Removing Designs.	5-82
Removing Design Libraries	5-83

Removing Technology Libraries	5-83
Removing Containers	5-84
Converting Objects to Black Boxes	5-85
Saving Information.	5-85
Saving Containers	5-86
Saving the Entire Formality Session	5-86
Restoring Information	5-88
Restoring Containers	5-88
Restoring a Session.	5-89
 6. Compare Point Matching and Verification	
Matching Compare Points	6-3
Performing Compare Point Matching	6-5
Reporting Unmatched Points	6-6
Undoing Matched Points	6-7
Debugging Unmatched Points	6-8
How Formality Matches Compare Points	6-10
Exact-Name Matching	6-12
Reversing the Bit Order in Multibit Registers	6-13
Name Filtering	6-14
Topological Equivalence	6-15
Signature Analysis	6-16
Compare Point Matching Based on Net Names	6-18
Performing Verification.	6-19
Verifying a Design	6-19
Verifying a Single Compare Point	6-21

Removing Compare Points From the Verification Set	6-22
Controlling Verification Runtimes	6-24
Distributing Verification Processes	6-25
Setting Up the Distributed Environment	6-25
Verifying Your Environment	6-27
Interrupting Verification	6-29
Performing Hierarchical Verification	6-29
Using Batch Jobs	6-31
Starting Verification	6-32
Controlling Verification	6-33
Interrupting Verification	6-33
Verification Progress Reporting	6-33
Reporting and Interpreting Results	6-34
 7. Debugging Failed Design Verifications	
Debugging Process Flow	7-3
Gathering Information	7-4
Handling Designs When Verification Is Incomplete	7-4
Determining Failure Causes	7-7
Debugging by Using Diagnosis	7-9
Debugging by Using Logic Cones	7-11
Eliminating Setup Possibilities	7-14
Black Boxes	7-14
Unmatched Points	7-15
Matching With User-Supplied Names	7-15

Matching With Compare Rules	7-19
Matching With Name Subset	7-24
Renaming User-Supplied Names or Mapping File	7-26
Design Transformations	7-28
Working With Schematics	7-29
Viewing Schematics	7-29
Traversing Design Hierarchy	7-33
Finding a Particular Object	7-34
Generating a List of Objects	7-35
Zooming In and Out of a View	7-36
Viewing RTL Source Code	7-37
Working With Logic Cones	7-38
Pruning Logic	7-43
Working With Failing Patterns	7-44
Saving Failing Patterns	7-47
Running Previously Saved Failing Patterns	7-48
 8. Technology Library Verification	
Overview	8-3
Initializing Library Verification	8-4
Loading the Reference Library	8-5
Loading the Implementation Library	8-6
Listing the Cells	8-7
Specifying a Customized Cell List	8-8

Elaborating Library Cells	8-9
Performing Library Verification	8-9
Reporting and Interpreting Verification Results.	8-12
Debugging Failed Library Cells	8-14
Appendix A. Appendix A - Tcl Syntax as Applied to Formality Shell Commands	
Using Application Commands	A-3
Summary of the Command Syntax	A-4
Using Special Characters	A-5
Using Return Types	A-5
Quoting Values	A-6
Using Built-In Commands	A-6
Using Procedures	A-7
Using Lists.	A-8
Using Other Tcl Utilities.	A-10
Using Environment Variables	A-10
Nesting Commands.	A-11
Evaluating Expressions	A-12
Using Control Flow Commands.	A-12
Using the if Command	A-13
Using while and for Loops	A-14
Using while Loops	A-14
Using for Loops	A-14

Iterating Over a List: foreach	A-15
Terminating a Loop: break and continue	A-15
Using the switch Command	A-16
Creating Procedures	A-16
Programming Default Values for Arguments	A-17
Specifying a Varying Number of Arguments	A-17

Appendix B. Appendix B - Formality Library Support

Overview	B-2
Supported Library Formats	B-3
Synopsys Synthesis Libraries	B-3
Verilog Simulation Libraries	B-3
Synthesizable RTL	B-4
Gate-Level Netlists	B-4
Updating Synthesis Libraries	B-4
Library Enhancement and Generation Process	B-8
Using Synthesis Libraries	B-9
Using Verilog Libraries	B-9
Library Verification Process	B-12
Verify Synthesis and Simulation Libraries	B-13
Library Verification Using Formality	B-15
Library Loading Order	B-15
Single-Source Packaging	B-16
Multiple-Source Packaging	B-16
Augmenting a Synthesis (.db) Library	B-16
Augmenting a Simulation (.v) Library	B-16

Appendix C. Appendix C - Reference Lists

Shell Variables C-2

Shell Commands C-10

Index

Preface

This preface includes the following sections:

- [What's New in This Release](#)
- [About This User Guide](#)
- [Customer Support](#)

What's New in This Release

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

To see the *Formality Release Notes*,

1. Go to <http://solvnet.synopsys.com/ReleaseNotes>. (If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
2. Click Formality, then click the release you want in the list that appears at the bottom.

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

This manual is written for engineers who use the Formality product on a UNIX workstation to perform equivalence checking. Additionally, you need to understand the following concepts:

- Logic design and timing principles
- Logic simulation tools

- UNIX operating system

Related Publications

For additional information about Formality, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys electronic software transfer (EST) system
- Documentation on the Web, which is available through SolvNet at <http://solvnet.synopsys.com/DocsOnWeb>
- Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at <http://mediadocs.synopsys.com>
- The documentation installed with the Formality software and available through the Formality Help menu

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

- ESP (see the *Formality ESP User Guide*)
- Design Compiler
- HDL Compiler
- PrimeTime

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates command syntax.
<i>Courier italic</i>	Indicates a user-defined value in Synopsys syntax, such as <i>object_name</i> . (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
Courier bold	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[]	Denotes optional parameters, such as <i>pin1 [pin2 ... pinN]</i>
	Indicates a choice among alternatives, such as <i>low medium high</i> (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
—	Connects terms that are read as a single term by the system, such as <i>set_annotated_delay</i>
Control-c	Indicates a keyboard combination, such as holding down the Control 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.

Customer Support

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

Accessing SolvNet

SolvNet includes the Solv-It electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call With the Support Center.”

To access SolvNet, do the following:

1. Go to the SolvNet Web page at <http://solvnet.synopsys.com>.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click SolvNet Help in the column on the left side of the SolvNet Web page.

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 call to your local support center from the Web by going to <http://solvnet.synopsys.com> (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
 - E-mail support_center@synopsys.com from within North America.
 - Find other local support center e-mail addresses at http://www.synopsys.com/support/support_ctr.
- Telephone your local support center.
 - Call (800) 245-8005 from within the continental United States.
 - Call (650) 584-4200 from Canada.
 - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Introduction to Formality

In this chapter, you are introduced to the Formality application. It includes the following sections:

- [What Is Formality?](#)
- [How Does Formality Fit Into My Design Methodology?](#)
- [What Designs Can I Verify?](#)
- [What Pieces Make Up Formality?](#)
- [General Process Flow](#)
- [Input and Output File Types](#)
- [Concepts](#)

What Is Formality?

Formality is an application that uses formal techniques to prove or disprove the functional equivalence of two designs or two technology libraries. For example, you can use Formality to compare a gate-level netlist to its register transfer level (RTL) source or to a modified version of that gate-level netlist. After the comparison, Formality reports whether the two designs or technology libraries are functionally equivalent. The Formality tool can significantly reduce your design cycle by providing an alternative to simulation for regression testing.

The techniques Formality uses are static and do not require simulation vectors. Consequently, for design verification you only need to provide a functionally correct, or “golden,” design (called the reference design) and a modified version of the design (called the implementation design). By comparing the implementation design against the reference design, you can determine whether they are functionally equivalent to each other. Technology library verification is similar except that each cell in the implementation library is compared against each cell in the reference library one cell at a time.

Today’s design methodology requires regression testing at several points in the design process. Currently, traditional simulation tools, such as event-driven and cycle-based simulators, handle this regression testing. However, 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, slowing down performance.

Formality fits well within established electronic design automation (EDA) methodologies used to create large-scale designs, because it can replace the traditional simulation tools used for regression testing. This replacement, combined with the continued use of static timing analysis tools, gives you two distinct advantages: significantly reduced verification times and complete verification.

Reduced verification times occur because Formality requires no input vectors. Reducing gate-level simulation time means you can spend more time verifying the initial golden RTL design to get better functional coverage. Formality maintains that coverage through all subsequent regressions.

Complete verification, the second advantage, means 100 percent equivalence over the entire vector space. Complete verification is significant because you no longer have to compromise the subset of vectors for gate-level simulation.

The following list summarizes the Formality features:

- Proves two designs or technology libraries are functionally equivalent, faster than verification using event-driven simulators.
- Provides complete verification (not vector-dependent).
- Performs RTL-to-RTL, RTL-to-gate, and gate-to-gate design verifications.

- Performs Verilog-to-database, Verilog-to-Verilog, database-to-database, Verilog-to-SPIICE, and database-to-SPIICE technology library verifications.
- Offers diagnostic capabilities to help you locate and correct functional discrepancies between designs.
- Reads Synopsys internal (.db or .ddc) formats.
- Reads synthesizable SystemVerilog, Verilog, and VHDL.
- Performs automatic hierarchical verification.
- Uses existing Design Compiler technology libraries.
- Saves and restores designs and verification sessions.
- Offers a graphical user interface (GUI) and a shell command-line interface (fm_shell) environment.
- Verifies a wide range of design transforms or modifications, including pipeline retiming and reencoded finite state machines.
- Offers schematic views and isolated “cone of logic” views that support location of design discrepancies.

How Does Formality Fit Into My Design Methodology?

For design verification, Formality fits into a design process exactly the same way a logic simulator fits when it is used for regression testing. Specifically, any time you make a nonfunctional change to a design, you can use Formality to prove that the implementation design is functionally equivalent to the reference design.

After proving the implementation design is equivalent to the reference design, you can use the implementation design as the reference design for the next regression test. By establishing the most recent design as the reference design throughout the process, you minimize the overall verification time. The time is reduced because Formality can compare structurally similar designs faster than structurally dissimilar designs.

Figure 1-1 shows the incremental usage model that best suits the design flow using Formality. The box with the bold border represents verification using Formality.

Figure 1-1 Formality Usage Model

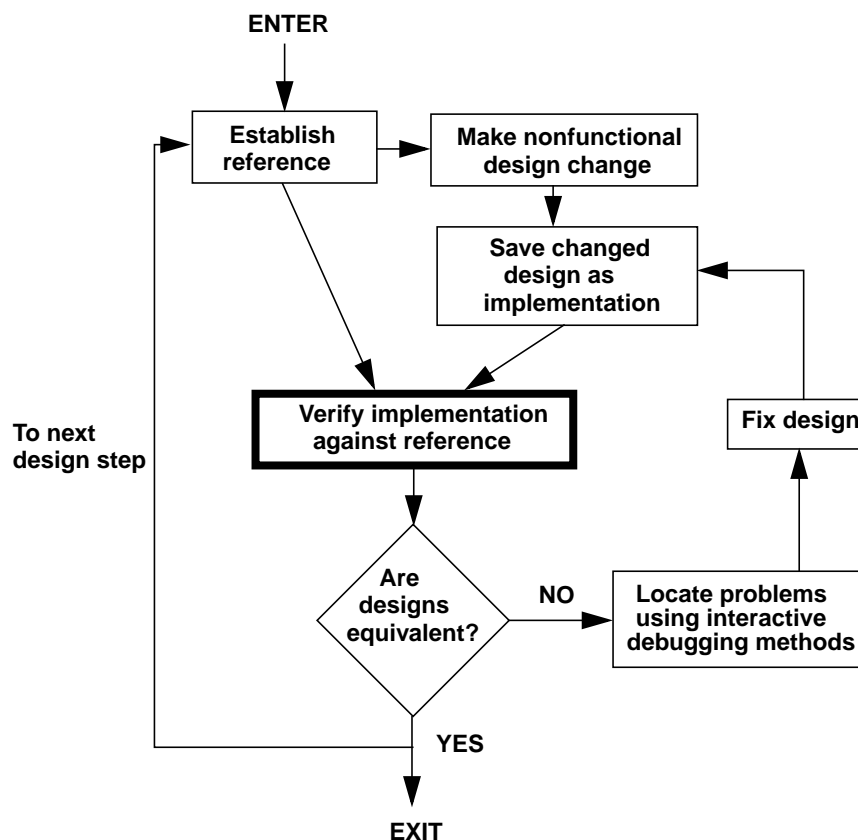


Figure 1-2 shows how Formality fits into a typical ASIC verification methodology. In Figure 1-2, ovals represent data and boxes represent processes. Boxes with bold borders represent verification using Formality.

```
graph TD
    subgraph TopRow [ ]
        direction LR
        A1([RTL Verilog, VHDL, or SystemVerilog])
        A2[RTL functional simulation]
        A3([Test bench])
        A4([Test bench])
        A5([Reference design])
    end
    A1 --> A2
    A2 --> A3
    A2 --> A4
    A2 --> A5

    A1 --> B1[Architectural refinement]
    B1 --> A2

    subgraph MiddleRow [ ]
        direction LR
        A6([RTL Verilog, VHDL, or SystemVerilog])
        A7[Formal verification]
        A8([Reference design])
    end
    A6 --> A7
    A7 --> A8
    A6 --> A5

    A6 --> B2[Synthesis and optimization]
    B2 --> A6

    B2 -- Design Compiler --> A9([Netlist])
    A9 --> A10[Static timing analysis]
    A10 -- PrimeTime --> A9
    A9 --> A11[Formal verification]
    A11 -- Formality --> A12([Reference design])
    A9 --> A13[Scan-chain stitching]

    A13 -- DFT Compiler --> A14([Netlist])
    A14 --> A15[Static timing analysis]
    A15 -- PrimeTime --> A14
    A14 --> A16[Formal verification]
    A16 -- Formality --> A17([Reference design])
    A14 --> A18[Physical design]

    A18 --> A19([Netlist])
    A19 --> A20[Static timing analysis]
    A20 -- PrimeTime --> A19
    A19 --> A21[Formal verification]
    A21 -- Formality --> A22([Reference design])
    A19 --> A23[Timing constraints]

    A23 --> A10
    A23 --> A20
    A23 --> A21
```

What Designs Can I Verify?

This section presents fundamental design requirements and describes situations where Formality works particularly well.

Design Requirements

The reference design and implementation design that you use with Formality must meet the following fundamental requirements:

- Design files must be in the Synopsys internal database (.db or .ddc) format or must use only synthesizable SystemVerilog, Verilog, or VHDL constructs accepted by (V)HDL Compiler (Presto). Formality can also read designs in Milkyway formats.
- Designs should use a synchronous design style. They should not contain state-holding loops implemented as combinational logic.
- Top-level I/O ports, sequential components, and black box components in both the reference design and implementation design must be aligned structurally. Formality automatically matches as many ports and components as possible between the implementation design and reference design during verification. If Formality has not automatically determined a match, you can use commands to create these matches.

Design Types

This section presents examples of situations where Formality offers a good solution for regression testing.

Verification of Two RTL Designs

When you make an architectural change to an RTL design, use Formality to verify that you did not change how the design functions. In this situation, you are verifying an RTL implementation design against an original RTL reference design.

Situations where this type of regression testing becomes necessary include

- Adding clock-gating circuitry for power reduction
- Restructuring critical paths
- Reorganizing logic for area reduction

Verification of an RTL Design and a Gate-Level Design

You can verify an RTL design against a gate-level design at several points in the design methodology. For example, it is important to verify the gate-level implementation design that results from synthesis against the golden RTL design for functional equivalence. This gate-level design becomes the golden design used in verifying subsequent implementation designs.

Another example is when you make minor functional changes in the gate-level netlist and you simultaneously update the RTL source without using synthesis. In this case, you can use Formality to verify that the changes made in the RTL source match the current implementation design.

Verification becomes important in situations such as the following:

- Maintaining the RTL design as the source design for future design revisions

- Using the RTL design in system-level simulations
- Using the RTL design as the official documentation of design functions
- Synthesizing the RTL design

Note:

Formality does not support programmable logic array (PLA), state table (.st), or equation table (.e) formats. Therefore, do not write your RTL to these formats when using Formality.

Verification of Two Gate-Level Designs

You can use Formality to verify the functional equivalence of the design when you produce a new gate-level implementation by making nonfunctional changes.

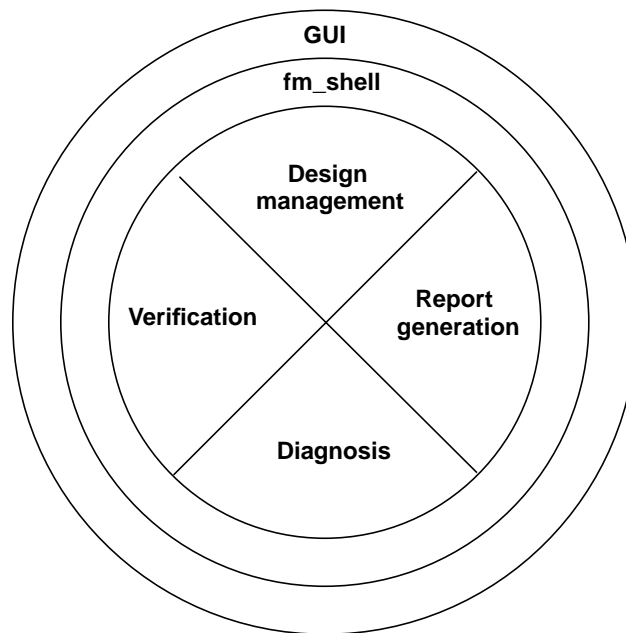
The following typical changes result in a new implementation design, but do not affect functional capabilities:

- Adding test logic (scan circuitry) to a design
- Reordering a scan chain in a design
- Inserting a clock tree into a design
- Adding I/O pads to the netlist
- Performing design layout
- Performing flattening and cell sizing
- Creating a netlist for hardware acceleration or emulation

What Pieces Make Up Formality?

Formality consists of four functional areas surrounded by a command-line interface (`fm_shell`) and a graphical user interface (GUI). [Figure 1-3](#) illustrates these areas.

Figure 1-3 Functional Areas Within Formality



This list describes the user-visible areas shown in [Figure 1-3](#).

GUI

The GUI offers a windows-based, menu-driven interface that lets you access most Formality functions. Through the GUI, you can perform design management, verify designs, generate reports, and diagnose and debug designs.

For more information about the GUI, see [“Using the Formality GUI Environment” on page 3-17](#).

fm_shell

The fm_shell command-line interface offers the same commands as the GUI and some unique functions. From fm_shell you can do everything that you can with the GUI, except view schematic representations of designs and view logic cones.

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

Design management

The design management functions let you set up and control the verification process. For example, you can load designs, establish environmental parameters, save and restore verification sessions, and help Formality match points in the designs.

For more information about design management, see [Chapter 5, “Preparing the Design for Verification,”](#) and [Chapter 7, “Debugging Failed Design Verifications.”](#)

Verification

Verification is the primary function of Formality. By default, Formality checks for design consistency when you verify a design or technology library. An implementation design is consistent with a reference design when it is functionally equivalent. Therefore, a don't care state (X) in the reference design can be represented by either a 0 or 1 state in the implementation design.

You can also test for design or technology library equality. An implementation design is equivalent to a reference design when it meets both of the following requirements:

- It is consistent with the reference design.
- Its set of don't care vectors matches that of the reference design set.

For more information, see [“Working With Equivalences” on page 5-25](#). For procedures that describe how to perform verification, see [Chapter 6, “Compare Point Matching and Verification.”](#)

Diagnosis

You use the diagnosis function when design verification fails. Diagnosis produces information that helps you isolate areas in the design that could cause a failed verification. For more information, see [Chapter 7, “Debugging Failed Design Verifications.”](#)

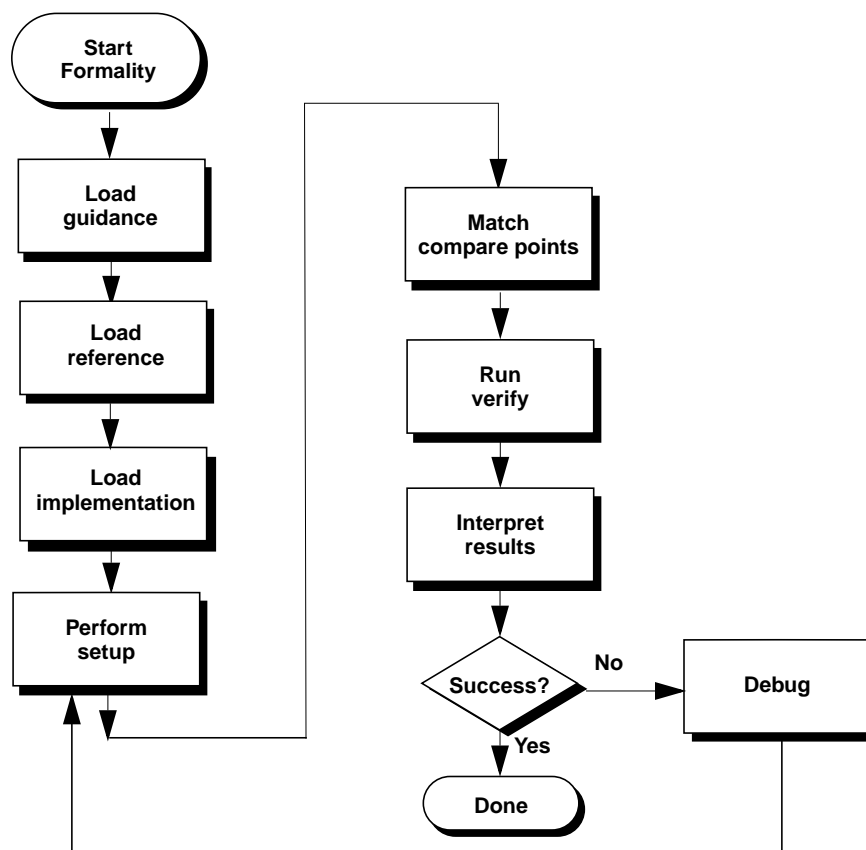
Report generation

Formality allows you to generate several types of reports. These reports provide information about the most recent verification, most recent diagnosis, environment parameters, or parameters that affect a particular design.

General Process Flow

The flow chart in [Figure 1-4](#) provides an overall guide to the Formality design verification process. Starting with [Chapter 3, “Starting Formality,”](#) each chapter describes one or more steps in detail. This chart appears in the beginning of each applicable chapter to remind you where you are in the process.

Figure 1-4 Design Verification Process Flow Overview



Note:

You generally use the application commands from `fm_shell` for the steps up to "Run verify." Thereafter you use the GUI. Unless otherwise noted, the instructions in this document are for both `fm_shell` and the GUI.

Technology (or cell) library verification is a self-contained process that is described in [Chapter 8, "Technology Library Verification."](#)

Input and Output File Types

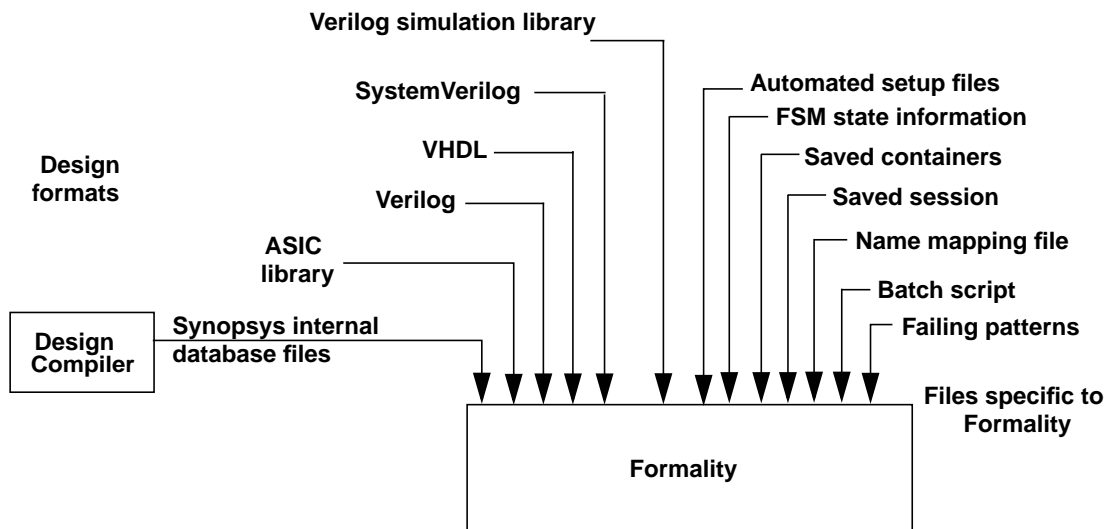
This section describes the types of files that the Formality tool accepts and generates for design verification. It includes the following sections:

- [Input](#)
- [Output](#)
- [Synopsys Setup File](#)

Input

Formality accepts several types of files as input. File formats consist of files that are specific to Formality and design formats. [Figure 1-5](#) illustrates Formality input.

Figure 1-5 Supported Input



Synopsys internal database files

Files formatted in the Synopsys internal database design format (.db and .ddc files). The database format is the default output format used for the Design Compiler tool. For information about reading Synopsys database files into Formality, see [“Reading in Libraries and Designs” on page 4-3](#).

ASIC library

Groups of cell-sized designs in the Synopsys internal database format. Libraries are described in detail in [“Libraries” on page-19](#).

Verilog

Verilog hardware description language design files. This design format specifies digital systems at a wide range of levels of abstraction. Formality supports the same synthesizable subset of Verilog as Design Compiler. For information about reading Verilog files into Formality, see [“Reading in Libraries and Designs” on page 4-3](#).

VHDL

VHDL design files. This design format specifies digital systems at a high level of abstraction. For information about reading VHDL files into Formality, see [“Reading in Libraries and Designs” on page 4-3](#).

SystemVerilog

SystemVerilog design files. This design format specifies digital systems at a wide range of levels of abstraction. For information about reading SystemVerilog files into Formality, see [“Reading in Libraries and Designs” on page 4-3](#).

Verilog simulation library

Verilog simulation library files. You can read the cell definition information into a technology library. The library reader extracts the pertinent information to determine the gate-level behavior of the design, and generates a netlist that represents the functionality of the Verilog library cells. Libraries are described in detail in [“Libraries” on page 1-19](#).

Automated setup files

Setup information in the form of guidance or automated setup files to help Formality understand and verify the transformations between designs. Automated setup files are described in detail in [“Working With Automated Setup Files” on page 5-72](#).

FSM state information

A file that contains flip-flop names and their encoding for finite state machines (FSMs). This file defines FSMs so that Formality can create the compare points necessary for verification. For information about reading these types of files, see [“Working With Reencoded Finite State Machines” on page 5-53](#).

Saved containers

The Formality internal representation of a container and its contents. You supply this file when you want to restore a previously saved container. For information about saving containers, see [“Saving the Entire Formality Session” on page 5-86](#).

Saved session

A file that contains the state of the verification session. You supply this file when you want to restore a previously saved Formality session. For information about restoring a Formality session, see [“Restoring a Session” on page 5-89](#).

Name mapping file

A file that contains one-to-one name mappings that Formality applies to specific designs. Formality performs a verification based on compare points composed of comparable design objects. Sometimes the naming scheme for design objects in an implementation design deviates measurably from that of the reference design. In such cases, a mapping file can be used to match the design objects. For information about mapping names between designs, see [“Unmatched Points” on page 7-15](#).

Batch script

A file that contains Formality shell commands. Because Formality supports batch mode operation, you can supply a file that contains valid Formality shell commands to direct the verification process. For information about preparing a batch script, see [“Using Batch Jobs” on page 6-31](#).

Failing patterns

A file that contains failing input vectors for the most recent failed verification or diagnosis, or the most recent application of a set of previously saved failing patterns. Formality uses this file as input when simulating previously failing patterns. For information about simulating patterns, see [“Running Previously Saved Failing Patterns” on page 7-48](#).

Libraries

Libraries are collections of designs. When you read design data into Formality, it is grouped into libraries. Formality supports two types of libraries: design libraries and technology libraries.

Design library

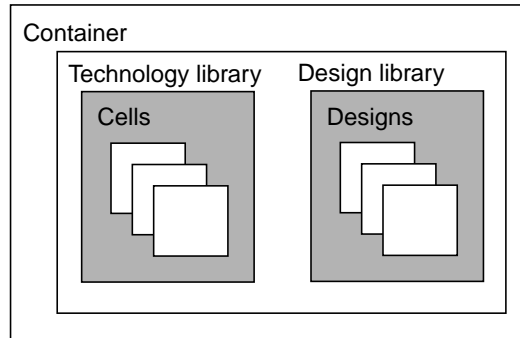
A collection of designs usually associated with a single design effort. For example, a design library might contain individual designs that represent the parts of a hierarchical design: core, control_block, mux_out, and mux_in. Depending on the size and complexity of the top-level design, it might be better to use more than one design library to organize the design data.

Technology library

A collection of cells usually associated with a particular vendor and design technology. For example, when you create a new container in Formality, it automatically loads in the generic technology (GTECH) library. There are two types of technology libraries: shared and unshared. Shared technology libraries are visible in every container and are automatically loaded into every container subsequently created during the Formality session. Unshared technology libraries are loaded into a particular container.

Figure 1-6 illustrates the library concept in the Formality environment. The figure shows one technology library and one design library in the container. Each container can have any number of such libraries.

Figure 1-6 Libraries



In `fm_shell`, you use the Formality read-type commands to load libraries. In some cases, a command option determines whether the data is read in as a technology library or as a design library.

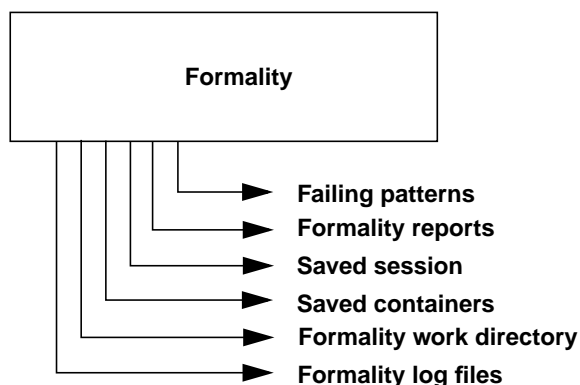
When specifying libraries in `fm_shell`, you provide a library ID. A library ID is like a path name that specifies the container in which the library resides and the name of the library.

In the GUI, you view libraries directly in container windows. You can expand and collapse libraries to view individual designs and cells.

Output

Formality generates several types of output files, as illustrated in Figure 1-7.

Figure 1-7 Generated Output



Failing patterns

A file that contains failing input vectors of the most recent failed verification or diagnosis, or the most recent application of a set of previously saved failing patterns. For information about simulating previously failing patterns, see [“Running Previously Saved Failing Patterns” on page 7-48](#).

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. For information about saving a Formality session, see [“Saving the Entire Formality Session” on page 5-86](#).

Saved containers

The Formality internal representation of a container. You create these files by saving individual containers. For information about saving containers, see [“Saving Containers” on page 5-86](#).

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

Formality maintains two log files: `formality.log` and `fm_shell_command.log`. The `formality.log` file contains verbose information not printed to the transcript. For example, during verification the transcript might print an informational 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 run during the session.

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  
formalityn.log  
fm_shell_commandn.log
```

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.

Controlling File Names Generated by Formality

Formality allows you to control the naming of its files and directory names. 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 suffix` command to specify unique file names. Formality constructs the file names and directories as follows:

- `formality_suffix.log`
- `fm_shell_command_suffix.log`
- `FM_WORK_suffix`

In addition, the `-overwrite` option allows you to overwrite existing files. If you use the `-name_suffix` 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 `-overwrite` option with the `fm_shell` 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`

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 automatically set variables to your preferred settings. The following list shows the order in which Formality reads the files:

1. 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. You create this `.synopsys_fm.setup` file, and it applies to all sessions started by you.
3. The directory where you have invoked Formality (current working directory). You create this `.synopsys_fm.setup` file and customize it for a particular design.

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

Concepts

This section presents key concepts that help you effectively use Formality. It includes the following sections:

- [Compare Points](#)
- [Compare Rules](#)
- [Containers](#)

- [Design Equivalence](#)
- [Logic Cones](#)
- [Reference Design and Implementation Design](#)
- [Solvers](#)

Compare Points

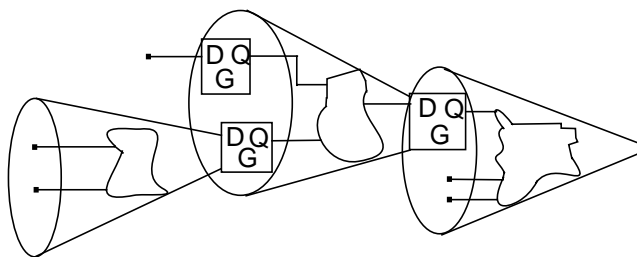
A compare point is a design object used as a combinational logic endpoint during verification. A compare point can be an output port, register, latch, black box input pin, or net driven by multiple drivers.

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

- 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 1-8](#).

Figure 1-8 Cone of Logic



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 passing status. If all compare points in the reference design pass verification, the final verification result for the entire design is a successful verification.

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 [“How Formality Matches Compare Points” on page 6-10](#).

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. However, the following cases do not require a one-to-one correspondence to attain complete verification when you are testing for design consistency:

- Implementation design contains extra primary outputs.
- Either the implementation 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.

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

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

For more information, see [“Matching Compare Points” on page 6-3](#).

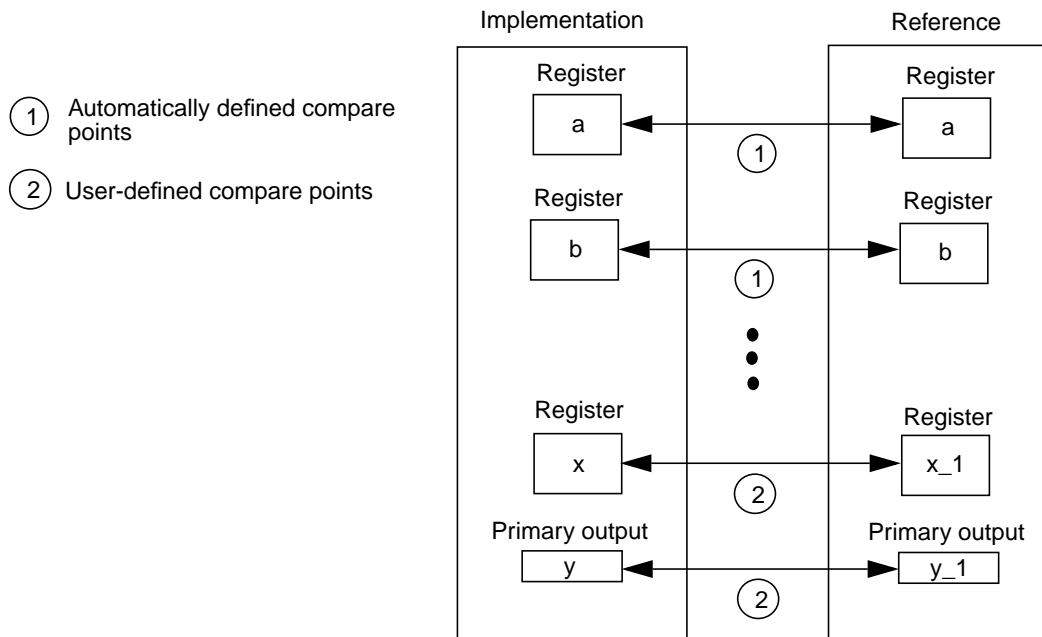
Unmatched design objects from either the implementation or reference design are reported as failing compare points, with a note indicating that there is no comparable design object in the reference design.

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 [“Unmatched Points” on page 7-15](#).

[Figure 1-9](#) 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 1-9 Constructing Compare Points



For compare point status messages, see [“Reporting and Interpreting Results” on page 6-34](#).

Compare Rules

A compare rule is a name translation rule applied by Formality during compare point creation. Compare rules are one of many methods that allow you to map object names between the implementation and reference designs. For example, suppose that a certain bus in the reference design has a different name in the implementation, and Formality cannot match the two objects using structural or signature analysis. You can define a compare rule to enable Formality to

correctly create compare points. Application of the compare rule maps the bus names in the reference design to those in the implementation.

Compare rules have a regular expression syntax identical to that supported by the `test_compare_rule` command. Therefore, you can use the `test_compare_rule` command to ensure your compare rules behave as desired.

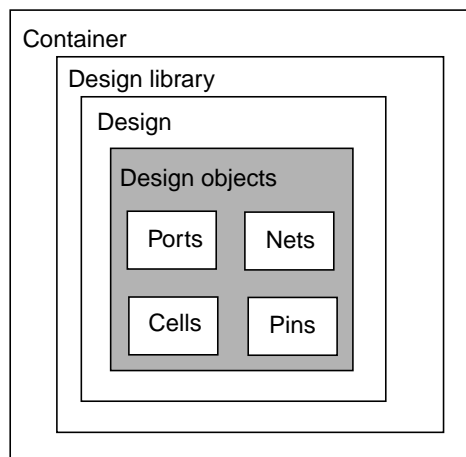
Containers

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 [“Reading in Libraries and Designs” on page 4-3](#).

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 1-10](#) and [Figure 1-11](#) illustrate the concept of containers.

Figure 1-10 Containers in a Hierarchical Design

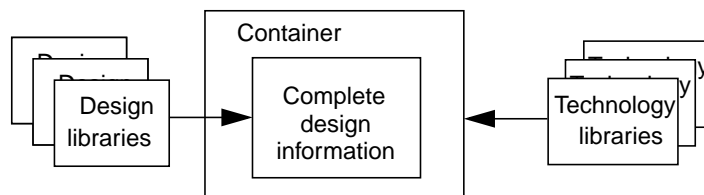


In general, to perform a design comparison, you should load all of the information about one design into a container (the reference), and all the information about the other design into another container (the implementation).

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 1-11](#) illustrates this concept.

Figure 1-11 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 4-29](#).

Design Equivalence

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 in the reference design, verification passes only when there is an X at the equivalent point in the implementation design.

For information about don’t care conditions, see [“Using Don’t Care Cells” on page 5-3](#) and [“Working With Equivalences” on page 5-25](#).

The type of design equivalence is also called the verification mode. By default, verification mode tests for design consistency, which is adequate in most cases. For the rare occasion that it is not, you can test for design equality.

Sometimes conditions exist where one design (design A) is consistent with a second design (design B), but design B is not consistent with design A. For example, design B might have a don't care condition that is implemented as a 0 or 1 in design A.

Here, design A gives an output value of X for the d=2'b11 input condition.

```
model A (d, q);
input [1:0] d;
output q;
reg q;

always @ (d)
case (d)
  2'b00: q = 1'b0 ;
  2'b01: q = 1'b0 ;
  2'b10: q = 1'b1 ;
  2'b11: q = 1'bX ;
endcase

endmodule
```

Here, design B gives an output value of 1 for the d=2'b11 input condition.

```
model A (d, q);
model A (d, q);
input [1:0] d;
output q;
reg q;

always @ (d)
case (d)
```

```
2'b00: q = 1'b0 ;
2'b01: q = 1'b1 ;
2'b10: q = 1'b1 ;
2'b11: q = 1'b1 ;
endcase

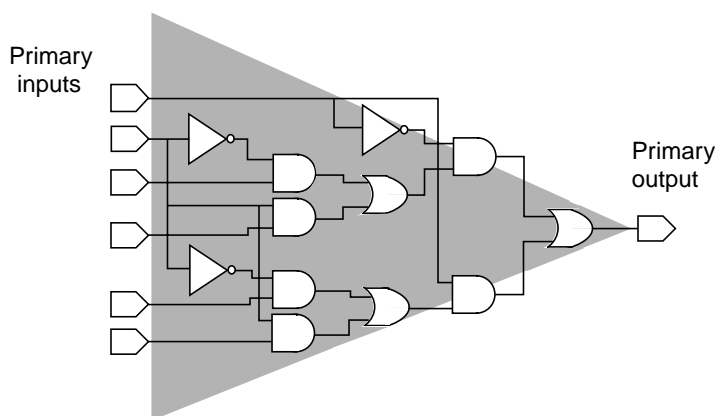
endmodule
```

If you run a verification with design A as the reference and design B as the implementation, verification passes (X in the reference versus 1 in the implementation). However, if you run a verification with design B as the reference and design A as the implementation, verification fails (1 in the reference versus X in the implementation).

Logic Cones

A logic cone consists of combinational logic originating from a particular 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. These design objects are primary outputs, internal registers, black box input pins, and 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 and those that Formality uses to create compare points. [Figure 1-12](#) illustrates the logic cone concept.

Figure 1-12 Logic Cone



In [Figure 1-12](#), the design object of concern is a primary output. Formality compares this design object (compare point) to a comparable object (compare point) in a second 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.

Reference Design and Implementation Design

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.

Note:

For technology library verification, the reference and implementation definitions do not apply because you always test for equality. In addition, you are generally specifying different types of libraries against one another, for example, a synthesis library against a SPICE library.

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 at a minimum. Conversely, working through an entire design methodology and then verifying the sign-off netlist against the original VHDL can result in difficult verifications and in longer overall verification times.

In the fm_shell or GUI environment, you can designate a design you have read into Formality as either the implementation or 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.

Solvers

Formality enlists various solvers before and during verification to attempt to prove the equivalence or nonequivalence of two designs using algorithms particular to the specific solver. To check for equivalence, the solvers look for internal equivalences, redundancies, constants, and so forth.

For example, the datapath solver uses a particular algorithm to solve multipliers in the preverification stage that can significantly reduce the entire verification runtime. When the solver successfully

preverifies a multiplier in your design, Formality converts it to a black box for the rest of verification. If the datapath solver is unable to preverify your multiplier, Formality returns an inconclusive result in the transcript.

For suggestions about preparing your designs for successful verification, see [“Handling Designs When Verification Is Incomplete” on page 7-4](#).

2

Quick Start With Formality

This chapter explains how to start and run Formality. The quick-start tutorial demonstrates the steps followed during a typical Formality design verification session. This chapter includes the following sections:

- [Before You Start](#)
- [Invoking the Formality Shell and GUI](#)
- [Graphical User Interface](#)
- [Verifying fifo.vg Against fifo.v](#)
- [Verifying fifo_with_scan.v Against fifo_mod.vg](#)
- [Verifying fifo_jtag.v Against fifo_with_scan.v](#)
- [Debugging Using Diagnosis](#)
- [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 and GUI

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, “Starting Formality.”](#)

Graphical User Interface

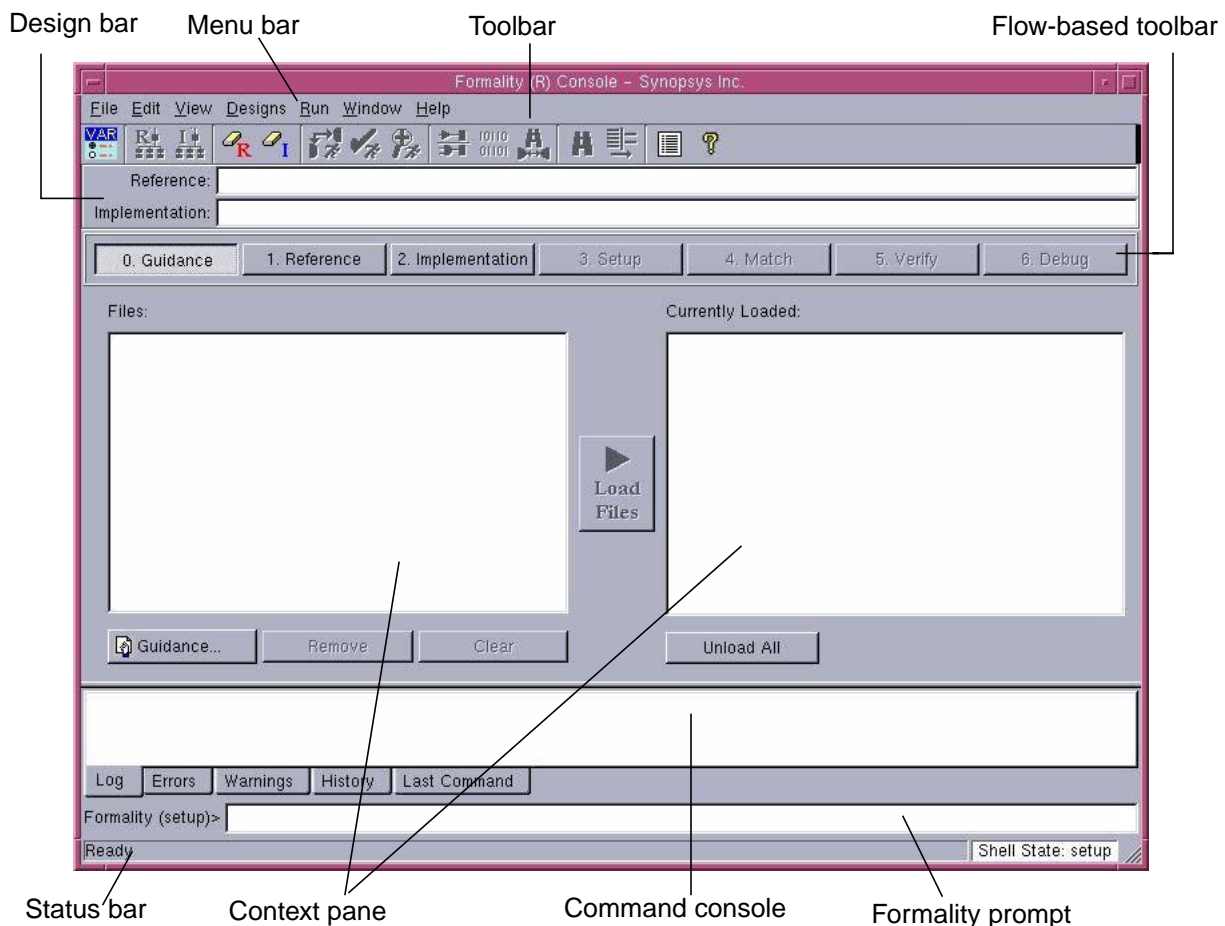
The main GUI session window contains the following window areas, as shown in [Figure 2-1 on page 2-5](#):

- 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: Easy-access options to common GUI commands. The contents of the toolbar change depending on the view displayed in the context pane.
- Flow-based toolbar: Options that indicate the correct flow to employ to perform formal verification. The options become 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 then 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. This is also where you view reports.
- Command console: Displays transcripts and other information, depending on the option selected below this area.
- Formality prompt: Text box that allows you to enter Formality prompts and environment variables that are not available through the GUI interface.
- Status bar: Current state of the tool.

Figure 2-1 GUI Session Window



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`.

Note:

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

Loading the Automated Setup File

Before specifying the reference and implementation designs, you can optionally load an automated setup file (`.svf`) into Formality. The automated setup 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 automated setup file that you load, Formality processes the content and stores the information to use during the name-based compare point matching period.

To load the automated setup file, do the following:

1. From the Guidance tab, click **Guidance**, then select the file.
2. Click **Load Files**.

Note:

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

Specifying the Reference Design

Figure 2-1 on page 2-5 shows the flow-based toolbar. The first step, which is optional, is to load the automated setup file from the Guidance tab. The next step is to load the reference. From the Reference tab, you do the following:

1. Read Design Files.
2. Read DB (technology) Libraries (optional).
3. Set Top 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`.

Note:

Because using the Guidance tab to load the automated setup file is optional, the tutorial skips this step.

To specify the reference design, do the following:

1. Click the Reference tab if its work area is not already displayed in the context pane.

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 RTL subdirectory where you copied the tutorial directory, then select the `fifo.v` file containing the reference design. Click Open.
4. Click Options.

The Set Verilog Read Options dialog box appears.

5. In the DesignWare root directory (hdlin_dwroot), browse to or specify the path name to the Synopsys software root directory.

This step is necessary because fifo.v contains a DesignWare instantiated RAM block. As needed, enter `echo $SYNOPSYS` at the Formality prompt to obtain the path name of the root directory.

6. From the Options pane of the VCS Style Options tab, select Library Directory (-y), and in the Enter Directory Name box browse to or specify the RTL subdirectory. Click Add.
7. In the Options pane, select Library Extension (+libext), and in the Enter File Extension box, type `.v`. Click Add.

The -y and +libext options you selected are Verilog simulator (VCS) options, where -y specifies to look in the current directory for any unresolved modules and +libext specifies to look in files with the specified extension (.v in this case).

8. Click OK to exit the dialog box, then click Load Files.

The fifo.v file appears as the loaded reference design file.

9. Click the Set Top Design tab.

In this case, you skip the Read DB Libraries tab because the fifo.v design does not require a technology library. Technology libraries contain cells to which the netlist must be mapped. The fifo.v design does not require mapping.

10. From the Choose a library pane, select the WORK library.

This pane lists all the loaded libraries, including the DesignWare (DW*) and GTECH library, that contain the GTECH components required to map RTL.

11. From the Choose a design pane, select fifo, the name of the top-level design.

12. Click Set Top.

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.

Before you click Set Top, only the top-level design (fifo) is listed; the lower-level modules are not listed. The reason for this is that during linking, the netlist reader searches for the unresolved modules under the directory specified using the VCS -y option.

After you click Set Top, the choose-a-design pane lists the entire design hierarchy.

A green check mark appears on the Reference tab located on the flow-based toolbar, indicating that you successfully specified the reference design. You can now specify the implementation design.

Specifying the Implementation Design

The procedure for specifying the implementation design is identical to that for specifying the reference design. From the Implementation tab, do the following:

1. Read Design Files.
2. Read DB (technology) Libraries (optional).
3. Set Top Design.

To specify the implementation design, do the following:

1. On the flow-based toolbar, click the Implementation 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 from the RTL directory to the GATE directory, then select the fifo.vg file containing the implementation design. Click Open.
4. Click Load Files.

The fifo.vg file appears as the loaded implementation design.

5. Click the Read DB Libraries tab, then click the DB option.

The Add DB Files dialog box appears. Unlike the fifo.v reference design, fifo.vg requires a technology library for mapping.

6. Navigate to the LIB directory, then select the technology library named lsi_10k.db. Click Open.

7. Click Load Files.

8. Click the Set Top Design tab.

9. From the Choose a library pane, select WORK.

10. From the Choose a design pane, select the top-level design, fifo.

11. Click Set Top.

Formality links and elaborates all the files and reports if there are any missing files. Because this tutorial uses a DesignWare RAM, the tool searches for technology-dependent modules from the lsi_10k technology library.

A green check mark appears at the Implementation option located on the flow-based toolbar, indicating that you successfully specified the implementation design. You can go to the setup stage, as necessary.

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 5, “Preparing the Design for Verification.”](#)

Compare Point Matching

Compare point matching 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 verification fails. Matching ensures that there are no mismatched logic cones and allows Formality to proceed with verifying the implementation design for functionality.

For conceptual information about compare points, see [“Compare Points” on page 1-25](#). For more information about how Formality matches compare points, see [“Matching Compare Points” on page 6-3](#).

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

1. On the flow-based toolbar, click the Match tab.
2. Click Run Matching.

When matching is completed, the results appear in the command console. You can also click the Summary tab to view the results. In this case, there are no unmatched compare points to debug.

If the summary indicated unmatched compare points, you would click the Unmatched Points tab to view them. For example, unmatched compare points in the reference design might indicate extra registers that were optimized away during synthesis. If you expect optimizations, you can ignore the extra (unmatched) compare points in the reference design.

3. Click OK to close the Information dialog box.

For more information about compare point matching, including how to debug unmatched compare points, see [Chapter 6, “Compare Point Matching and Verification,”](#) and [Chapter 7, “Debugging Failed Design Verifications.”](#)

Verifying the Designs

You are now ready to verify the features of `fifo.vg` against its reference design, `fifo.v`.

To verify `fifo.vg` against `fifo.v`, do the following:

1. On the flow-based toolbar, click the Verify tab.
2. Click Verify.

Verification begins, as shown by the scrolling transcript in the command status area. Verification fails, meaning that the implementation design, `fifo.vg`, was transformed during synthesis, rendering it unequivalent to the reference design, `fifo.v`.

3. Click OK to close the dialog box notifying you of the failed verification.

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.

To debug the implementation design, `fifo.vg`, do the following:

1. 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.

2. To run diagnosis on the failing points, click Diagnose.

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

Note:

Although this is not shown in this tutorial, if while debugging 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 click Diagnose Selected Points. This might help to direct diagnosis down to a specific section of the design.

3. From the Error Candidates Window, right-click on the error U81 and select View Logic Cones.

You see a list of related failing points for that error, from which you select one of those points (for this example, use `push_logic/pop_count_svd_reg[0]`) and then double-click it to bring up 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, click Isolate Error Candidate Pruning Mode in the cone view. This prunes away all the logic and shows the error inverter.

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 orange. The corresponding matching region is highlighted in yellow.
4. Observe the patterns annotated on the CLK net. The reference design shows logic 0, while the implementation design shows logic 1.

To discover the cause for this functional difference, do the following:

- Select the net in the implementation design.
- Right-click and select Isolate Subcone.
- Select the net in the reference design.
- Right-click and select Isolate Subcone.

The screens change to display just the net in question. Notice that the logic driving the implementation CLK pin includes an inverter. During synthesis an inverter might have been inserted to fix hold time problems.

You can zoom in by clicking the Zoom In Tool toolbar option and then clicking in the schematic. Deselect the option to return to the pointer.

You can copy selected objects in design and cone schematics. From the context pane, you can highlight the object, select Edit, then selecting Copy Name, Copy Reference Name, or Copy Implementation Name. You can paste these names into the Formality prompt or any other editable text box by pressing Control-v or by right-clicking the mouse and choosing Paste.

5. 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.

6. After closing the Cone Schematics window, verify the corrected implementation design, `fifo_mod.vg`, against the reference design. First, specify `fifo_mod.vg` again as the new implementation design as follows:
 - On the flow-based toolbar, click Implementation.
 - From the Read Design Files tab, click the Verilog tab, then click the Verilog option.
 - Click Yes to remove the current implementation design data.

Note:

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

- Navigate to the GATE subdirectory, and select the fifo_mod.vg file.
- Click Open.
- Click Load Files.

Skip the Read DB Libraries tab because the technology library is shared.

- Click the Set Top Design tab.
 - Make sure that WORK and fifo are selected and click Set Top.
7. As before, skip the Setup step. In this case, you can also skip the Match step because you did not change the setup, which could alter 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.
 8. From the Verify tab, click Verify.

Formality performs automatic compare point matching prior to 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, select Remove Reference, then click Yes.

2. From the Designs menu, select Remove Implementation, then click Yes.

Note:

Clicking Yes permanently removes the current reference and implementation data. In practice, be sure to save (as necessary) prior to removing 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 selecting File > Save Session. To invoke that session again, select File > Restore Session.

In this portion of the tutorial, you specify the successfully verified netlist, `fifo_mod.vg`, as the reference design. You then verify a design that went through a design transformation against it. The `fifo_with_scan.v` implementation design, as the name suggests, had scan logic inserted.

To go through the verification steps (reference, implementation, setup, match, verify, and debug) in one continuous flow, do the following:

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.
3. Navigate to the `fifo_mod.vg` file in the GATE directory, click Open, then click Load Files.
4. Click the Read DB Libraries tab and make sure that the “Read as a shared library” option is selected.

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

If you use a Verilog or VHDL technology library, you must specify the `read_verilog -tech` or `read_vhdl -tech` command at the Formality prompt, because they are not shared libraries.

5. Click DB, then navigate to the technology library named `lsi_10k.db` in the LIB directory.
6. Click Open.
7. Click Load Files.

8. Click the Set Top Design tab and ensure that the fifo design inside the WORK library is selected as the top-level design.

9. Click Set Top.

Next you specify the implementation design, a procedure similar to the one described in [“Specifying the Implementation Design” on page 2-9](#).

10. On the flow-based toolbar, click the Implementation tab.

The Read Design Files and Verilog tab are selected by default.

11. Click Verilog.

12. In the Add Verilog Files dialog box, navigate to the fifo_with_scan.v design file located in the GATE_WITH_SCAN directory. Click Open.

13. Click Load Files.

14. Click the Set Top Design tab and make sure that the fifo design inside the WORK library is selected as the top-level design.

15. Click Set Top.

You skipped the Read DB Libraries step because you had previously specified lsi_10k.db as a shared technology library.

16. Click the Setup tab.

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

17. Click the Constants tab, then 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.

18. Click the Implementation tab and make sure that `fifo` is selected and that Ports appears in the drop down box near the top of the display area.

19. Scroll or search for the port named `test_se` and select it.

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

20. In the Constant Value area at the bottom of the dialog box, select 0, then 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. Notice that `test_se` now appears in the Command console.

21. On the flow-based toolbar, click the Match tab, then click Run Matching.

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

22. Click OK to remove the Information dialog box, then select the Unmatched Points tab.

You see a report on the unmatched point, `test_se`. It is an extra compare point 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 2-13](#).

23. On the flow-based toolbar, click the Verify tab, then 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, then click Yes.
2. From the Designs menu, select Remove Implementation, then click Yes.

Note:

Clicking Yes permanently removes the current reference and implementation data. In practice, be sure to save (as necessary) prior to removing 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_jtag.v` Against `fifo_with_scan.v`

Next, you specify the successfully verified scan-inserted netlist, `fifo_with_scan.v`, as the reference design. You then verify a design that went through a different type of design transformation against it. The `fifo_jtag.v` implementation design includes a JTAG insertion and a scan insertion.

To go through the verification steps (reference, implementation, setup, match, verify, and debug) in one continuous flow, do the following:

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 `fifo_with_scan.v` file in the `GATE_WITH_SCAN` directory, highlight it, and click Open.
4. Click Load Files.
5. From the Read DB Libraries tab, make sure that the “Read as a shared library” option is selected.

Because this is a gate-to-gate verification, the technology library needs to be available for both the `fifo_with_scan.v` and `fifo_jtag.v` files.

6. Click DB, then navigate to and highlight the technology library file, `lsi_10k.db`, in the LIB directory. Click Open.
7. Click Load Files.

8. From the Set Top Design tab, make sure that the fifo design inside the WORK library is selected as the top-level design, then click Set Top.
9. On the flow-based toolbar, click the Implementation tab.
10. From the Verilog tab, click Verilog.

The Add Verilog Files dialog box appears.

11. Navigate to the fifo_jtag.v design file located in the GATE_WITH_JTAG directory, highlight it, and click Open.
12. Click Load Files.
13. From the Set Top Design tab, ensure that the fifo design inside the WORK library is selected as the top-level design and click Set Top.

Notice that because of the inserted JTAG modules listed at the top of the choose-a-design pane, you might need to scroll down to find the fifo design.

Note:

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

You skipped the Read DB Libraries step because you had previously specified lsi_10k.db as a shared technology library.

14. On the flow-based toolbar, click the Setup tab.

For this verification you must disable the scan in `fifo_with_scan.v` just as you did in the previous section of the tutorial. Remember that this design is now the reference design. You must also disable JTAG signals in the implementation design.

15. From the Constants tab, click Set.

The Set Constant dialog box appears with the Reference tab selected.

16. Make sure that `fifo` is selected and that Ports appears in the drop down box near the top of the display area.

17. Scroll or search for the port named `test_se` and select it.

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

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

19. Click the Implementation tab and make sure that `fifo` is selected and that Ports appears in the drop down box near the top of the display area.

20. Repeat steps 17 and 18 to disable the `test_se` test signal for the implementation design.

21. In a similar process, disable the JTAG signals, `jtag_trst`, and `jtag_tms`, by setting them to constant 0. Click OK to close the dialog box.

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

22. On the flow-based toolbar, select the Match tab and then click Run Matching.

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

23. Click OK to close the Information dialog box.

24. From the Unmatched Points tab, evaluate the compare points.

You see that the extra compare points are located in the implementation design and 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 on to verification.

25. On the flow-based toolbar, click the Verify tab, then 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 (this section assumes 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. Run a diagnosis on all of the failing points listed in this window by clicking Diagnose.

Note:

After clicking Diagnose, 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 click 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 the recommended ones 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, then select 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, “Starting Formality,”](#) which describes the user interfaces and describes how to invoke the tool.
- [Chapter 4, “Setting Basic Elements for Design Verification,”](#) which describes how to read in designs and libraries, and how to define the reference and implementation designs.
- [Chapter 5, “Preparing the Design for Verification,”](#) which describes how to set design-specific parameters to help Formality perform verification and to optimize your design for verification.
- [Chapter 6, “Compare Point Matching and Verification,”](#) which describes how to match compare points and perform verification.
- [Chapter 7, “Debugging Failed Design Verifications,”](#) which describes diagnostic procedures that can help you locate areas in the design that caused failure.
- [Chapter 8, “Technology Library Verification,”](#) which describes how to compare two technology libraries.
- [Appendix A, “Appendix A - 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.
- [Appendix B, “Appendix B - Formality Library Support,”](#) which describes how to verify and modify technology libraries to make them suitable for Formality to perform equivalence checking.

- [Appendix C, “Appendix C - Reference Lists,”](#) which provides reference tables listing Formality application commands and environment variables.

3

Starting Formality

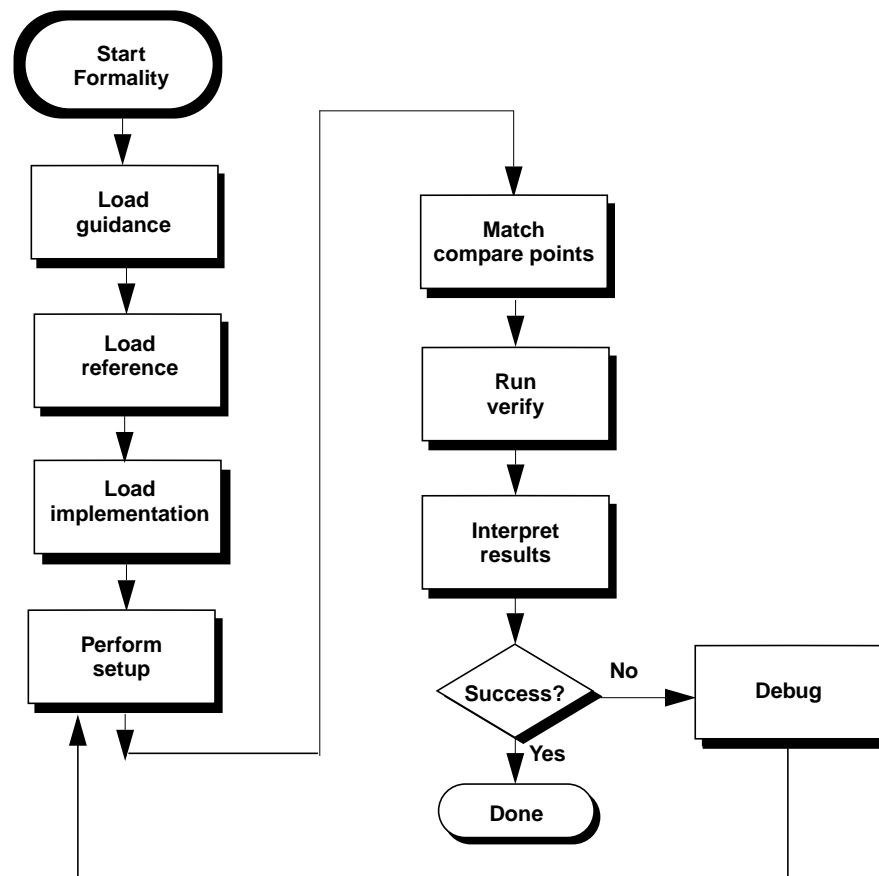
Formality offers two working environments: the Formality shell (a command-line-based user interface) and the Formality GUI (a graphical windows-based interface). This chapter describes how to invoke these environments and how to use interface elements, such as the command log file and the help facility.

This chapter includes the following sections:

- [Invoking Formality](#)
- [Formality Shell Environment](#)
- [Using the Formality GUI Environment](#)
- [General Formality Usage Options](#)

[Figure 3-1](#) outlines the design verification process flow for Formality. This chapter focuses on how to start Formality.

Figure 3-1 Design Verification Process Flow Overview



Invoking Formality

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 that is used in many applications in the EDA industry.

The Formality GUI is the graphical, menu-driven interface. It allows you to perform verification and provides schematic and logic cone views to help you debug failed verifications.

Starting the Shell Interface

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.

You can use the following command-line options when starting `fm_shell`:

```
-file filename
```

Invokes Formality in a shell and runs a batch script. For example,

```
% fm_shell -file my_init_script.fms
```

`-no_init`

Prevents setup files from being automatically read upon invocation. This is useful when you have a command log or other script file you want to use to reproduce a previous Formality session. For example,

```
% fm_shell -no_init -f fm_shell_command.log.copy
```

`-64bit | -32bit`

Invokes Formality using the 64-bit binary executable on platforms that support it. The default is 32 bits. Use 64 bits only when Formality runs out of memory running with the default 32-bit executable.

`-overwrite`

Overwrites existing FM_WORK, formality.log, and fm_shell_command.log files.

`-name_suffix filename_suffix`

Appends the suffix to the log files created by Formality. For example,

```
% fm_shell -name_suffix tmp
```

This command generates files named FM_WORK_tmp, formality_tmp.log, and fm_shell_command_tmp.log.

`-version`

Prints the version of Formality, then exits.

`-session session_file_name`

Specifies a previously saved Formality session.

`-gui`

Starts the Formality graphical user interface.

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, at the operating system prompt enter one of the following:

```
•% 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
```

```
•% fm_shell -gui
```

If you use the Formality GUI, a pop-up window appears listing all of 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.

Formality Shell Environment

This section includes the following topics related to using the Formality `fm_shell` commands:

- [Entering Commands](#)
- [Supplying Lists of Arguments](#)
- [Editing From the Command Line](#)
- [Listing Previously Entered Commands](#)
- [Recalling Commands](#)
- [Redirecting Output](#)
- [Using Command Aliases](#)
- [Listing Design Objects](#)

For more information about the Tcl syntax, see [Appendix A, “Appendix A - Tcl Syntax as Applied to Formality Shell Commands.”](#) If you want more information about the Tcl language, consult books about Tcl in the engineering section of your local bookstore or library.

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 refer to two different containers, named `r` and `R`:

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

Supplying Lists of Arguments

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 follows:

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

In the preceding 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 $DESIGNS/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 will fail 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 `list_key_bindings` command to display current key bindings and the current edit mode. To change the edit mode, set the `sh_line_editing_mode` variable to either the `.synopsys_fm.setup` file or directly in the shell. To disable this feature you must set the `sh_enable_line_editing` variable to false 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.

Listing Previously Entered Commands

The `history` command with no arguments lists the last *n* commands that you entered. The most recent 20 commands are listed by default.

The syntax is as follows:

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

`keep number_of_lines`

Changes the length of the history buffer to the number of lines you specify.

`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:

```
fm_shell (setup)> history
 1 alias warning_only "set message_level_mode warning"
 2 include commands.pt
 3 warnings_only
 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 }
```

Recalling Commands

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

!!

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 you are currently in when using commands. 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)>
```

Redirecting Output

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"
```

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

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 UNIX, 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 allows output redirection.

Using Command Aliases

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 refer to 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

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

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

name

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

definition

Represents the command and list of options for which you are creating an alias. If an alias is already specified, *definition* overwrites the existing definition. If no *definition* 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 syntax of the `unalias` command is as follows:

```
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
```

Listing Design Objects

At the `fm_shell` prompt, you can create lists of pins, ports, cells, nets, and references by using a variety of `fm_shell` commands that begin with `find_` (such as `find_cells` and `find_nets`). Here are some examples using the `find_` commands.

The following command lists all the input ports that start with SCAN:

```
find_ports -input container:/library/design/SCAN*
```

The following command returns a list of all pins (across hierarchy) that are connected to `net123` in the specified design:

```
find_pins -hierarchy -of_object container:/library/design/  
net123
```

The following command returns the name of the design that is referenced by `cell123`:

```
find_references -of_object container:/library/design/  
cell123
```

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

Using the Formality GUI Environment

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

- [Managing Formality Windows](#)
- [Using the Formality Prompt](#)
- [Saving the Transcript](#)
- [Copying Text From the Transcript Area](#)
- [Copying Text to the Formality Prompt](#)

Managing Formality 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. It lists up to nine Formality windows. Selecting any window in the list “activates” that window (restores the window from icon form, if necessary, and moves it to the front).

For more information about the various windows available to you, see the following sections:

- [“Working With Schematics” on page 7-29](#)
- [“Working With Logic Cones” on page 7-38](#)
- [“Working With Failing Patterns” on page 7-44](#)

Using the Formality Prompt

The Formality prompt allows you to run `fm_shell` commands without leaving the GUI.

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

1. Enter the desired command in the text area at the Formality prompt. You can use any of these methods:
 - Type the command directly.
 - Click History, 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.

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.

Copying Text From the Transcript Area

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 UNIX-style method of selecting with the left-mouse button and pasting with the middle-mouse button to transfer text into a shell window.

Copying Text to the Formality Prompt

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 UNIX-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.

General Formality Usage Options

Whether you work in `fm_shell` or the GUI, Formality provides certain facilities that can increase your productivity and the tool's overall ease of use. The following sections describe these facilities:

- [Getting Help](#)
- [Interrupting Formality](#)
- [Controlling Messages](#)
- [Working With Script Files](#)
- [Using the Command Log File](#)
- [Controlling the File Search Path](#)

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 (setup)> help find*
find_cells           # Find the specified cells
find_designs         # Find the specified designs
find_nets            # Find the specified nets
find_pins            # Find the specified pins
find_ports           # Find the specified ports
find_references      # Find the specified design
                    references
find_svf_operation   # Find the specified svf operations
```

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

```
fm_shell (setup)> current_container -help
Usage: current_container # Set or get the current (default)
container
        [containerID]      (Container ID)
fm_shell (setup)>
```

Man pages are supplied for each Formality shell command. For more information about a specific command, use the `man` command in the following form to see the man page for that command:

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

To display the man page for an environment variable, use the `printvar` command followed by the variable name. For example,

```
fm_shell (setup)> printvar verification_auto_loop_break
```

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

```
fm_shell (setup)> man cputime
```

NAME

`cputime`
Returns the CPU time used by the Formality shell.

SYNTAX

`cputime`

DESCRIPTION

Use this command to return `cputime` used by the Formality shell. The time is rounded to the nearest hundredth of a second.

RETURN VALUES

The `cputime` command returns the following:

- * 0 for failure
- * The CPU time rounded to the nearest hundredth of a second for success

EXAMPLES

This example shows the output produced by the `cputime` command:

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

Interrupting Formality

In `fm_shell`, you can interrupt Formality by pressing Control-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 Control-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 UNIX shell by pressing Control-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 Messages

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

Represents the level of severity (note, warning, or error) as described in [Table 3-1](#).

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 3-1](#) describes the different error message levels.

Table 3-1 *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`.

Setting Message 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 3-1](#). 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, do one of the following:

fm_shell	GUI
Specify: <code>set message_level_mode threshold</code>	1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.
Specify error, warning, info, or none for threshold.	2. From Setup, select the <code>message_level_mode</code> variable. 3. In the Choose a value text box, select error, warning, info, or none. 4. Choose File > Close.

Working With Script Files

You can use the source 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 source command is:

```
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 # set newstr "New"; # comment
	Add inline comments by using a semicolon to end a command, and then using a pound sign to begin the comment.	

Table 3-2 *Script File Actions (Continued)*

Task	Description	Example
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 <code>sh_continue_on_error</code> variable to true. (This is generally not recommended because the results might be invalid if an error has occurred.)	<pre>set sh_continue_on_error true</pre>
Find scripts using the <code>search_path</code> variable	Set the <code>sh_source_uses_search_path</code> variable to true. (See “Tcl Source Command” on page 3-28.)	<pre>set sh_source_uses_search_path true</pre>

Using the Command Log File

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

You can use the 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 log file for reference when you contact Synopsys. Move the log file to another file name to prevent it from being overwritten by the next `fm_shell` session.

Controlling the File Search Path

You can define the search path Formality uses for reading data by setting the Tcl `search_path` environment variable. If you do not set this variable, Formality searches for files only in the current directory.

Use a command in the following form to set the `search_path` variable:

```
set search_path "path_1 path_2 path_3 ... "
```

By surrounding the path list with double quotation marks, you expand any environment variables listed as part of a path. Be sure to separate individual paths with a space character.

The following example instructs Formality to search in the current directory, in the expanded `$TEST` directory, and in the `/proj/files` directory (in that order) when reading data:

```
fm_shell (setup)> set search_path ". $TEST/files /proj/
files"
```

Tcl Source Command

An exception to the search path control described in the previous section occurs when you use the Tcl `source` command, which ignores the Tcl `search_path` environment variable. However, you can direct Formality to use the path as defined by the `search_path` variable by setting another Tcl environment variable, `sh_source_uses_search_path`, to `true`, as follows:

```
fm_shell (setup)> set sh_source_uses_search_path true
```

Examining the File Search Path

You can examine the current file search path that Formality is using by entering the following command:

```
fm_shell (setup)> echo $search_path
```


4

Setting Basic Elements for Design Verification

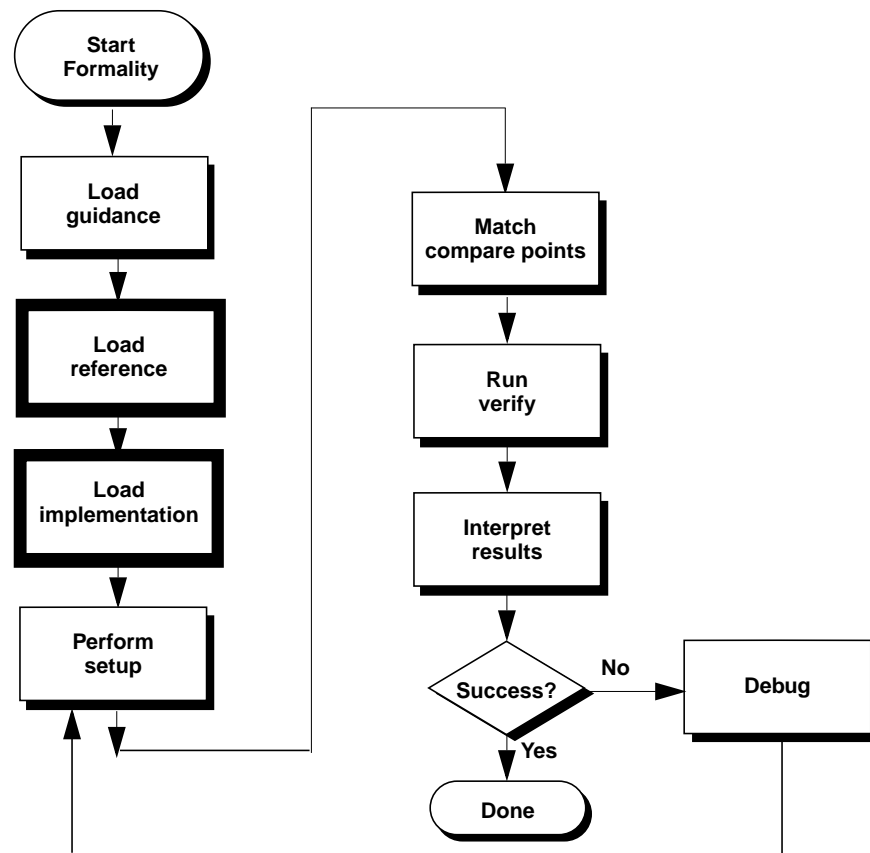
Prior to design verification you must have a few basic elements in place, such as your libraries and designs. This chapter describes the read-design flow where you specify the libraries, designs, and top-level cells.

This chapter includes the following sections:

- [Reading in Libraries and Designs](#)
- [Loading the Reference Design](#)
- [Loading the Implementation Design](#)
- [Setting Up and Managing Containers](#)

This chapter's subject matter pertains to the boxes outlined in [Figure 4-1](#).

Figure 4-1 Design Verification Process Flow Overview

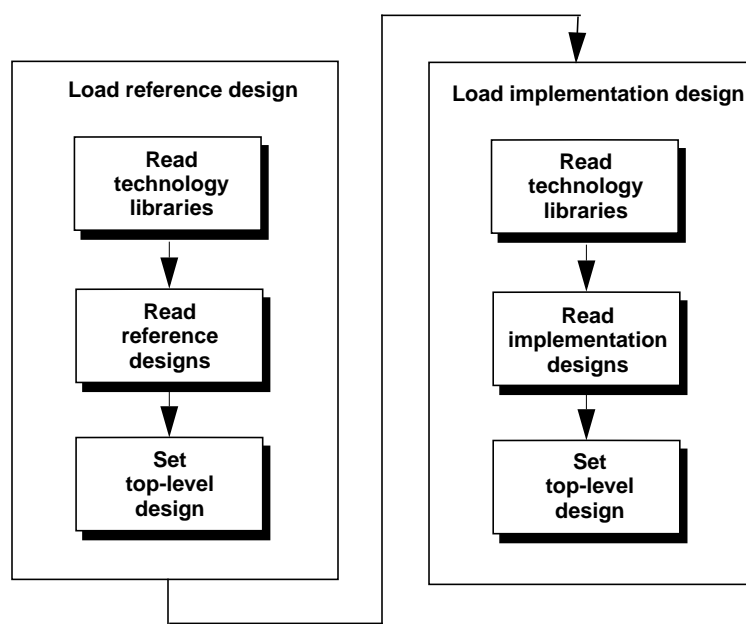


Reading in Libraries and Designs

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

Figure 4-2 outlines the Formality tool's read-design process flow.

Figure 4-2 Formality Read-Design Process Flow



To run Formality, you must read in both a reference and an implementation design and any related technology libraries. Optionally, you can pass additional setup information from Design Compiler to Formality by using the `synopsys_auto_setup` mode, and you can load automated setup files from other related tools. As shown in Figure 4-2, you first read in these automated setup files. Next, you read in only the libraries and designs that are needed for the reference, then immediately specify its top-level design. Read in only the files that you need for the implementation design, then

immediately specify its top-level design. Finally, you must set the top-level design for the reference design before proceeding to the implementation design.

To use `synopsys_auto_setup_mode`, the `synopsys_auto_setup` variable must be set to `true` before the `set_svf` command is used. If `synopsys_auto_setup_mode` is set to `true`, the following Formality variables will be set as follows:

```
hdlin_ignore_parallel_case = false
```

```
hdlin_ignore_full_case = false
```

```
verification_set_undriven_signals = 0:X
```

```
hdlin_error_on_mismatch_message = false
```

```
svf_ignore_unqualified_fsm_information = false (only  
changed when there are guide_fsm_reencoding commands in  
the automated setup file)
```

You can override any individual variable setting of the `synopsys_auto_setup` mode by manually setting any of the above variables after the `synopsys_auto_setup_mode` variable has been set.

If `synopsys_auto_setup` is set to `true` and the `set_svf` command was used to read in a automated setup file, then additional setup information (external constraints) will be passed to Formality through the automated setup file. Design Compiler began generating this additional information in the automated setup file starting with the 2007.03 release.

A summary report will be generated and included in the Formality transcript that lists all non-default variable settings and external constraint information passed via the automated setup file.

Setup-free Flow

Automated setup information is provided by Design Compiler, including clock gating and scan insertion, and this is recognized by Formality.

Formality recognizes the following commands if they appear in the automated setup file when the `synopsys_auto_setup` variable is set to `true` before the `set_svf` command is issued:

`guide_environment`

`guide_scan_input`

`guide_port_constant`

If the `verification_verify_directly_undriven_output` variable is set to `true`, Formality will issue the `set_dont_verify -directly_undriven_output` command at the beginning of the match phase. This prevents undriven outputs (typically scan outputs) in the reference design from causing a failing verification. The default value for this variable is `false`.

If clock gating is inserted by Design Compiler or Power Compiler, a `guide_environment` command is issued by Design Compiler to the automated setup file. Formality will set the value of the `verification_clock_gate_hold_mode` variable to the any value.

Formality will use the following variable values in the `guide_environment` command in the automated setup file:

`bus_dimension_separator_style`

`bus_extraction_style`

`bus_naming_style`

`bus_range_separator_style`

`hdl_naming_threshold`

`hdlin_array_instance_naming_style`

`hdlin_cell_naming_style`

`hdlin_generate_naming_style`

`hdlin_generate_separator_style`

`hdlin_sv_packages`

`hdlin_while_loop_iterations`

`template_naming_style`

`template_parameter_style`

`template_separator_style`

`hdlin_dyn_array_bnd_check` (only if `synopsys_auto_setup` is set to true)

Any of the variable values changed by the `synopsys_auto_setup` variable can be changed by the user after the `set_svf` command is completed and will retain the user-defined values.

Designs compiled using Design Compiler Z-2007.03 will contain the automated setup file commands mentioned above. To ensure setup is complete, `synopsys_auto_setup` set to `true` before issuing the `set_svf` command.

A summary report will be generated and included in the Formality transcript listing all non-default variable settings and external constraint information passed via the automated setup file.

Technology Libraries

For proper design verification, you generally need to read in technology libraries that contain descriptions of the lowest-level, or primitive, cells used in the design. Cell libraries is an alternative term for technology libraries. To learn how to verify two technology libraries, see [Chapter 8, “Technology Library Verification.”](#)

Formality often needs primitive-cell information to determine the behavior of each cell so that it can perform a gate-level comparison between the reference and implementation designs.

Formality can read the following types of technology libraries:

- Synopsys internal database (.db) files
- Synthesizable Verilog RTL files
- Synthesizable VHDL RTL files
- Verilog simulation library files

Formality reads one or more cell description files in the specified format and puts the cell definition information into a technology library. Formality uses this information when it performs a comparison between different designs.

For a purely RTL design description, reading a technology library is not necessary.

Designs

Designs are collections of design objects. A design library provides the same functionality as a technology library except that the details of a design library are visible to the user. Use a design library when you want the data to be restricted to a single design, such as RTL or gate-level netlists.

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

Milkyway database

Files in binary format that specify the designs for a Milkyway database. Milkyway only supports designs that are fully mapped and unique. The hierarchical netlist data is read in; however, synthetic constraints and physical data, such as floor plan data, are not read in.

Synopsys internal database (.db or .ddc) designs

Files compiled by Design Compiler or files that represent technology libraries. Reading a Synopsys internal database file into a container results in a design library that can consist of one or more designs.

SystemVerilog files

Files consisting of SystemVerilog modules that, when read into a container, represent a design library that can consist of one or more designs.

Verilog files

Files consisting of Verilog modules that, when read into a container, represent a design library that can consist of one or more designs.

VHDL files

Files consisting of VHDL entity and architecture pairs that, when read into a container, represent a design library that can consist of one or more designs.

The following sections describe environment variables you might need to set prior to reading in your designs.

Changing Bus Naming and Dimension Separator Styles

When you read Verilog or VHDL designs into the Formality environment, Formality uses a default bus naming style and a default bus dimension separator to support bus names. Formality uses %s[%d] as the bus naming style and a pair of square-bracket characters ([]) as the bus dimension separator.

For example,

```
BUSA[1]  
BUSA[2]  
BUSB[1][1]  
BUSC[1][2]
```

Your design might not follow this default bus naming scheme. If it does not, you can change the default bus naming scheme for both the bus naming style and the bus dimension separator by setting environment variables.

Changing the Bus Naming Style. To change the bus naming style, do one of the following:

fm_shell	GUI
Specify: <code>set bus_naming_style value</code>	<ol style="list-style-type: none">1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variables Editor dialog box appears.2. From Setup, select the <code>bus_naming_style</code> variable.3. In the “Enter a value” box, enter the text string that you want to use and press Enter.4. Choose File > Close.

For `bus_naming_style`, supply the string `%sx[%dx]`, where `x` is any character or character string. For an explanation of this variable, see the man page.

Changing the Bus Dimension Separator. To change the bus dimension separator style, do one of the following:

fm_shell	GUI
Specify: <pre>set bus_dimension_separator_style style</pre>	<ol style="list-style-type: none"> 1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variables Editor dialog box appears. 2. From Setup, select the <code>bus_dimension_separator_style</code> variable. 3. In the “Enter a value” box, enter the character string you want to use and press Enter. 4. Choose File > Close.

For *style*, supply any character string. For more information about this variable, see the man page.

Supporting DesignWare Components

DesignWare is a library of functions that you can instantiate in a netlist or infer in RTL code. Following is an example of instantiated DesignWare:

```
module mult8 ( Y, A, B );

parameter width=8;
parameter A_width=8, B=width=8;
output [width*2-1:0] Y;
input [width-1:0] A, B;

//instantiation of DesignWare multiplier
DWO2_mult #(A_width,B_width) mil ( A, B, 1'b0, Y);
endmodule
```

To read in a VHDL or Verilog design that has DesignWare components instantiated directly, do one of the following:

fm_shell	GUI
Specify: <code>set hdlin_dwroot "root_path"</code>	<ol style="list-style-type: none">1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variables Editor dialog box appears.2. From Reading Designs, select the <code>hdlin_dwroot</code> variable.3. In the "Enter a directory path" box, enter the full path to the directory containing the DesignWare tree and click Enter.4. Click File > Close.

Set the `hdlin_dwroot` variable to the location of your Design Compiler or DesignWare software installation tree. For example,

```
fm_shell (setup)> set hdlin_dwroot "/synopsys/2002.09"
```

This variable is required only for instantiated DesignWare. It applies to both Verilog and VHDL RTL code. Do not create separate libraries, such as DWARE, DW01, or DW02 for VHDL RTL code. Creating separate libraries causes linking errors.

When your design is elaborated in the linking phase, all your instantiated DesignWare components are generated automatically.

Setting Variables for VHDL and Verilog Directives

Before you read in a VHDL or Verilog design file, you might need to specify how Formality should treat VHDL or Verilog Design Compiler directives that are defined by Synopsys. Formality either ignores or does not ignore each directive to get a simulation interpretation of

the RTL. You can override the Formality tool's default behavior regarding these directives with the variables described in [Table 4-1 on page 4-13](#).

Formality directly supports the Verilog preprocessor directives ``define`, ``ifdef`, ``else`, ``endif`, ``ifndif`, and ``undefineall`. Therefore, you do not need to set a variable before reading a Verilog design containing these directives.

Although the default variable settings provide a good starting point for verification, there might be cases where overriding the default behavior toward a specific directive can help you accurately verify your design.

Note:

Overriding the default variable settings can cause Formality to overlook simulation or synthesis mismatches. The default settings minimize this risk.

[Table 4-1](#) lists the common Formality variables used to control VHDL and Verilog Design Compiler directives defined by Synopsys. If a variable is set to `true`, the corresponding directive is ignored. If the variable is set to `false`, the corresponding directive is not ignored.

Table 4-1 Variables for Design Compiler Directives

Variable	Description
<code>hdlin_ignore_full_case</code>	Used to ignore or not ignore the Verilog or VHDL <code>full_case</code> directive. For more information about this directive, see the <i>HDL Compiler for Verilog Reference Manual</i> .
<code>hdlin_ignore_parallel_case</code>	Used to ignore or not ignore the Verilog or VHDL <code>parallel_case</code> directive. For more information about this directive, see the <i>HDL Compiler for Verilog Reference Manual</i> .

Table 4-1 Variables for Design Compiler Directives (Continued)

Variable	Description
<code>hdlin_ignore_synthesis</code>	Used to ignore or not ignore the Verilog or VHDL synthesis directive. For more information about this directive, see the <i>HDL Compiler for VHDL Reference Manual</i> .
<code>hdlin_ignore_translate</code>	Used to ignore or not ignore the Verilog or VHDL translate directive. For more information about this directive, see the <i>HDL Compiler for Verilog Reference Manual</i> .

For information about changing these variables, see [“Using Environment Variables” on page A-10](#).

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` environment variable to cause Formality to create black boxes when it encounters unresolved or empty designs during linking. For information about this variable, see the man page.

Loading the Reference Design

This section describes in detail the steps required for loading the reference design, as shown in [Figure 4-2 on page 4-3](#). 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.

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 files, do one of the following:

fm_shell	GUI
Specify: read_db file_list [-libname library_name] [-merge] [-replace_black_box]	<ol style="list-style-type: none">1. Click Reference > Read DB Libraries.2. (Optional) Specify a target library other than the default WORK, as desired.3. (Optional) Select “Read as a shared library” if necessary.4. Click DB.5. Navigate to the appropriate files and click Open.6. Click Load Files.

You use the `read_db` command to read both designs and technology library information into Formality. The command automatically determines the type of data being read and puts that information into the appropriate type of Formality library, either a design library or a technology library. The `read_db` options listed in this table pertain to reading in technology library data only. For option descriptions, see the man page.

SystemVerilog, Verilog, and VHDL RTL Format

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.

When reading libraries in Formality, you use the ``celldefine` Verilog attribute to indicate that a logic description is a library cell. This step might be necessary to take advantage of the extra processing needed to build the correct logical behavior (you would use the `hdlin_library_enhanced_analysis` variable to run this extra processing). However, because ``celldefine` 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. The `hdlin_library_files` and `hdlin_library_directory` variables provide you with an easier mechanism for defining a library to Formality.

Note:

Unlike .db files, SystemVerilog, Verilog, and VHDL technology files are not shared. You must read them in for the reference design and then for the implementation design by including the `-r` and `-i` options, respectively.

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

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set hdlin_library_file <i>file</i></code>	<code>set hdlin_library_file <i>file</i></code>
<code>set hdlin_library_directory <i>directory</i></code>	<code>set hdlin_library_directory <i>directory</i></code>
<code>read_verilog</code> <code>[-r -i -con containerID]</code> <code>[-technology_library]</code> <code>[-libname <i>library_name</i>]</code> <code>[-01] [-95] <i>file_list</i></code>	<code>read_verilog</code> <code>[-r -i -con containerID]</code> <code>[-technology_library]</code> <code>[-libname <i>library_name</i>]</code> <code>[-01] [-95] <i>file_list</i></code>
or	or
<code>set hdlin_library_file <i>file</i></code>	<code>set hdlin_library_file <i>file</i></code>
<code>set hdlin_library_directory <i>directory</i></code>	<code>set hdlin_library_directory <i>directory</i></code>
<code>read_vhdl</code> <code>[-r -i -con containerID]</code> <code>[-technology_library]</code> <code>[-libname <i>library_name</i>]</code> <code>[-87 -93]</code> <code><i>file_list</i></code>	<code>read_vhdl</code> <code>[-r -i -con containerID]</code> <code>[-technology_library]</code> <code>[-libname <i>library_name</i>]</code> <code>[-87 -93]</code> <code><i>file_list</i></code>

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

Use the `-technology_library` 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, nor does it support Verilog technology library cells with mixed user-defined primitives and synthesizable constructs. When you specify the `-technology_library` option, you must also specify `-r`, `-i`, or `-c`.

Verilog Simulation Data

You generally read in Verilog simulation data by specifying the `-y` option with the `read_verilog` command when you read in designs, as discussed in [“Reading Design Libraries” on page 4-20](#). Certain users, such as ASIC vendors, however, use the `read_simulation_library` or `read_verilog -vcs -y` command to read in the Verilog simulation data. Either command allows you to explore the library in detail for validation because it loads all data into Formality and displays all warnings. These capabilities are not necessary for most users.

To read cell definition information contained in Verilog simulation library files, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>read_simulation_library [-r -i -con containerID] [-libname library_name] [-work_library libname] [-skip_unused_udps] [-write_verilog] [-verbose] [-save_binary] [-merge] [-replace_black_box] [-halt_on_error] file_list</pre>	<pre>read_simulation_library [-r -i -con containerID] [-libname library_name] [-work_library libname] [-skip_unused_udps] [-write_verilog] [-verbose] [-save_binary] [-merge] [-replace_black_box] [-halt_on_error] file_list</pre>
Or	Or
<pre>read_verilog -vcs -y</pre>	<pre>read_verilog -vcs -y</pre>

Either command allows you to specify the building of non-black-box models for cells that use user-defined primitives and Verilog primitives as the leaf objects. The library reader extracts the pertinent information from the Verilog library to determine the gate-level behavior of the design, and generates a netlist that represents the functionality of the Verilog library cells. For option descriptions, see the man pages.

The `read_simulation_library` command supports cells that contain user-defined primitives and structural constructs. It does not support cells that use synthesizable behavioral and structural constructs.

To generate the gate-level models, the library reader parses the structural Verilog modules and table-based 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 library reader might make certain assumptions about the intended gate-level functions of the user-defined primitives in the simulation model. The library 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),” you can obtain more information by entering the command `man FM-016`.

The library reader supports the following features:

- Sequential cells (each master-slave element pair is merged into a single sequential element)
- User-defined strengths (assign statements, user-defined primitive instantiations, and built-in primitives)
- 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)

Reading Design Libraries

To read the reference design into the current session, do one of the following, where the `-r` option indicates the reference design.

Note:

You might need to set some environment variables before reading in your design. Review [“Designs” on page 4-8](#).

fm_shell	GUI
<p>Specify one of the following, depending on your design type:</p> <ul style="list-style-type: none">• <code>read_db -r files</code>• <code>read_milkyway -r files</code>• <code>read_sverilog -r files</code>• <code>read_verilog -r files</code> <code>-vcs "-v lib_file"</code> <code>-vcs "-y lib_dir"</code>• <code>read_vhdl -r files</code>	<ol style="list-style-type: none">1. Click Reference > Read Design Files.2. Click the DB, SystemVerilog, Verilog, or VHDL tab.3. (Optional) For Verilog designs, click Options, select one of the options (such as -v or -y), as needed, then click OK.4. Select the applicable DB..., SystemVerilog, Verilog..., or VHDL... option to open the file navigator.5. Navigate to the appropriate files and click Open.6. Click Load 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 a `designID` argument. The `designID` argument is a path name whose elements indicate the container (by default, “r” or “i”), library, and design name.

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.

When reading in Verilog designs, you can use the `hdlin_auto_netlist` variable to specify that Formality automatically use the Verilog netlist reader. The default is to use the netlist reader. Using the Verilog netlist reader can decrease loading times. If the Verilog netlist reader is unsuccessful, Formality uses the default Verilog reader.

Use the VCS options `-v` and `-y` if you have Verilog simulation libraries or design modules you want to link to the reference or implementation designs. These options tell Formality to search in the specified library or file for unresolved module references. These options do not support Verilog technology library cells with mixed user-defined primitives and synthesizable constructs.

Reading in design libraries with one of the commands listed in the preceding table creates a design library with the default name `r:/WORK` (or `i:/WORK` for the implementation design).

Reading Milkyway and .ddc Databases

To read design netlists and technology libraries from Milkyway and .ddc databases, you use the `read_milkyway` and `read_ddc` commands, respectively. These commands allow Formality to read design information, including netlists and technology libraries, from Milkyway databases produced by Astro and from .ddc databases produced by Design Compiler in XG mode.

Milkyway Databases

To read designs from a Milkyway database, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>read_milkyway [-r -i -container <i>containerID</i>] [-libname <i>library_name</i>] [-technology_library] [-version <i>version_number</i>] -cell_name <i>mw_cell_name</i> mw_db_path</pre>	<pre>read_milkyway [-r -i -container <i>containerID</i>] [-libname <i>library_name</i>] [-technology_library] [-version <i>version_number</i>] -cell_name <i>mw_cell_name</i> mw_db_path</pre>

Formality reads in designs from a Milkyway design database (created using Physical Compiler and updated using Jupiter or Astro) using the external reader program (fmxr). The fmxr reader reads the logic view netlist description from the Milkyway database, sets the current design to the design you define, links the design, and updates by ECO the logical information from the CEL view to the design in memory. Make certain you set the search path in Formality before running the `read_milkyway` command, because the fmxr reader uses the information to locate the designs.

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

Note:

Although linking is done inside the fmxr reader to update design changes, you must run the `set_top` command in Formality to link the entire design.

.ddc Databases

To read designs from a .ddc database into a container, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>read_ddc [-r -i -container containerID] [-libname library_name] [-technology_library] file_list</pre>	<pre>read_ddc [-r -i -container containerID] [-libname library_name] [-technology_library] file_list</pre>

Formality reads in files formatted as Synopsys .ddc database designs. Formality returns a 1 if the design is successfully read; it returns 0 if the design is not successfully read into the destination container. In situations where existing designs are present in the destination container, Formality overwrites them with the designs being read. For more information, see the man page.

Setting the Top-Level Design

To set the top-level design for the reference design, do the one of the following:

fm_shell	GUI
Specify:	1. Click Reference > Set Top Design.
<pre>set_top [-vhdl_arch name] [moduleName designID -auto] [-parameter value]</pre>	2. Select the library that contains the top-level design, and then select a design.
	3. Click Set Top.

If you are elaborating VHDL and you have more than one architecture, use the `-vhdl_architecture` option. For option descriptions, see the man page.

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. Be aware of the following:

- After you start reading in a reference or implementation design, you must finish before specifying the `set_top` command. In addition, you cannot start reading in the implementation design until you have specified `set_top` for the reference design.
- The `set_top` command always applies to the design data just previously read into Formality (whether it is the implementation or reference design). You receive an error if you specify a design that you have 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 with the `set_reference_design`, `set_implementation_design`, or `verify` command. The design you specify must reside within the top-level design.

To set parameters in your top-level design, use the `-parameters` option to the `set_top` command. Use this 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 parameters or VHDL generics on the design you are setting them on. The parameter values can either be integers or specified in the format `<param_name> <hex value format> <base>'h<value>`. 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.

You can generate a report on any simulation/synthesis mismatches in your design after setting the top level of your design. Formality automatically generates an RTL report summary describing any simulation/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. For more information about reporting simulation/synthesis mismatches, see the `report_hdlin_mismatches` man page.

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

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

fm_shell	GUI
Specify: <pre>set hdlin_auto_top true</pre>	<ol style="list-style-type: none"> 1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variables Editor dialog box appears. 2. From Reading Designs, select the <code>hdlin_auto_top</code> variable. 3. Select "Set top design automatically." 4. Choose File > Close.

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 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.

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 nearly identical to that described in [“Loading the Reference Design” on page 4-15](#). See that section for more information.

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. Reading the implementation design uses the exact same procedures and commands as reading the reference design, only the switch is different. You can use the `-i` option to specify the implementation container or use the `-c 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 4-18](#). Otherwise, if you have Verilog simulation data, use the VCS `-y` option with the `read_verilog` command.

Setting Up and Managing Containers

As described in [“Containers” on page 1-29](#), 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 in Libraries and Designs” on page 4-3](#), Formality uses default containers named “r” and “i.”

You generally do not need to work directly with containers. However, Formality allows you the option if, for example,

- You want to read your reference and implementation designs into names you specify rather than the default “r” and “i”
- You require backwards compatibility with existing Formality scripts
- 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_containers` command, Formality issues a warning that the container already exists.

To create a container, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>create_container</code> [containerID]	<code>create_container</code> [containerID]

Formality uses the `containerID` string as the name of the new container. For more information about this command, see the `man` page. If using this command, you must do so before reading in your libraries and designs.

Alternatively, you can specify a container with the `-con containerID` option to the `read_db`, `read_ddc`, `read_milkyway`, `read_sverilog`, `read_verilog`, or `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. Only the `read_db` command can be used to read shared technology libraries.

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 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 do one of the following:

- Specify `set_top` for the current container.
- Specify `remove_container` to remove the container and its contents. For the default `r` and `i` containers, this command merely removes 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 `write_container` and `save_session` commands will not be executed if you have not used the `set_top` command to link the top-level design.

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

5

Preparing the Design for Verification

After reading designs into the Formality environment and linking them, you generally need to set design-specific options to help Formality perform verification. For example, if you are aware of certain areas in a design that Formality cannot verify, you might want to prevent the tool from attempting to verify those areas. Or, if you want to speed up verification, you might declare blocks in two separate designs black boxes.

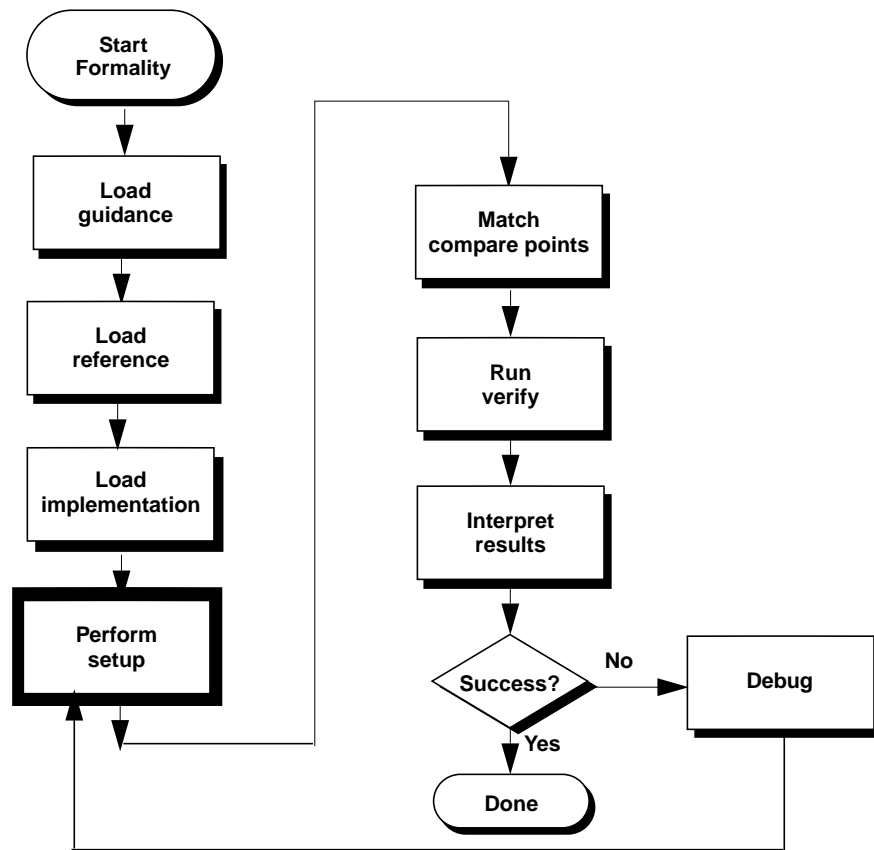
This chapter points out aspects you should consider in order to optimize your design for verification. Examples are black boxes, equivalences, external constraints, and don't care cells. The chapter includes the following sections:

- [Using Don't Care Cells](#)
- [Setting Up Designs](#)
- [Removing Information](#)

- Saving Information
- Restoring Information

This chapter's subject matter pertains to the box outlined in Figure 5-1.

Figure 5-1 Design Verification Process Flow Overview



Using Don't Care Cells

Don't care conditions are input patterns for which the function of a design or block is not defined. Prior to verification you must define don't care conditions as necessary.

There are three ways to create don't care conditions:

- Explicitly in the design. For input patterns that you do not expect to occur, you can specify the output values as X (don't care) rather than 0 or 1 in the HDL description of the design.
- Implicitly in the design. In the HDL description, you can specify the output values for certain combinations of input values and not for others. The output values for the unspecified combinations of input values are don't care.
- User-specified in Formality. In Formality, you can insert user-defined don't care values into the verification by applying external constraints to the design. For example, you can constrain an input bus so that Formality considers only "one-hot" values for that bus during verification. "One-hot" means that exactly one bit is 1 and all others are 0 on the bus. For more information, see ["Working With External Constraints" on page 5-30](#).

When Formality encounters a don't care condition, it models the logic by using an extra cell called a don't care cell. This cell has two inputs, dc (don't care) and f (function), and a single output. When the dc input is 0, the f input is passed through to the output unchanged. When the dc input is 1, the f input is ignored and the output value is X (don't care).

In the consistency verification mode (the default), the value X is the don't care parameter in the reference design and unknown in the implementation design. Therefore, the value 0, 1, or X in the implementation design is considered equivalent to an X in the reference design.

If it is important for the implementation design to have the same don't care set as the reference design, use the equality (rather than consistency) verification mode. In the equality mode, all don't care conditions must match exactly between the reference and implementation designs to pass verification. Doing this is appropriate only when both the implementation and reference designs are RTL designs, because gate-level designs do not have don't care information.

Setting Up Designs

How to set up your designs for verification is described in the following sections. Depending on your design, you might not perform all of the operations described.

- [Supporting Multibit Library Cells](#)
- [Resolving Nets With Multiple Drivers](#)
- [Eliminating Asynchronous State-Holding Loops](#)
- [Working With Black Boxes](#)
- [Working With Constants](#)
- [Working With Equivalences](#)
- [Working With External Constraints](#)

- Working With Hierarchical Designs
- Working With Combinational Design Changes
- Working With Sequential Design Changes
- Working With Reencoded Finite State Machines
- Working With Retimed Designs
- Working With Single-State Holding Elements
- Working With System Functions for Low-Power Design
- Working with Retention Registers without RTL System Functions for Low Power Design
- Working With Multiplier Architectures
- Working With Automated Setup Files
- Automated Setup File Diagnostic Messages
- Automated Setup File Conversion to Text

Supporting 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.

Resolving 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 instruct Formality to use resolution functions to resolve these types of nets.

To define net resolution, do one of the following:

fm_shell	GUI
Specify: <code>set_parameters</code> <code>[-resolution <i>function</i>]</code> <code>designID</code>	<ol style="list-style-type: none">1. Choose Setup > Design Parameters.2. Click the Reference or Implementation tab.3. Select a library, then a design.4. Click Consensus, Treat Drivers as Black Boxes, 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. For more information about this command, see the man page.

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 5-1](#) 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 5-1 Consensus Resolution for a Net With Two Drivers

	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	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 that 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 5-2](#), 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 5-2 Default Resolution Function: Part One

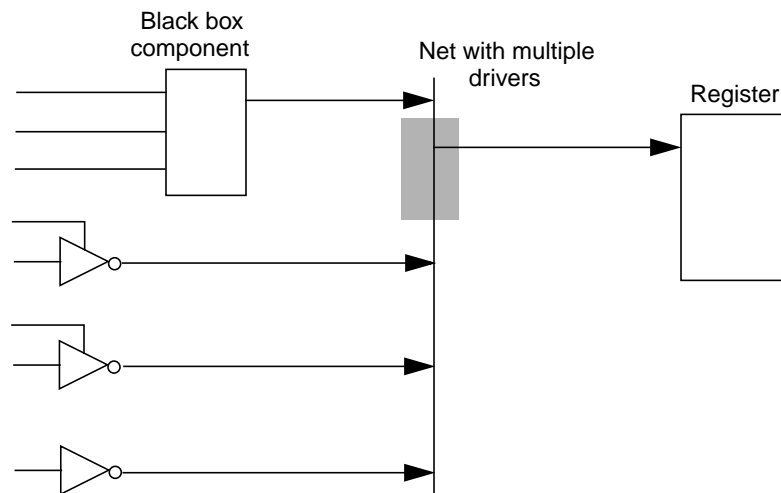
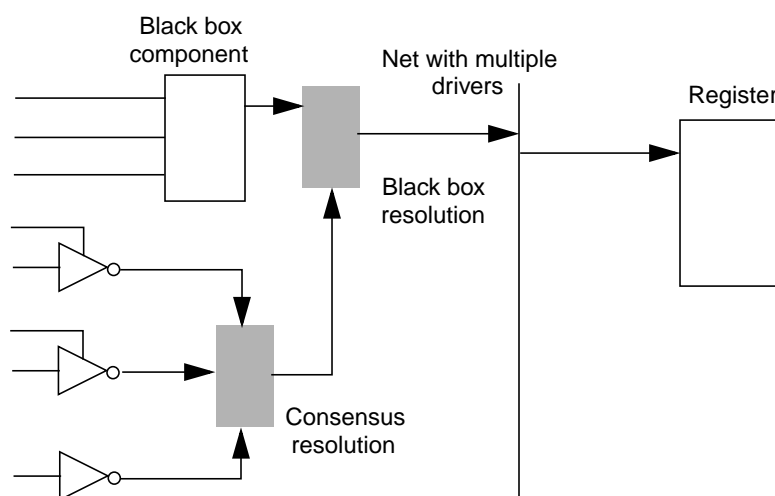


Figure 5-3 shows how Formality resolves the net in this case. The three drivers at the bottom of the circuit can be resolved by the consensus function. That function in turn drives a black box resolution function that ultimately drives the register.

Figure 5-3 Default Resolution Function: Part Two



Eliminating 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 `verification_auto_loop_break` variable to false. 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, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>report_loops [-ref] [-impl]</code> <code>[-limit N] [-unfold]</code>	<code>report_loops [-ref] [-impl]</code> <code>[-limit N] [-unfold]</code>

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 `-limit` 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, as described in the next section.

Working With Cutpoints

As previously mentioned, Formality uses black box input pins as compare points. You can also manually insert compare points at a specific net or hierarchical block pin. Inserting a compare point at a specific net or hierarchical block pin in a design has the effect of creating a new primary output and a new primary input to the downstream logic. This point is called a cutpoint because it effectively cuts the net on which it is inserted.

You can use cutpoints to create specific compare points to verify designs that contain asynchronous state-holding loops. Setting cutpoints on hierarchical block pins and internal nets allows Formality to verify these points independently and use them as a free variable for verification of downstream compare points.

When the design you want to verify has asynchronous state-holding loops, you can manually break each loop by specifying a net or hierarchical block pin of a cell in the loop, then inserting a cutpoint. Likewise, when you encounter a design that is difficult to verify, you can simplify the cones under verification by inserting cutpoints at

specific nets or hierarchical block pins, thereby creating new compare points and reducing the size of the logic cones to be verified.

You can refer to the following sources to help you determine a point in the design that is causing (or is likely to cause) a verification problem:

- Your prior knowledge of loops in the circuit design. If you properly insert cutpoints from the start, you can avoid the time spent on generating an inconclusive verification.
- The occurrence of inconclusive verification results. Following the verification run, use the `report_loop -ref` and `report_loop -impl` commands to locate the loops.
- Timing loop reports generated by other tools, such as PrimeTime and Design Compiler.

After you determine the location of a loop in the design, you can insert cutpoints anywhere in the loop without risk of harming the verification setup. Adding a cutpoint merely introduces a new compare point that is verified by Formality. Be sure to place cutpoints in exactly the same locations in the reference and implementation designs. Use the `set_user_match` command to match the corresponding cutpoints.

Creating a Cutpoint

To insert a cutpoint in a design, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set_cutpoint objectID</code> <code>[-type objectID_type]</code>	<code>set_cutpoint objectID</code> <code>[-type objectID_type]</code>

Specify the name of the net or pin along with the design object where Formality should insert the cutpoint. If the name of the specified design object is associated with more than one type of design object, you must specify the type (either pin or net), using the `-type` option. For more information about the `set_cutpoint` command, see the man page.

Reporting Information About Cutpoints

To report the cutpoints inserted with the `set_cutpoint` command, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>report_cutpoints</code>	<code>report_cutpoints</code>

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

Removing a Cutpoint

To remove cutpoints added with the `set_cutpoint` command, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>remove_cutpoint object_list</code> <code>-type -all</code>	<code>remove_cutpoint object_list</code> <code>-type -all</code>

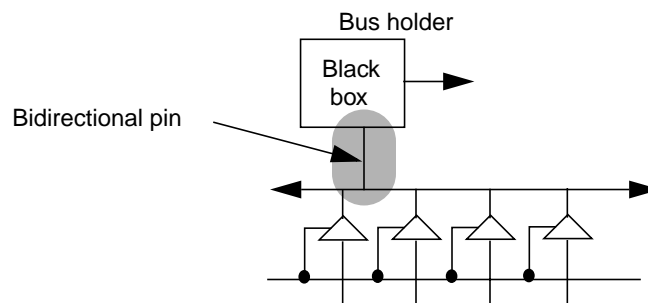
Working With Black Boxes

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 found in the reference design do not match those found in the implementation design, compare points will not match.

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 5-4](#) shows an example.

Figure 5-4 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 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` environment variable to cause Formality to create black boxes when it encounters unresolved or empty designs during linking. For more information about this variable, see the man page.

Note:

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

You can use the `verification_ignore_unmatched_implementation_blackbox_input` variable to cause Formality to allow successful verification of unmatched input pins on matched black boxes in the implementation design. For more information about this variable, see the man page.

Because of the uncertainty that black boxes introduce to verification, Formality allows you some control over 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

If you know you want an object to be a black box, specify the `hdlin_interface_only` environment variable rather than loading nothing into Formality. 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, do one of the following:

<code>fm_shell</code>	GUI
Specify: <code>set hdlin_interface_only</code> <code>"designs"</code>	<ol style="list-style-type: none"> 1. Click Reference or Implementation. 2. Click Options. 3. Click the Variables tab. 4. In the "Read interface only for these designs" box, enter list of designs. 5. Click OK.

The `hdlin_interface_only` environment variable allows 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 wildcards. 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 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 hdlin_interface_only "DRAM* SRAM*"
```

Marking a Design as a Black Box for Verification

To mark a design as a black box for verification, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set_black_box designID</code>	<code>set_black_box designID</code>

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.

The `set_black_box` command allows you to specify the designs that you want to black box. Designs you specify with the `hdl_in_interface_only` variable on unresolved references always retain their black box attribute.

Reporting Black Boxes

To report black boxes, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>report_black_boxes [design_list -r -i -con containerID] [-all] [-unresolved] [-empty] [-interface_only] [-set_black_box]</pre>	<pre>report_black_boxes [design_list -r -i -con containerID] [-all] [-unresolved] [-empty] [-interface_only] [-set_black_box]</pre>

By default, this command lists the black boxes for both the reference and implementation designs. Formality issues an error 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 by the `set_top` command. Once linked, the design might show up as a black box if it is not really 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 `-unresolved`, `-empty`, `-interface_only`, and `-set_black_box` options. The default `-all` 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, do one of the following:

fm_shell	GUI
Specify: <code>set verification_blackbox_match_mode identity</code>	<ol style="list-style-type: none"> 1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. 2. From Matching, select the <code>verification_blackbox_match_mode</code> variable. 3. Select Identity. 4. Choose File > Close.

The `verification_blackbox_match_mode` variable causes Formality to perform an identity check between each set of two comparable black boxes, ensuring that both black boxes have the same library and design names. For more information about this variable, see the man page.

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 circumvent 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, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set_direction</code> <i>objectID direction</i>	<code>set_direction</code> <i>objectID direction</i>

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. For more information, see the man page.

Working With 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 enables you to generate reports on them. You can specify how Formality propagates constants through different levels of the design hierarchy.

You can define constant states on ports or nets within a design to restrict the scope of a verification. This technique is useful when you have designs that contain scan logic and you want to disable that logic during verification. For example, you can define a constant to disable a scan enable line and thus verify the design in mission mode.

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, do one of the following:

fm_shell	GUI
Specify: <code>set_constant [-type type] instance_path constant_value</code>	<ol style="list-style-type: none">1. Choose Setup > Constants.2. Click Add, then click the Reference or Implementation tab.3. In the left pane, navigate through the tree view to the instance and select it.4. In the right pane, select the object.5. Click 0 or 1.6. Click OK.

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 `-type` 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 if you want 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. For more information about the `set_constant` command, see the man page.

Removing User-Defined Constants

To remove a user-defined constant, do one of the following:

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

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

Listing User-Defined Constants

To list user-defined constants, do one of the following:

fm_shell	GUI
Specify:	Choose Setup > Constants.
<code>report_constants</code> <code>[instance_path ...]</code>	

If you omit `instance_path`, Formality returns a list of all user-defined constants. You can specify an object ID or instance-based path name for `instance_path`. 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.

Working With Equivalences

You might want to declare two design objects as equivalent. You can use the `set_user_match` command to match one object in the reference design to many objects in the implementation design. For example, suppose the reference design has a single clock port, and the implementation design has several clock ports. You would use `set_user_match` to match all of the implementation ports to the reference port.

To make an equivalence declaration, use the `set_equivalence` command. The `set_equivalence` command declares a pair of nets or ports to be equivalent in the reference and implementation designs. You can also declare a pair of ports, pins, or nets in the same design as equivalent; therefore, when two pins are set as equivalent, the second pin uses the value of the first pin.

In the case of a net, Formality disconnects the net from the driver in the implementation design and then reconnects that net to the corresponding driver in the reference design. The net in the implementation design is now driven by the equivalent net in the reference design. Most importantly, this means that any logic driving the net in the implementation design is not verified. The same is true for two input ports declared to be equivalent.

Defining an Equivalence

To make two design objects equivalent, do one of the following:

fm_shell	GUI
Specify: set_equivalence [-type <i>type</i>] [-propagate] [-inverted] <i>objectID_1</i> <i>objectID_2</i>	<ol style="list-style-type: none">1. Choose Setup > Equivalences.2. Click Add, and then select Ports or Nets for the object type.3. In the Reference section, select a library, design, and object.4. In the Implementation section, select a library, design, and object.5. (Optional) Select the Propagate or Invert option.6. Click OK.

If more than one design object shares the same name with either specified design object, use the `-type` option and specify port or net (whichever applies) for the object type. For information about this command, see the man page.

Removing User-Defined Equivalences

To remove a single user-defined equivalence, do one of the following:

fm_shell	GUI
Specify: remove_equivalence [-all] [-type <i>type</i>] <i>objectID_1</i> <i>objectID_2</i>	<ol style="list-style-type: none">1. Choose Setup > Equivalences.2. Select a user-defined equivalence from the list.3. Click Remove.

If more than one design object shares the same name as a specified item, use the `-type` option and specify port or net (whichever applies) for the type. For more information about this command, see the man page.

Listing User-Defined Equivalences

To list user-defined equivalences, do one of the following:

<code>fm_shell</code>	GUI
Specify: <code>report_equivalences</code>	Choose Setup > Equivalences.

Formality produces a list of user-defined equivalences for designs, ports, and nets. For more information about this command, see the man page.

Using Verilog 2001 Constructs. You can perform equivalency checking on designs that incorporate commonly used Verilog 2001 constructs. These constructs include, but are not limited to,

- Indexed vector part
- Signed arithmetic extensions
- Power operator
- Combinational logic sensitivity token
- Automatic width extensions
- Combined port and data type declarations
- American National Standards Institute (ANSI) style of input and output declarations

The following sections describe and provide examples for each of the supported Verilog 2001 constructs.

Indexed Vector Part

Indexed vector part selects are supported. Formality allows indexed part select, a base expression, and an offset direction. For instance, the following is allowed:

```
[base_expr +: width_expr]
[base_expr -: width_expr]
wire [7:0] byteN = word[byte_num*8 +: 8]
```

Signed Arithmetic Extensions

Formality accepts signed arithmetic extensions: data type, system function (\$signed, \$unsigned), and arithmetic shift operators (>>> and <<<). Data type support includes register and net data types and ports and functions to be declared as signed types. System function support includes \$signed and \$unsigned that are used to convert signed values to unsigned and vice versa. Arithmetic shift operator support maintains the sign of a value by filling in the signed-bit value as it shifts.

This example shows a signed data type:

```
reg signed [63a:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

This example shows a system function:

```
reg [63:0] a; //unsigned data type
always @(a) begin
    result1 = a + 2; //unsigned
    result2 = $signed(a) +2; //signed
end
```

This example shows arithmetic shift operators:

```
D >> 3 //logical shift yields 8'b00010100
D >>> 3 //arithmetic shift yields 8;b11110100
```

Power Operator

Formality accepts a power operator that is similar to the C pow () function. This is useful when you need the power operator to calculate values, such as a2. An example of a power operator is

```
always @(posedge clock)
result = base ** exponent;
```

Combinational Logic Sensitivity Token

Formality now accepts the combinational logic sensitivity token. This token (@*) represents a logic sensitivity list, indicating that statement groups should automatically be sensitive to changes on any values read in that group. An example of this is

```
always @* //combinational logic
if (sel)
    y = a;
else
    y = b;
```

Automatic Width Extensions

Formality can handle automatic width extensions beyond 32 bits. Therefore, an unsized value of Z or X automatically expands to fill the full width of the vector on the left side of the argument. An example showing how Formality handles automatic width extensions is

```
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = `bz; //fills with `hzzzzzzzzzzzzzzzzzzzz
```

Combined Port and Data Type Declarations

Formality accepts declarations that combine the direction of the port and data type of that signal into one statement. The following shows an example of this:

```
module mux8 (y, a, b, en);  
  output reg [7:0] y;  
  input wire [7:0] a, b;  
  input wire en;
```

ANSI C-Style Module Input and Output Declarations

Formality supports ANSI C-style port declarations for modules. An example of this is

```
module mux8 (output reg [7:0] y,  
  output reg [7:0] a,  
  input wire [7:0] b,  
  input wire en );
```

Working With 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 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, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<pre>set_constraint type_name [-name constraint_name [-map map_list1 map_list2] constraint_type control_point_list [designID]</pre>	<pre>set_constraint type_name [-name constraint_name [-map map_list1 map_list2] constraint_type 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 control points (primary inputs, registers, and black box outputs) to which the constraint applies. Use `designID` to specify a particular design; the default is the current design. For more information about the `set_constraint` command, see the man page.

Creating a Constraint Type

To create a user-defined (arbitrary) constraint type and establish the mapping between the ports of a design that define the constraint and control points in the constrained design, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>create_constraint_type</code> <code>type_name</code> <code>[designID]</code>	<code>create_constraint_type</code> <code>type_name</code> <code>[designID]</code>

For `type_name`, specify the type of constraint. For `designID`, specify a particular design; otherwise, the default is the current design. For more information about this command, see the man page.

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 [“Using Don't Care Cells” on page 5-3](#).

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, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>remove_constraint</code> <code>constraint_name</code>	<code>remove_constraint</code> <code>constraint_name</code>

For *constraint_name*, specify the name of the constraint to remove. For more information about this command, see the man page.

Removing a Constraint Type

To remove external constraint types, do one of the following:

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

For *type_name*, specify the type of user-defined constraint to remove. For more information about this command, see the man page.

Reporting Constraint Information

To report information about the constraints set in your design, do one of the following:

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

For *constraint_name*, specify the name of the constraint for which you want to obtain a report. For more information about this command, see the man page.

Reporting Information About Constraint Types

To report information about constraint types set in your design, do one of the following:

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

For more information about `report_constraint_type` command, see the man page.

Working With Hierarchical Designs

You can control the following features of hierarchical design verification: the separator character used to create flattened path names, the operating mode for propagating constants throughout

hierarchical levels, and whether or not Formality disregards hierarchical boundaries. The following sections describe these features.

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, do one of the following:

fm_shell	GUI
<p>Specify:</p> <pre>set name_match_flattened_hierarchy_separator_style character</pre>	<ol style="list-style-type: none">1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.2. From Matching, select the <code>name_match_flattened_hierarchy_separator_style</code> 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 allows Formality to understand where the hierarchical boundaries are. For more information about this variable, see the man page.

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 [“Working With Constants” on page 5-22](#).

In some cases, you might not want to propagate all constants during hierarchical verification. To determine how Formality propagates constants, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set verification_constant_prop_mode mode</code>	<code>set verification_constant_prop_mode mode</code>

You can use the `verification_constant_prop_mode` variable to specify where Formality is to start propagation during verification. In `auto` mode, the default, Formality traverses up the reference and implementation hierarchy in lock step to automatically identify the top design from which to propagate constants. Therefore, correspondence between the hierarchy of the two designs affects this mode. Specify `top` to tell Formality to propagate from the design you set as top with the `set_top` command. Specify `target` to instruct Formality to propagate constants from the currently set reference and implementation designs.

Note:

Set `verification_constant_prop_mode` to `top` or `target` only if your reference and implementation designs do not have matching hierarchy. Setting the mode to `auto` 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.

Working With Combinational Design Changes

This section describes how to prepare designs that have undergone a combinational transformation. Special attention is required if your design contains combinational design changes, such as

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

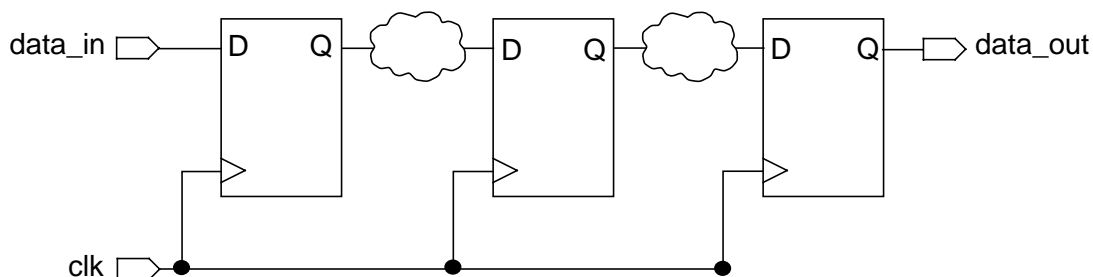
Your design can also include sequential transformations. For more information, see [“Working With Sequential Design Changes”](#) on page 5-42.

Disabling Scan Logic

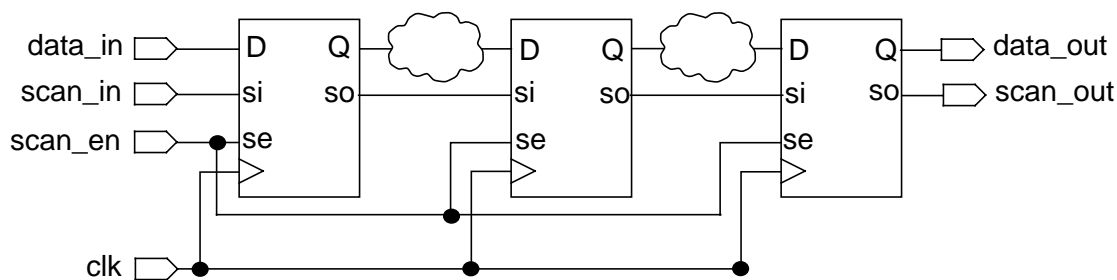
Internal scan insertion is a technique used to make it easier to set and observe the state of registers internal to a design. During scan insertion, scan flops replace flip-flops. The scan flops are then connected into a long shift register. The additional logic added during scan insertion means that the combinational function has changed, as shown in [Figure 5-5](#).

Figure 5-5 Internal Scan Insertion

Pre-Scan



Post-Scan

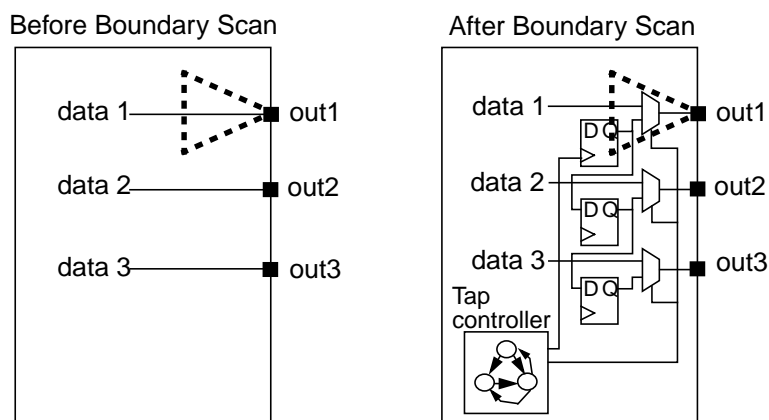


After determining which pins disable the scan circuitry, disable the inserted scan logic by specifying 0 for the `set_constant` command. For more information, see the procedure in [“Defining Constants”](#) on page 5-23.

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 5-6](#). Boundary scan is also referred to as the IEEE 1149.1 Std. specification.

Figure 5-6 *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,

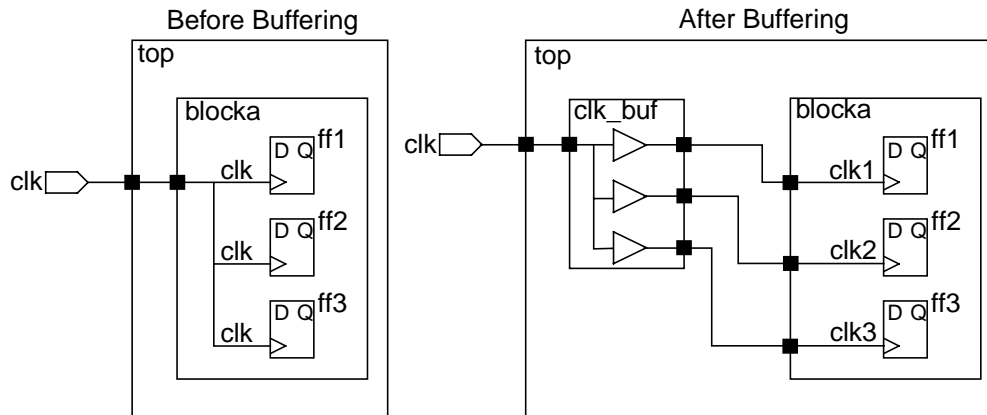
```
fm_shell (setup)> set_constant gates:/WORK/TSRTS 0
fm_shell (setup)> set_constant gates:/WORK/alu/somenet 0
```

Specify 0 for the `set_constant` command, as described in the procedure in [“Defining Constants” on page 5-23](#).

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 5-7](#).

Figure 5-7 Clock Tree Buffer



Without setup intervention, verification of blocka fails. As shown in the figure, the clock pin of ff3 is clk in the prebuffer design, while in the postbuffer design the clock pin of ff3 is clk3. The logic cones for ff3 are different, resulting in a failing point.

To counteract clock tree buffering, you must use the `set_user_match` command to specify that the buffered clock pins are equivalent. With `set_user_match` 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 `-inverted` option to the `set_user_match` 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.

Working With Sequential Design Changes

Like the combinational design changes described in [“Working With Combinational Design Changes” on page 5-38](#), sequential design changes also require setup prior to verification. Sequential design changes include:

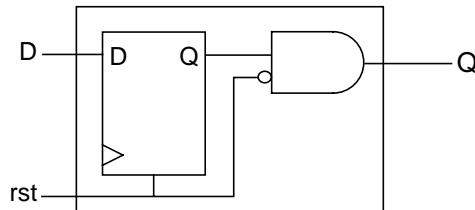
- Adding asynchronous bypass circuitry to registers
- Clock gating
- Pushing inversions across registers

FSM reencoding and module retiming are also considered sequential design changes. For more information, see [“Working With Reencoded Finite State Machines”](#) on page 5-53 and [“Working With Retimed Designs”](#) on page 5-59.

Managing Asynchronous Bypass Logic

A sequential cell where some of the asynchronous inputs have combinational paths to the outputs (bypassing the SEQGEN) is said to have an asynchronous bypass, as shown in [Figure 5-8](#).

Figure 5-8 Asynchronous Bypass Logic



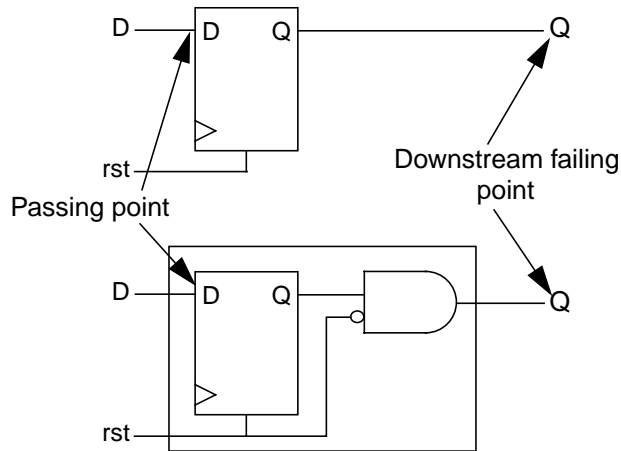
Asynchronous bypass logic can result from

- 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 SEQGEN's capability to model asynchronous behavior in order to avoid creating explicit bypass paths around a sequential element.

Asynchronous bypass logic results in a failing point, as shown in [Figure 5-9](#).

Figure 5-9 Asynchronous Bypass Failing Point



To prevent an aborted verification due to the downstream failing point, do one of the following:

fm_shell	GUI
Specify:	1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option.
set	The Formality Tcl Variable Editor dialog box appears.
verification_asynch_bypass true	2. From Verification, select the verification_asynch_bypass variable.
	3. 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:

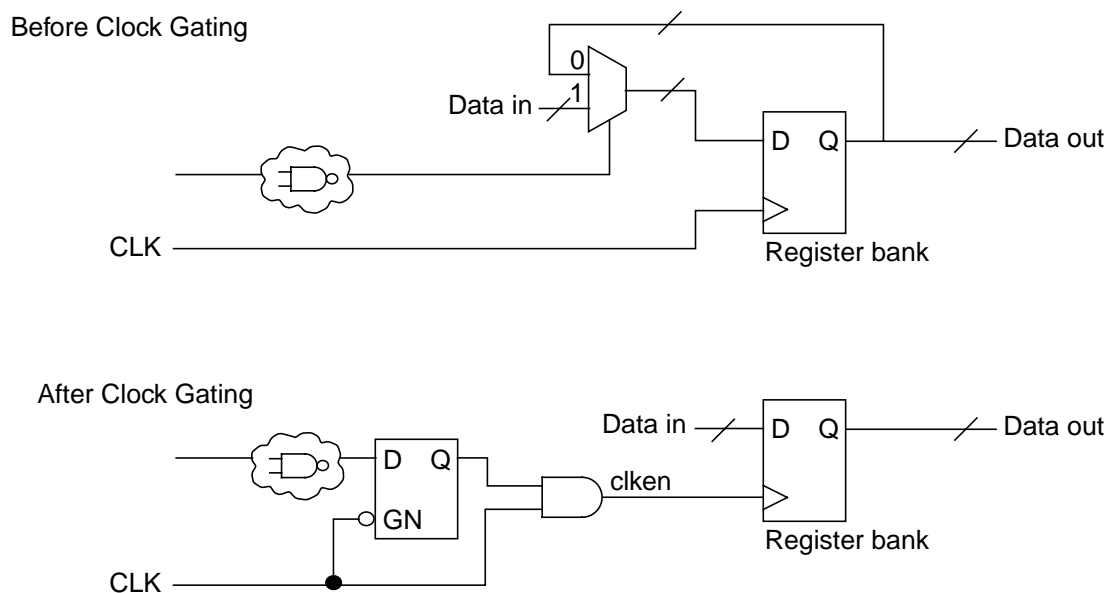
- Longer verification runtimes
- Introduction of loops into the design
- Aborted verification due to design complexity

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.

Setting Clock Gating

Clock gating can be used to implement load enable signals in synchronous registers. It results in more power-efficient circuits than multiplexer-based solutions. In its simplest form, clock gating is the addition of logic in a register's clock path that disables the clock when the register output is not changing, as shown in [Figure 5-10](#). You use clock gating to save power by not clocking register cells unnecessarily.

Figure 5-10 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 can also verify clock gating inserted by other tools or manually. In general, verification of a design with no gating against a design with inserted gating can result in a failure because of extra logic (latches) 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 instruct Formality to account for clock-gating logic, do one of the following:

fm_shell	GUI
Specify: set verification_clock_gate_hold_mode [none low high any collapse_all_cg_cells]	<ol style="list-style-type: none"> 1. Choose Edit > Formality Tcl Variables or the Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears. 2. From Verification, select the verification_clock_gate_hold_mode variable. 3. From the “Choose a value” drop down, select the desired level from the menu. 4. Choose File > Close.

The `verification_clock_gate_hold_mode` variable can have the following values:

- `none` (off) is the default.
- `low` allows clock gating that holds the clock low when inactive.
- `high` allows clock gating that holds the clock high when inactive.
- `any` allows both high and low styles of clock gating to be considered within the same design.

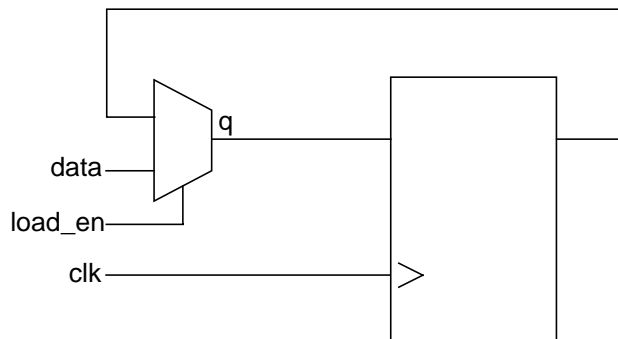
- `collapse_all_cg_cells` has the same effect as the `any` value, in addition, if a clock gating cell is in the fanin of a register and in the fanin of a primary output port or black box input pin it will be treated as a clock gating cell in all of those logic cones.

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

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

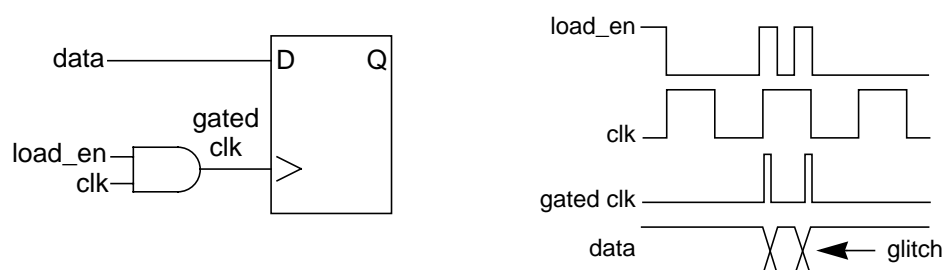
Combinational Gate Clocking. Assume the reference design in [Figure 5-11](#).

Figure 5-11 Reference Design



The typical combinational clock-gating circuitry is presented in [Figure 5-12](#). The gate has two inputs, `en` and `clk`, the output of which feeds a register clock. To the right, you see the corresponding waveforms.

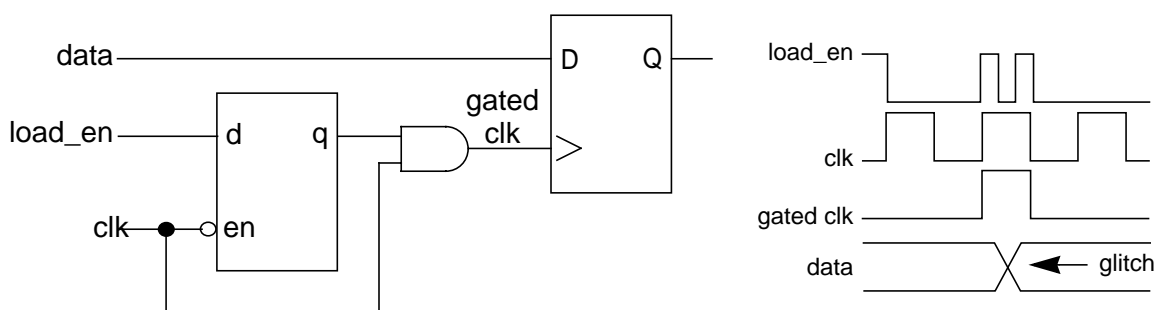
Figure 5-12 Combinational Clock Gating Using AND Gate



You see that if glitches occur on the signal `load_en`, invalid data can be loaded into the register. Hence, this circuit is functionally nonequivalent to that in [Figure 5-11](#). In default mode, Formality considers this glitch a possible input pattern and produces a failing point. Formality disregards such a nonequivalence if you set the `verification_clock_gate_hold_mode` variable to `low`.

Latch-Based Clock Gating. The typical latch-based clock-gating circuitry, such as that used by Power Compiler, is presented in [Figure 5-13](#). 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 5-13 Latched-Based Clock Gating Using AND Gate



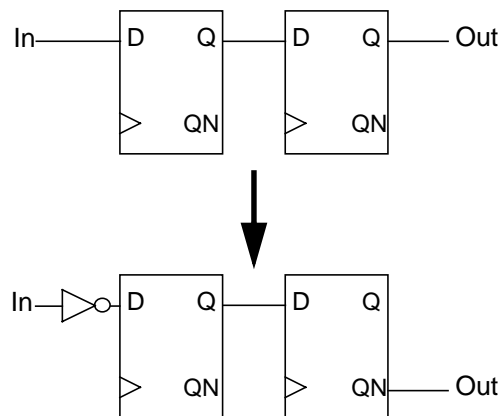
During verification, when the `verification_clock_gate_hold_mode` variable is set, Formality recognizes clock-gating latches and takes into account their role in the design under verification.

You can see that whenever `load_en` goes low, gated `clk` also goes low. Data coming out of the register is transformed at the same instant and continues to remain there until `load_en` goes up again. If you set the `verification_clock_gate_hold_mode` variable to `low`, Formality determines that this setup is the same as that of a design that has no clock gating ([Figure 5-11](#)).

Enabling an Inversion Push

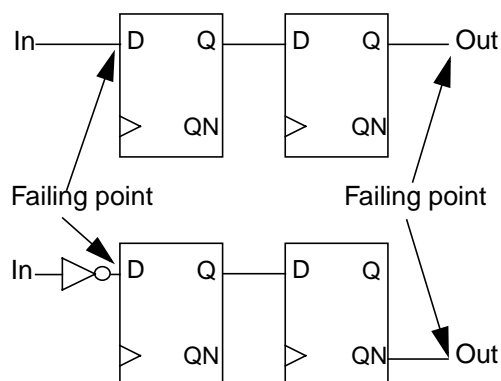
Inversion pushing means moving an inversion across register boundaries, as shown in [Figure 5-14](#).

Figure 5-14 Inversion Push



Inversion pushing causes two failing points, as shown in [Figure 5-15](#).

Figure 5-15 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 pushing specifies that a specific register had an inversion pushed across it. Formality must push an inversion across the register. This is useful when you know which register had an inverter pushed across it. This method can be applied to library cells. You apply instance-based inversion pushing before verification begins. Next state and Q/QN pins are inverted.

To remedy the resulting failing points, do one of the following:

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,

```
fm_shell (setup)> set_inv_push ref:/WORK/alu/z_reg
```

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 `set_inv_push` and `set_user_match` applied to the same point are resolved by using the polarity specified by `set_user_match`.

For more information about the `set_inv_push` and `set_user_match` commands, see their 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 by using `set_user_match -inverted`.

Environmental inversion pushing allows Formality to match all state points automatically with the correct polarity during matching. 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. This is done in the following way:

fm_shell	GUI
Specify: <pre>set verification_inversion_push true</pre>	<ol style="list-style-type: none">1. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.2. From Verification, select the <code>verification_inversion_push</code> 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.

Working With Reencoded Finite State Machines

The architecture for a finite state machine (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 FSMs, see the Design Compiler documentation.

Before verifying a reencoded 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 your intervention, 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 automated setup 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 at once, 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.

Using the Automated Setup File for FSM Reencoding

The automated setup file generated by Design Compiler contains FSM state vector encoding. This encoding is in the form of `guide_fsm_reencoding` commands. Use the following variable to tell Formality to use the FSM guidance in the Design Compiler automated setup file:

```
set svf_ignore_unqualified_fsm_information false
```

Set this variable before reading the automated setup file. For more information, see [“Working With Automated Setup Files” on page 5-72](#). You can also manually perform the `guide_fsm_reencoding` commands. For more information, see the man page.

Reading a User-Supplied FSM State File

To simultaneously name the FSM state vector flip-flops and provide state names with their encoding, do one of the following:

fm_shell	GUI
Specify: <code>read_fsm_states filename</code> <code>[designID]</code>	<ol style="list-style-type: none">1. Click the View Reference Hierarchy or View Implementation Hierarchy toolbar option.2. Choose File > Read FSM States.3. Navigate to and select the FSM state file.4. Click OK.

This is the recommended method when your FSM has many states. If your FSM has only a few states, consider the method described in the next 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.

Defining FSM States Individually

To first name an FSM state vector flip-flop and then define the state name and its respective encoding, do one of the following:

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:
<pre>set_fsm_encoding encoding_list [designID]</pre>	<pre>set_fsm_encoding encoding_list [designID]</pre>

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. For more information about the `set_fsm_state_vector` and `set_fsm_encoding` commands, see the man pages.

Multiple Reencoded FSMs in a Single Module

Formality supports multiple reencoded FSMs in a single module. FSM reencoding can occur during synthesis, with the result that the reference and implementation designs have different state registers due to different state-encoded machines. Formality supports these

reencoded FSMs if you provide both the FSM state vector and the state encoding either by using the `-name` option with the `set_fsm_state_vector` and `set_fsm_encoding` commands, or by using the `read_fsm` 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
```

Formality uses this information to verify FSM reencoded designs as follows: First it modifies the reference design by replacing the original state registers with the new state registers. Then it synthesizes the logic around the new state registers to keep the new reference design functionally equivalent to its original. Finally, Formality can verify the FSM reencoded 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, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>report_fsm [designID]</code>	<code>report_fsm [design_ID]</code>

Formality produces a list of FSM state vector flip-flops and their encoding. For more information about this command, see the man page.

Working With FSMs Reencoded in Design Compiler

If you are verifying a design with an FSM that has been reencoded in Design Compiler, you need to supply the state register mapping and state encoding to Formality first, before matching. If FSMs are present but the encoding has not been changed, no setup information is required.

Several methods are available for addressing FSM setup in Formality if you used Design Compiler to do the reencoding. These methods are listed in order of preference.

The first, and preferred, method is to write out an automated setup file (.svf) from Design Compiler, then read the file back into Formality.

The second method is to 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.

The third method is to 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 reencoded your design, or if the reencoding was done by a tool other than Design Compiler, you can use the following two commands in Formality to specify the state encoding and register state mapping:

```
set_fsm_encoding  
set_fsm_state_vector
```


You must use these two commands for both the reference and implementation designs.

Working With Retimed Designs

Retiming a design moves registers across combinational logic in an effort 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 become unmatchable. In this case, setup is required to prepare Formality to match and verify the design. If the design has been retimed using the `optimize_registers` command in Design Compiler, you can use the automated setup file in Formality. 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

When using the `optimize_registers` command in Design Compiler, a set of retiming guidance commands is written into the automated setup file. If you do not specify an automated setup file with the `set_svf` command, a default file called `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, 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, do the following:

1. From Design Compiler, use the `set_svf` command to indicate the automated setup 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 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 multistep 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, do the following:

1. In Formality, enable the following variable:

```
fm_shell (setup)> set svf_retiming true
```

The `svf_retiming` variable controls whether Formality processes all retiming guidance commands located in the user-specified automated setup file. A value of `true` indicates that retiming guidance commands are accepted. A value of `false` indicates that related retiming guidance commands are ignored. The `svf_retiming` variable affects only the retiming guidance commands.

2. Use the `set_svf` command to tell Formality the location of the automated setup file that contains the retiming guidance commands.

```
fm_shell (setup)> set_name.svf
```

3. Read the design data into Formality.

Read the original design as the reference design and the retimed netlist as the implementation design.

4. Apply any additional setup requirements for the designs.
5. Perform a verification of the design.

Retiming Using Other Tools

For designs that have been retimed outside of Design Compiler or for pipelined DesignWare parts, specify that the design has been retimed by doing one of the following:

fm_shell	GUI
Specify: <code>set_parameters -retimed designID</code>	<ol style="list-style-type: none">1. Choose Setup > Design Parameters.2. Click the Reference or Implementation tab.3. Select a library and a design.4. Select the “Design has been retimed” box.

Note:

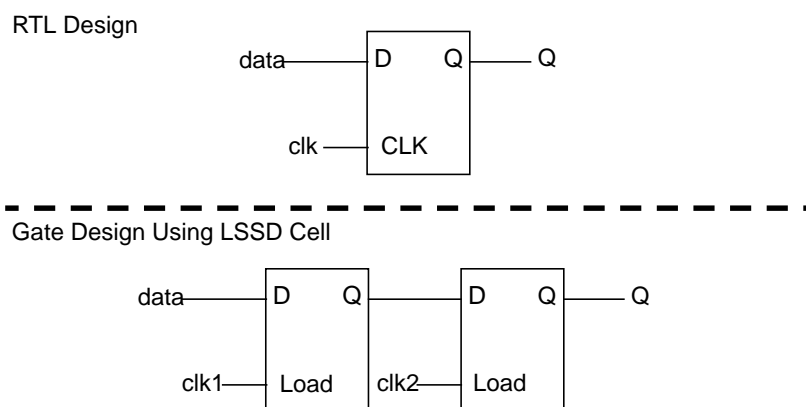
The pipelined DesignWare parts, DW02_mult_n_stage and DW_pipe_mult, are detected and handled automatically by Formality. Therefore, you do not need to set the retimed parameter for these parts.

Working With 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 5-16](#). The RTL design contains a SEQGEN that results in one compare point. The dotted line separates the reference design from the implementation design.

Figure 5-16 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.

Working With System Functions for Low-Power Design

Formality supports the Synopsys standard low power functions used for simulating, creating and verifying your low power and multi-voltage designs. Formality can read an RTL design that contains the low power constructs described below and verify it against a gate level netlist generated from Design Compiler and IC Compiler.

There are three constructs supported in both Verilog and VHDL. These are referred to by the Verilog system function call names: `$isolate()`, `$power()`, and `$retain()`. There are equivalent VHDL function calls in the `snps_ext.power` package provided with Formality (and Design Compiler). If you are using these constructs in your design, Formality can be used to verify these designs.

Both `$power()` and `$retain()` specify hierarchical blocks and control signals for those blocks. Formality will add ports to the hierarchical blocks in the design to propagate the required control signals into those blocks. The function `$isolate()` is used add different types of isolation cells between nets in the RTL. When the isolation function appears in the RTL, Formality handles it automatically by inserting the appropriate isolation cells into the design.

For more specific information on using these functions, refer to the Design Compiler and Power Compiler documentation.

There are two ways designs synthesized using these constructs can be verified:

- 1) Verify a netlist from Design Compiler.

Formality will verify the power up functionality of the netlist. In the blocks controlled by `$power`, Formality will not allow a failure at any compare point in a block when that block is powered down in the RTL. As the netlist out of Design Compiler does not have power/ground connectivity, the gates will never power down. Thus, Formality is only verifying the power up behavior in this case.

- 2) Verify a netlist from IC Compiler.

You must use IC Compiler to create a netlist that has the power and ground nets connected to each cell. Formality can verify the power up functionality (as in 1. above). In addition, if the gates power down unexpectedly, when compared with the RTL, there will be failing points.

If \$retain is used in the RTL, Formality will substitute (in the specified blocks) retention registers for the normal registers inferred by the RTL. Formality will use the control signals specified in the \$retain function to connect to the inserted retention registers to the save and restore functionality. The netlist to be verified must have retention registers (consistent with clock-free RTL model) in the blocks specified in the RTL or the verification will have failing points.

To support the verification of designs from IC Compiler using netlists that contain power and ground connections to each cell, additional steps may be required to read and link the netlist. Formality will automatically link the cells in the netlist to a library that does not have power and ground pins specified in the technology library.

If there are two pins on the instances of cells with names that match the value of the Formality variables:

```
mw_logic0_net= "VSS" (for ground, which is set to logic 0)
```

```
mw_logic1_net= "VDD" (for power, which is set to logic 1)
```

For example, a Verilog netlist instance with power/ground pins might look like the following:

```
and2p1 U1(.A(state_3_0) , .B(state_4_0) , .Z(dc  
42) , .VDD(VDD) , .VSS(VSS))
```

Formality can automatically link this instance to a cell that is declared without the power and ground pins VDD and VSS.

Some cells in the technology library may have multiple power pins - retention cells will typically have primary and secondary power and ground pins; level shifters might also have multiple power and ground pins. Formality cannot automatically link those cells to a library without additional power and ground pin information. For these cases, when reading the technology .db file containing these cells with multiple power pins, use the `read_db` command with the `-pg_pin` switch.

```
read_db -pg_pin mytechlib.db
```

Formality will then look for `pg_pin` attribute information that is stored in the .db and generate the appropriate power and ground pins when reading the technology library. This should allow you to read and link netlists containing power and ground connections into Formality for verification.

Working with Retention Registers without RTL System Functions for Low Power Design

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 mode. 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. Set the `enable_power_gating` variable to true to enable Formality to check retention registers. The default is false.

You can define the power of the gating style by using the `set_power_gating_style` command. To apply different retention registers to different HDL blocks, use the command

```
set_power_gating_style  
-hdl_blocks block_label -type type_name. The  
-hdl_blocks option is required for this command. The value of  
block_label should match the Verilog name always blocks or VHDL  
processes. The type_name argument should name the  
power_gating_cell attribute value.
```

To have Formality issue a retention register check report, use the `report_power_gating` command. This report summarizes matching register pairs without power-gating attributes, with compatible power-gating attributes, and with incompatible power-gating attributes.

Working With 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`)

For more information about the mechanism for transferring details of `csmult` and `mcarch` multipliers, see [“Working With Automated Setup Files” on page 5-72](#).

Setting the Multiplier Architecture

You can set the multiplier architecture 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, do one of the following:

<code>fm_shell</code>	GUI
Specify:	At the Formality prompt, specify:
<code>set hdlin_multiplier_architecture csa</code>	<code>set hdlin_multiplier_architecture csa</code>
<code>set enable_multiplier_generation true</code>	<code>set enable_multiplier_generation true</code>
<code>read_verilog myfile.v</code>	<code>read_verilog myfile.v</code>

The default value for `hdlin_multiplier_architecture` is `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 `enable_multiplier_generation` variable and the `set_architecture` command with the specific cell ObjectID and specific architecture to set the desired multiplier architecture.

To instruct Formality to use a specific multiplier architecture for a specific ObjectID, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set_enable_multiplier_generation true</code>	<code>set_enable_multiplier_generation true</code>
<code>set_architecture ObjectID [csa nbw wall]</code>	<code>set_architecture ObjectID [csa nbw wall]</code>

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:

Verilog

```
// formality multiplier nbw
z <= a*b;
```

VHDL

```
-- formality multiplier nbw
z <= a*b;
```

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 certain ObjectID, as well as what caused that ObjectID to be selected, you can use the `report_architecture` command.

To instruct Formality to report on the multiplier architecture used in your design, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>report_architecture -all</code>	<code>report_architecture -all</code>

For more information about the `report_architecture` command and its options, see the man page.

Working With Automated Setup Files

The benefit of an automated setup file (.svf) is that it allows the implementation tool to automatically provide setup information to Formality. It helps Formality understand and process design changes caused by other tools that were used in the design flow. Formality uses this information to assist compare point matching and correctly set up verification without your 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 automated setup file (.svf). When you read the automated setup file into Formality, the tool can account for the phase inversion during compare point matching and verification.

The first step in the automated setup flow is to create the automated setup file during synthesis. Both Design Compiler and Presto record data in the automated setup file that Formality can use. Next, you instruct Formality to read this file at the start of the verification process.

Creating an Automated Setup File

You would usually obtain an automated setup file from other tools, such as Design Compiler. In Design Compiler, you can write out an automated setup file that describes the design changes. Design Compiler automatically creates an automated setup file called `default.svf` during a synthesis session.

To change the name of the file, use the `set_svf` Design Compiler command as follows:

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

When Presto or Design Compiler performs an optimization that Formality needs to know about, the relevant Formality guidance commands are added to the automated setup file (.svf). If you choose to use the `set_svf` command to specify the name of the automated setup file, you must do this before optimization takes place. Otherwise, those optimizations are not written to the user specified automated setup file. Design Compiler may also create a directory named `dwsvf_process/D`, where *process/D* is the process ID. This directory is referenced by the automated setup file. It contains information about datapath optimizations performed by Design Compiler.

To append the setup information to an existing automated setup file, use the following syntax:

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

You can use the `-append` argument if you compile a design using multiple invocations of Design Compiler. You may want to keep all the setup information in a single file rather than using a separate automated setup file for each Design Compiler invocation.

Reading an Automated Setup File Into Formality

To read an automated setup file into Formality, use the `set_svf` command. You must read in the automated setup file (.svf) before you read in any design data. Formality uses the information in the setup file during matching as well as verification. Formality automatically converts these binary files into text. It creates a directory named `formality_svf` which will contain a text file named `svf.txt` representing all automated setup files read in and also any sub-directories for any automated setup file netlists encountered.

The following example reads in the automated setup file, *myfile.svf*.

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

The `set_svf` command returns 1 if the command succeeds or 0 if the command fails. If you use `set_svf` without specifying the automated setup file (.svf) to use, Formality resets the automated setup file. However, the appropriate method for removing the stored setup data is to use the `remove_guidance` command.

Note:

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

Reading In Multiple Automated Setup Files

The commands in the automated setup 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 automated setup 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 automated setup files into Formality using the `set_svf` command or invoke the command from the Guidance tab in the GUI. To read multiple automated setup files, use the following syntax:

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


You can use the `-ordered` option to indicate that the list of automated setup files you specify is already ordered and that timestamp should not reorder them. If you use `-ordered` 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 automated setup files, `bot.svf` and `top.svf`, for Formality to process:

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

You can use the `-extension` option to load and automatically order all matching files in the directory you define, based on the extension that you define. For example, Formality automatically looks for files with the `.svf` extension. If you have automated setup files in a directory with extensions other than `.svf`, you use this option to instruct Formality to read and order those files with that extension. The ordering of the files is done using time stamp information typically found in the setup file header. Formality does not require the timestamp information to be in the header and can use specific guide commands for passing time stamp information directly.

The following example instructs Formality to load and order setup files ending in `fm` in the `fmdir` directory:

```
set_svf -extension fm fmdir
```

Automated Setup File Messaging

Formality's automated setup file flow information is correlated in a summary table and all relevant information is stored in a single directory (`formality_svf`). Any messages produced via report commands or in the `formality.log` file will reference line numbers/operations within the single `svf.txt` file automatically produced in the `formality_svf` directory.

Automated Setup File Summary Table. The automated setup file guidance summary table covers all relevant guide commands found in the automated setup file. A table similar to the one below will be produced at the end of automated setup file processing:

command	Status				Total
	Accepted	Rejected	Unsupported	Unprocessed	
fsm_reencoding:	1	0	0	0	1
reg_constant :	0	3	0	0	3
transformation					
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.

Automated setup files read:

/very/long/path/name/file1.svf

/very/long/path/name/file3.svf

Automated setup files produced:

formality_svf/

svf.txt

d1/

The above table can also be produced using `report_guidance -summary`.

The results of the status fields are:

- Accepted - Formality validated and applied this guide command to the reference design

- Rejected - Formality either could not validate or could not apply this guide command to the reference design
- Unsupported - Formality does not currently support some aspect of this guide command
- Unprocessed - Formality has not processed this guide command

For more information about the guide commands, see [Appendix C, “Shell Commands”](#) or the specific man page.

Automated Setup formality_svf Directory. Regardless of the number of .svf files read in, Formality will create a single auto-decrypted text automated setup file (svf.txt) which represents the ordered automated setup file guide commands read in. All messages related to guide commands generated will reference this file. This file, along with all decrypted dwsvf netlists, will be placed under a single directory (`formality_svf`) in the current working directory.

Users can easily examine the automated setup file contents if needed. In addition, if test cases are ever required, a single directory will contain all the relevant guide commands and associated netlist directories.

The name of `formality_svf/` will be tied to the name of the `formality.log` file and will follow the same "numbering" suffix that it uses. The integer number will always match that of the `formality.log` file.

Example:

```
set_svf foo1.svf foo2.svf foo3.svf
```

Formality will create:

```
formality.log
formality_svf/
    svf.txt
    d1/...
    d2/...
    ...
```

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

Reporting Automated Setup File Information. Several commands in Formality aid in reporting automated setup file information.

find_svf_operation:

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

Usage:

```
find_svf_operation # Get a list of automated setup file
operation IDs
```

```
[ -command ] (Find operations of the specified command type(s))
```

```
[ -status ] (Find operations with the specified status(es))
```

For command arguments, use what is found in the automated setup file summary table. Note that you do not include the "guide_" 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.

Refer to the man page for details on `find_svf_operation`.

report_svf_operation:

The `report_svf_operation` command will take a list of operation IDs and produce information regarding the same.

Usage:

```
report_svf_operation  # Report information about specified operations
                        [-guide]  (Report only the guide command itself)
                        [-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)
```

Refer to the man page for details on `report_svf_operation`.

report_guidance:

There are three main uses of the report guidance command.

1.) Produce a summary table

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

```
report_guidance -summary
```

2.) Produce a "packed" Automated Setup File

In future releases it is intended that Design Compiler will write out a single "packed" automated setup file which will be a single file binary representation containing both the svf guide commands and the dwsvf netlists associated with them.

Formality supports this format, and you can write out these "packed" automated setup files.

For instance, given the following automated setup files and associated dwsvf directories:

```
foo1.svf foo2.svf foo3.svf dwsvf-1234 dwsvf-2345  
dwsvf-3456
```

The following two commands will produce one single file (`new.svf`), which contains all the information found in the three automated setup files and the three dwsvf directories:

```
set_svf foo1.svf foo2.svf foo3.svf  
report_guidance -to new.svf -pack
```

This packed automated setup file can also be read in by Formality using the following command:

```
set_svf new.svf
```

This command can also be used as an effective way of archiving all automated setup file data.

3.) Produce a user defined text version of the Automated Setup file

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

This is very similar, but not formatted the same exact way as the automatically generated `formality_svf/svf.txt` file. Because of this, it is not reliable to use this file to correlate error messages for the current run, but could be used as input for any subsequent runs.

Automated Setup File Diagnostic Messages

Formality places detailed automated setup 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]'.
```

Automated Setup File Conversion to Text

For every run, Formality will create the `formality_svf/svf.txt` file which is a concatenated version of all svf read in. This file is to be used to correlate any messages produced during the run.

Formality can also be instructed to convert automatically the encrypted automated setup file information into ASCII text in place during the execution of the `set_svf` command. That is, a decrypted automated setup file has the suffix `.txt` appended to the original file name in the original file location. To do this, enable the following variable which is disabled by default.

Set `svf_auto_decrypt` to `true`.

Removing Information

Situations can arise that call for removing information from the Formality session prior to verification. This section describes how to remove designs, design libraries, technology libraries, and containers.

Removing Designs

Note:

Before removing a design, be sure that its removal does not affect the verification of surrounding logic. Surrounding logic can depend on states generated inside the removed design.

To remove one or more designs from the Formality environment, do one of the following:

fm_shell	GUI
Specify: <code>remove_design [-hierarchy] design_list</code>	<ol style="list-style-type: none">1. Click the Reference or Implementation tab, and then the Read Design Files tab.2. Click the Verilog, VHDL, or DB tab as applicable to your design.3. Select the design you want to remove.4. Click Remove.

The `remove_design` command is enabled only during setup mode. After you have executed the `set_top` command during the read-design flow as described in Chapter 4, Formality calls the `set_black_box` command when you specify `remove_design`. The design is treated as a black box for verification.

Removing Design Libraries

Note:

You cannot specify the `remove_design_library` command after you have executed `set_top` during the read-design flow as described in Chapter 4. Formality issues an error.

To remove one or more design libraries from the Formality environment, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>remove_design_library</code> <code>library_list</code>	<code>remove_design_library</code> <code>library_list</code>

Removing Technology Libraries

Note:

You cannot specify the `remove_library` command after you have executed `set_top` during the read-design flow as described in Chapter 4. Formality issues an error.

To remove one or more technology libraries from the Formality environment, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>remove_library [-all]</code> <code>library_list</code>	<code>remove_library [-all]</code> <code>library_list</code>

To remove a technology library from a specific container, specify the container name, as in the following example:

```
fm_shell (setup)> remove_library container1:/CBA_CORE
```

To remove a shared technology library from all containers, specify the technology library name without a container name, as follows:

```
fm_shell (setup)> remove_library CBA_CORE
```

Removing Containers

To remove one or more containers from the Formality environment, do one of the following:

fm_shell	GUI
Specify:	1. Choose Designs > Remove Reference or Designs > Remove Implementation.
<code>remove_container [-all]</code> <code>container_list</code>	2. Click Yes.

From `fm_shell`, you can use the `remove_container` command to remove any container. The GUI only recognizes the default reference and implementation containers, named `r` and `i`, respectively. Removing nondefault containers does not affect what you see in the GUI.

You generally remove containers when you want to reset the reference or implementation design and start over.

The GUI allows you to import data stored within nondefault containers into the `r` or `i` container. For more information, see [“Restoring Containers” on page 5-88](#).

Converting Objects to Black Boxes

Using black boxes in a designs can be beneficial when you are using a bottom-up verification style for hierarchical designs. For example, during the course of verification, some large blocks in the lower levels of the design hierarchy can be successfully verified. After they are verified, you might not want to spend time reverifying these subdesigns as you move up the hierarchy.

You can create black boxes in a design in one of three ways:

- After you have issued `set_top` on your design, use the `set_black_box` command on the individual design blocks or technology cells from the design library that you want converted to black boxes.
- Use the `hdlin_interface_only` variable before you begin to read in your design objects. When the design is read in, the objects you specified when setting the variable are black boxes.
- Read in your design, then issue `set_top`. If there are design references that cannot be found, by default the process terminates with an error. If you set the `hdlin_unresolved_modules` variable to `black_box`, Formality treats the missing references as black boxes.

Saving Information

Situations can arise that call for saving information in the Formality session. This section describes how to save individual containers as well as how to save the entire Formality session.

Saving Containers

Note:

You must execute the `set_top` command on the designs within a container before you can save the container.

Occasionally the time required to read a design into the Formality environment and link it can be great enough that you might not want to repeat the process. In such cases you can save the container that holds the loaded design.

To save the current container to a file, do one of the following:

fm_shell	GUI
Specify:	1. Choose Designs > Save Reference or Designs > Save Implementation.
<code>write_container</code>	2. Navigate to the appropriate file or specify it in the "File name" box.
<code>[-r -i -container container]</code>	3. Click OK.
<code>[-replace]</code>	
<code>[-quiet] filename</code>	

Issuing the `write_container` command causes Formality to save the container to a file whose extension is `.fsc`. Formality does not save any environment or design parameters. For more information about this command, see the man page.

Saving the Entire Formality Session

Note:

You must execute the `set_top` command on the designs within a session before you can save the session.

Sometimes you might find it necessary to save the entire Formality session. When you save a Formality session, you save all design data, environment parameter settings, and verification results in a single file.

To save a Formality session, do one of the following:

fm_shell	GUI
Specify:	1. Choose File > Save Session.
<code>save_session filename</code>	2. Navigate to the appropriate file or specify it in the "File name" box.
	3. Click OK.

Formality appends an .fss extension to the specified file name. If you do not supply directory information as part of the file name, Formality uses the current working directory.

Saving a session is useful when you read, set up, and verify the design by running Formality in batch mode and you want to use the GUI to interactively debug a failed verification. In such a case, you can save the session during the batch run and read it in later after invoking the GUI version of Formality. For more information, see ["Restoring Information" on page 5-88](#).

Saving the session is also useful for long debugging sessions. You can save the session and continue later when it is convenient to do so.

Note:

Only containers can be used between different Formality versions; sessions cannot. For archival purposes or for sending protected information, use containers with the appropriate Formality run scripts.

Restoring Information

Formality allows you to restore information by reading previously saved containers or by reading in a previously saved Formality session. This section describes how to restore both types of data.

Restoring Containers

Note:

Containers saved from releases prior to Formality version 2002.03 must have been linked or verification fails. In addition, you must execute `set_top` on the designs within the pre-2002.03 containers before performing verification.

To restore a saved container, do one of the following:

fm_shell	GUI
Specify: <code>read_container</code> <code>[-r -i -container</code> <code>container_name]</code> <code>[-replace] file_name</code>	<ol style="list-style-type: none">1. Choose Designs > Restore Reference or Designs > Restore Implementation.2. Navigate to the appropriate file or specify it in the “File name” box.3. Click OK.

Issuing the `read_container` command causes Formality to read the file and restore libraries to a container having the same name as the original saved container. For more information about this command, see the man page.

In the GUI, you can only restore data saved within the default `r` and `i` containers. If you want to restore data located in nondefault containers for viewing within the GUI, you must first import the data into the `r` or `i` container. Ensure that data currently located in the `r` or `i` container has been saved.

To import design data into the r or i container for viewing in the GUI, do the following:

1. Choose Designs > Choose New Reference or Designs > Choose New Implementation.
2. Select the design you want to import into r or i.
3. Click OK.

Restoring a Session

To restore a previously saved Formality session, do one of the following:

fm_shell	GUI
Specify:	1. Choose File > Restore Session.
<code>restore_session file_name</code>	2. Navigate to the appropriate file or specify it in the "File name" box.
	3. Click OK.

Issuing the `restore_session` command causes Formality to discard all information in the current session and then restore all containers, design data, setup parameters, and verification results from the specified file. For more information about this command, see the man page.

6

Compare Point Matching and Verification

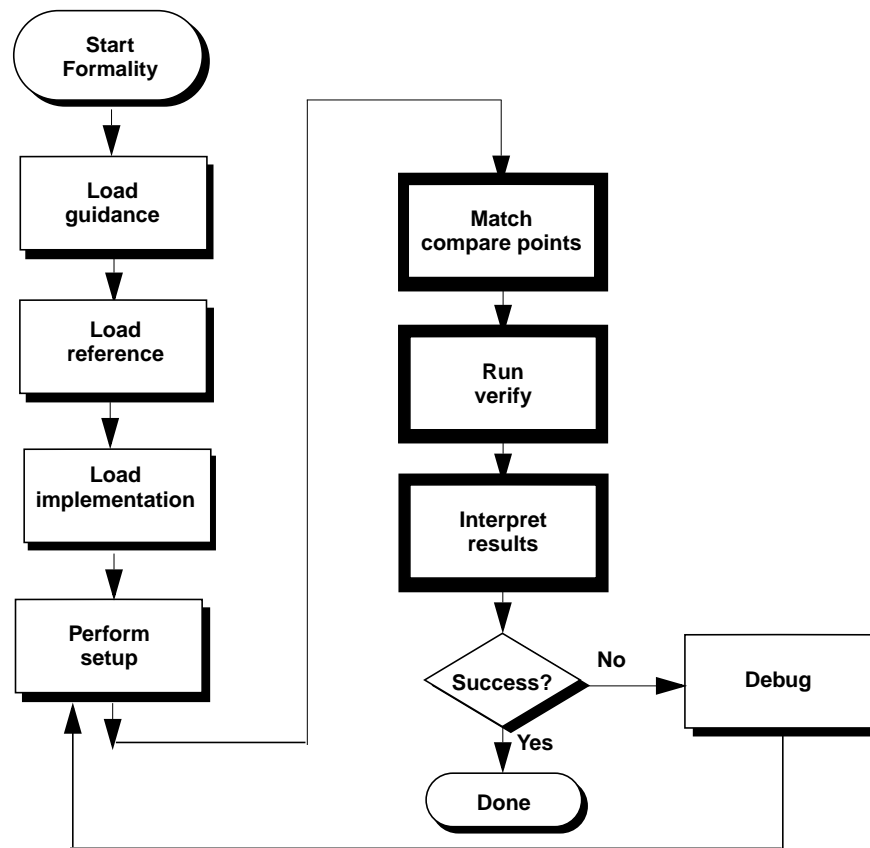
After you have prepared your verification environment and set up your design, you are ready to match compare points and then verify the design. This chapter describes how to match compare points and verify one design against another. It also offers some tips for batch verifications, interpreting results, and saving data.

This chapter includes the following sections:

- [Matching Compare Points](#)
- [Performing Verification](#)
- [Reporting and Interpreting Results](#)

This chapter's subject matter pertains to the boxes outlined in [Figure 6-1](#).

Figure 6-1 Design Verification Process Flow Overview



Matching Compare Points

Prior to verification, Formality must match compare points in the designs as described in [“Compare Points” on page 1-25](#). This matching occurs automatically when you specify the `verify` 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.

Formality allows you to match compare points in a separate step prior to specifying the `verify` command. Consequently you can iteratively debug unmatched compare points, as follows:

- Perform compare point matching.
- Report unmatched points.
- Modify or undo results of the match, as needed.
- Debug the unmatched compare points.
- 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 in [Chapter 5, “Preparing the Design for Verification,”](#) because the following commands and variables cannot be changed in the matched state:

```
set_cutpoint
remove_black_box
remove_constant
remove_cutpoint
remove_design
remove_inv_push
remove_object
remove_parameters -resolution -retimed -all_parameters
remove_resistive_drivers
rename_object
set_black_box
set_constant
set_direction
set_equivalence
set_fsm_encoding
set_fsm_state_vector
set_inv_push
set_parameters -resolution -retimed
ungroup
uniquify
verification_assume_reg_init
verification_auto_loop_break
verification_clock_gate_hold_mode
verification_constant_prop_mode
verification_inversion_push
verification_merge_duplicated_registers
verification_set_undriven_signals
verification_use_partial_modeled_cells
```

You can return to setup mode by using the `setup` command, but this causes all points matched during match mode to become unmatched.

Performing Compare Point Matching

To match compare points, do one of the following:

fm_shell	GUI
Specify:	1. Click the Match tab.
<code>match -datapath -hierarchy</code>	2. Click Run Matching.

This command attempts to match 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 7-19](#).

The matching results from incremental matching can differ from those you receive when you run the `match` command once 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.

Also, you can instruct Formality to match datapath blocks or all hierarchical blocks during compare point matching by using the `-datapath` or `-hierarchy` option, respectively. By default, datapath or hierarchical blocks are matched during compare point matching only when you set these options.

You can interrupt matching by pressing Control-c. All matched points from the interrupted run remain matched.

To return to setup mode, specify the `setup` 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 `remove_compare_rules` command and the `remove_user_match` 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, do one of the following:

fm_shell	GUI
Specify:	Click Match > Unmatched.
<pre>report_unmatched_points [-compare_rules] [-datapath] [-substring string] [-point_type point_type] [-status status] [-except_status status] [-method matching_method] [-last] [[-type ID_type] compare_point...]</pre>	

This command reports compare points, input points, and higher-level matchable objects that are unmatched. Use the options to filter the report as desired. For more information about options, see the man page.

To view a list of matched points, specify the `report_matched_points` command or click Match > Matched in the GUI. 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 click Match > Summary. For more information, see the man page.

Undoing Matched Points

To undo the results of the `match` command, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>undo_match [-all]</code>	<code>undo_match [-all]</code>

This command returns all points matched during the most recent `match` command back to their unmatched state. It returns you to the matched state achieved by the previously specified `match` command. Use the `-all` option to undo all matches.

You remain in the matched state even if you undo the first match or specify the `-all` option. To return to the setup state, specify the `setup` command in `fm_shell` or at the Formality prompt in the GUI.

This command is especially useful when you have made changes that did not achieve the results you desired for compare point matching.

Debugging Unmatched Points

After you report the unmatched compare points, you can use any or all of the following user-specified techniques to match them:

One of the most common reasons for unmatched compare points is design transformations incurred during Design Compiler processing. The intent of these transformations is not to change the features of the design, but to optimize it for speed or by area, or to prepare the design for back-end tools. Unfortunately, such design transformations can cause severe compare point matching problems because object names often change significantly.

Common transformations include moving features up and down the design hierarchy, explicitly applying name rules to objects in the design, and eliminating constant registers.

To resolve compare point mismatches caused by Design Compiler design transformations, you can specify a change file generated by Design Compiler as input to Formality. This encrypted file lists all the name changes incurred during Design Compiler processing. Formality uses this file to map the new names back to their originals for compare point matching.

- Matching with user-supplied names
- Matching with compare rules
- Matching with name subset
- Renaming user-supplied names or mapping file

Each matching technique is described in detail in [“Unmatched Points” on page 7-15](#).

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. For more information, see [“Changing Bus Naming and Dimension Separator Styles” on page 4-9](#).

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 6-1](#) shows the actions you can take for unmatched compare points.

Table 6-1 *Unmatched Compare Points Action*

Symptom	Possible cause	Action
Same number of unmatched points in reference and implementation designs	Names have undergone a transformation	Use <code>set_user_match</code> command. Write and test compare rule. Modify name match variables. Turn on signature analysis. For all, see “Unmatched Points” on page 7-15 .

Table 6-1 Unmatched Compare Points Action

Symptom	Possible cause	Action
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 7-28 .
	Black box created for missing cells	Reread reference design, including the missing cells. Make black box in reference design.

How Formality Matches Compare Points

As described in [“Compare Points” on page 1-25](#), compare point matching is either named-based or non-name-based. The following matching techniques occur by default when you match compare points.

Name-based matching:

- Exact-name matching
- Compare point matching based on net names
- Name filtering

Non-name-based matching:

- Topological equivalence
- Signature and topological analysis

These matching techniques are executed in the following order:

- Exact-name matching
- Name filtering
- Topological equivalence
- Signature analysis
- Compare point matching based on net names

Note:

Four other user-specified matching techniques are also available, but you normally use them to debug unmatched points. For more information, see [“Unmatched Points” on page 7-15](#).

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.

All name matching in Formality is case-insensitive. The following sections describe each default compare point matching technique.

Table 6-2 lists variables that control matching. Some are described in the following sections. For more information about these variables, see the man pages.

Table 6-2 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, 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 follows:

fm_shell	GUI
Specify:	1. Click Match.
<code>set name_match</code> <code>[all none port cell]</code>	2. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.
	3. From Matching, select the <code>name_match</code> variable.
	4. In the "Choose a value" list, select all, none, port, or cell.
	5. Choose Edit > Close.

The default value, "all," enables all 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.

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 will be reversed. For more information about Formality multibit support, see ["Supporting Multibit Library](#)

Cells” on page 5-5. In the GUI, you can access this variable from the Formality Tcl Variable Editor dialog box by choosing Edit > Formality Tcl Variables, then from Matching, select the variable.

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, do one of the following:

fm_shell	GUI
Specify: <code>set name_match_use_filter false</code>	<ol style="list-style-type: none">1. Click Match.2. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.3. From Matching, select the <code>name_match_use_filter</code> 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 `name_match_filter_chars` variable. The default character list is ``~!@#$$%^&*()_-=|\\[]{}"'<>? ,./`

For example, the following command resets the filter characters list to include “V”:

```
fm_shell (match)> set name_match_filter_chars \
    {~!@#$$%^&*()_-=|\\[]{}"'<>? ,./V}
```

Topological Equivalence

Formality attempts to match the remaining unmatched compare points by topological equivalence. In other words, 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 uniquely identify 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` command.

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 or hierarchical blocks and pins, set the

`signature_analysis_match_datapath` or `signature_analysis_match_hierarchy` variable to false, respectively. 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 by doing one of the following:

fm_shell	GUI
<p>Specify:</p> <pre>set signature_analysis_match_compare _points false</pre>	<ol style="list-style-type: none">1. Click Match.2. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.3. From Matching, select the <code>signature_analysis_match_compare_points</code> 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 `signature_analysis_match_primary_output` variable to true. In addition, signature analysis does try to match primary input ports.

You might reduce matching runtimes by writing a compare rule rather than disabling signature analysis. Compare rules, for example, work well if there are extra registers in both the reference and implementation designs. For more information, see [“Matching With Compare Rules” on page 7-19](#).

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, do one of the following:

fm_shell	GUI
Specify: <code>set name_match_based_on_nets false</code>	<ol style="list-style-type: none">1. Click Match.2. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.3. From Matching, select the <code>name_match_based_on_nets</code> 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.

Performing Verification

When you issue the `verify` 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.

Verifying a Design

To verify the implementation design against the reference design, do one of the following:

<code>fm_shell</code>	GUI
Specify:	1. Click Verify.
<code>verify</code> <code>[reference_designID]</code> <code>[implementation_designID]</code>	2. Click Verify All.

If you omit the reference and implementation design IDs from the command, Formality uses the reference and implementation designs you specified when you read in your designs. For more information, see [“Reading in Libraries and Designs” on page 4-3](#).

If you did not match compare points prior to verification as described in [“Matching Compare Points” on page 6-3](#), 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 prior to 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 super-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. Set the `verification_super_low_effort_first_pass` variable to `true` (the default) to force the `verify` command to run this super-low-effort verification first. Afterward, Formality proceeds with verifying all the remaining compare points.

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 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.

If you do not want Formality to independently verify blocks under a certain level, use the `-level` 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.

For more information about the `verify` command, see the man page. For information about interpreting results, see [“Reporting and Interpreting Results” on page 6-34](#).

Verifying a Single Compare Point

Single compare point verification is useful when you have trouble verifying a complete design and you have isolated problem areas in the implementation design.

To verify a single compare point, do one of the following:

fm_shell	GUI
Specify:	1. Click Verify.
<code>verify [-type type]</code> <code>objectID_1 objectID_2</code> <code>-inverted</code> <code>[-constant0 -constant1]</code>	2. Select a compare point in the list. 3. Click Verify Selected Point.

Sometimes design objects of different type share the same name. If this is the case, change the `-type` option to the unique object type. For more information about this command, see the man page.

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 given points, use the `-inverted` switch to the `verify` command.

In addition, you can verify a single compare point with a constant 1 or 0. Using either the `-constant0` or `-constant1` option to the `verify` 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. For more information about the `-constant` option, see the `verify` command man page.

To verify a subset of compare points, see [“Removing Compare Points From the Verification Set” on page 6-22](#). For information about interpreting results, see [“Reporting and Interpreting Results” on page 6-34](#).

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 need to pinpoint problems in a failed verification.

To prevent Formality from checking for design equivalence between two objects that constitute a matched compare point, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>set_dont_verify_point</code> <code>[-type ID_type]</code> <code>[object_1 [object_2] ...]</code>	<code>set_dont_verify_point</code> <code>[-type ID_type]</code> <code>[object_1 [object_2] ...]</code>

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 `-type` option to the unique object type. For more information, see the man page.

Specify instance-based path names or object IDs for compare points in the reference and implementation designs. Black boxes and hierarchical blocks are not compare points, but black box input pins are compare points.

Specify the `remove_dont_verify_point` command to undo the effect of `set_dont_verify_point` 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_point`. These commands accept instance-based path names or object IDs. For more information about these commands, see the man pages.

Controlling Verification Runtimes

In addition to the `signature_analysis_match_compare_points` variable described in “[Signature Analysis](#)” on page 6-16, an environment variable exists that you can use to limit verification runtimes.

To control the verification time limit specify the maximum number of failing compare points allowed before Formality halts the verification process by doing the following:

fm_shell	GUI
Specify: <code>set verification_timeout_limit value</code>	<ol style="list-style-type: none">1. Click Verify.2. Choose Edit > Formality Tcl Variables or the Modify Formality Tcl Variables toolbar option. The Formality Tcl Variable Editor dialog box appears.3. From Verification, select the <code>verification_timeout_limit</code> variable.4. In the Enter a time box, enter none for no limit or specify a time in (hh:mm:ss) format and press Enter.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. For more information about this environment variable, see the man pages.

Distributing Verification Processes

You can execute Formality verification processes in parallel across several CPUs to increase performance. The following sections describe a typical user flow for working with parallel verification.

Setting Up the Distributed Environment

You must set the default working directory to FM_WORK/server; this is the storage area for all the files required for exchanging data and information between the various machines. Make sure that the directory is accessible to each machine involved in the distributed process and that each machine is able to read from and write to this directory. If this directory is not accessible from any one of the servers, or if you want to use another directory as a working directory, you must change the working directory with the `distributed_work_directory` Tcl variable. For example,

```
fm_shell (setup)> set distributed_work_directory /home/  
myname/  
dist/work
```

You specify the working directory by using an absolute path name starting from the root of the system. Relative paths are not supported.

After you set the working directory, you have two options for specifying distribution. First, you can populate the distributed servers list. For example,

```
fm_shell (setup)> add_distributed_processors {pandora  
hermes}
```

After defining the distributed servers list, you receive messages from Formality similar to the following:

```
Arch: sparc-64, Users: 22, Load: 2.18 2.14 2.17
Arch: sparc-64, Users: 1, Load: 1.45 1.41 1.40
```

For each machine, Formality checks that `fm_shell` can be started on the remote machine; that the release version matches the master; and that the distributed work directory is accessible. Formality prints the type of platform (architecture), the number of users currently logged on to that machine, and the load average. There is a limit to the number of servers you can add; you can have one master, with four additional CPUs during verification. To avoid starting more distributed servers than the number of available processors, determine the number of processors on that machine.

A second option for specifying distribution is to specify an LSF executable along with the number of servers. For example,

```
add_distributed_processors -lsf bsub_exec -nservers d
-toptions
"optionslist"
```

If using the `-lsf` option, you need to specify the path to the LSF submission executable.

You can use the `report_distributed_processors` command to report the list of servers. For example,

```
fm_shell (setup)> report_distributed_processors
Working directory==> ``/remote/dtg654/fm/dfs'' (32bit)
-----***-----
MACHINE: pandora [ARCH: sparc-64]
MACHINE: hermes [ARCH: sparc-64]
-----***-----
```

Formality lists the name of the machine and its architecture; one line per server. It also displays the working directory in the report along with the type of executable in use (32 bit or 64 bit). The master

machine automatically determines the executable type, subject to the setting of the `distributed_64bit_mode` Tcl variable. This variable is false by default; therefore, spawned servers run a 32-bit executable by default.

On occasion you might want to remove a server from the server list, for instance if you have an overloaded machine. In this situation, you use the `remove_distributed_processors` command to remove servers from the server list. For example,

```
fm_shell (setup)> remove_distributed_processors pandora
fm_shell (setup)> report_distributed_processors
Working directory ==> ``/remote/dtg654/fm/dfs'' (32bit)
-----****-----
MACHINE: hermes [ARCH: sparc-64]
-----****-----
2.3.2 Starting Distributed Equivalence Checking
```

When you add servers and set the `distributed_verification_mode` to enable (the default) Tcl variable, the `verify` command is executed in distributed mode:

```
fm_shell (setup)> verify ref:/WORK/test imp:/WORK/test
```

At the end of the verification, Formality reports verification results in the same format as in serial process mode.

If unexpectedly one server process stops (for example, due to an internal error or system problem), the entire `verify` command is immediately terminated.

Verifying Your Environment

Whenever Formality starts a distributed server, it executes the `fm_shell` command. Your `$PATH` environment variable must find the `fm_shell` command on each distributed host.

Remote Shell Considerations. Formality relies on the `rsh` (`remsh` for HP platforms) UNIX command to start a process on a remote machine. This command runs the login shell on the remote host and executes your shell initialization file. Interactive commands, runtime errors, and environment settings can cause remote execution to fail. The following notes can help diagnose such problems.

You need to have special privileges to start a distributed process with an `rsh` (or `remsh`) command. In many UNIX installations, those privileges are given by default. However, the system administrator might have changed them. If you experience a problem starting servers and suspect it is due to a problem with `rsh`, you can test remote execution from the UNIX shell command prompt by using the following command:

```
% rsh distributed_server_machine fm_shell
```

If you get an error message, its cause can be either commands or environment settings in your shell initialization file or privilege settings that prevent you from executing a remote shell.

Tuning the Shell Resource File. Be aware of what is in your shell initialization file, such as `.cshrc` and `.profile`, to avoid having shell commands that have the following behavior:

- They interact with the user (that is, the shell asks you to enter something from the keyboard). Because you do not have the ability to answer (distributed processes are not interactive), the process might hang while waiting for an answer to a question it does not see.
- They require some GUI display. The `DISPLAY` environment variable is not set on the servers. X Windows clients fail.

If you have any trouble using `add_distributed_processors`, you might want to have a dedicated shell initialization file for running distributed tasks.

Interrupting Verification

To interrupt verification, press Control + 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.

Performing Hierarchical Verification

By default, Formality incorporates a hybrid verification methodology that combines the easy setup associated with flat verification with the benefits of traditional hierarchical performance capacities. From a user perspective, Formality appears to be performing a flat verification, regardless of the design hierarchy, and all results are presented in the context of the flat top-level design.

Formality allows you to perform a traditional hierarchical verification that is helpful when you want to view explicit, block-by-block hierarchical results.

To perform traditional hierarchical verification, do one of the following:

fm_shell	GUI
Specify:	Specify:
<pre>write_hierarchical_verification_ script [-replace] [-noconstant] [-noequivalence] [-match type] [-save_directory <i>pathname</i>] [-save_file_limit <i>integer</i>] [-level integer] <i>filename</i></pre>	<pre>write_hierarchical_verification_ script [-replace] [-noconstant] [-noequivalence] [-match type] [-save_directory <i>pathname</i>] [-save_file_limit <i>integer</i>] [-level integer] <i>filename</i></pre>

This command generates an editable Tcl script that you can source to perform traditional hierarchical verification. You can customize this script to verify specific blocks, as well as to specify constraining context information about instantiated blocks. For more information about this command, see the man page.

The Tcl script instructs the tool to initially attempt verification on comparable lower hierarchical blocks in their isolated context. Verification starts at the lowest levels of hierarchy and works upward. Explicit setup commands are generated to capture top-level context.

The script specifies that each block be verified once, regardless of the number of instantiations. It reports the verification result for each block in a text file that is concatenated to the transcript.

By default, for each matched block for the current top-level implementation and reference designs, the Tcl script

- Generates black boxes for matched subdesigns.
- Removes unused compare points.

- Sets port matches for ports matched by means other than their names.
- Sets input port constants. Override this behavior by specifying the `-noconstant` option.
- Sets input port equivalences for unmatched input ports known to be equivalent to other matched ports. Override this behavior by specifying the `-noequivalence` option.
- Verifies the target block as a top-level design.
- Saves the Formality session if the verification fails. Override this behavior by specifying the `-save_file_limit` option.

The script ignores inconsistent setup information for port matches, constants, and equivalencies, and a comment appears in the generated script.

The script produced by `write_hierarchical_verification_script` is designed to run in the same session that it is created. If you run the hierarchical verification script separately, you must manually insert commands that read and link the reference and implementation designs.

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 once. 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

Given 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  
.  
.  
.  
? shell_command  
? !
```

- 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 `-f` 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 -f 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 6-34](#).

Controlling Verification

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  
}
```

Interrupting Verification

To interrupt the batch job, press Control-c from the shell. Doing so causes script processing to stop. Any Formality process interrupted is immediately stopped, and no intermediate results are retained. Note that it is not possible to have Formality continue verification from the point of interruption.

Verification Progress Reporting

You can specify how much time is to elapse between each progress report by using the `verification_progress_report_interval` variable. During

long verifications, Formality issues a progress report every 30 minutes. For updates more or less frequently, you can set the value of this variable in minutes to the desired interval.

Reporting and Interpreting Results

As part of your troubleshooting efforts, Formality allows you to report on passing, failing, unverified, and aborted compare points. Do one of the following:

fm_shell	GUI
Specify any of the following commands:	1. Click Debug.
<code>report_passing_points</code> <code>[-point_type <i>point_type</i>]</code>	2. Click the Passing Points, Failing Points, Aborted Points, or Unverified Points tab.
<code>report_failing_points</code> <code>[-point_type <i>point_type</i>]</code>	
<code>report_failing_unverified</code> <code>[-point_type <i>point_type</i>]</code>	
<code>report_aborted_points</code> <code>[-point_type <i>point_type</i>]</code>	

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, you can display compare points with either their original names or the names that they were mapped to due to the compare rules. To choose either original names or mapped names, click the display name.

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.

During verification, Formality assigns one of three types of status messages for each compare point it identifies:

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.

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.

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.

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

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

If verification is interrupted, either because you press Control-c or a user-defined time-out occurs (such as the CPU time limit), and if at least one failing point was detected prior to the interruption, Formality reports a verification failure.

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 Control-c or a user-defined time-out occurred, and no failing compare points were detected prior to the interruption.

For information about what to look for when you have an inconclusive verification due to an aborted compare point, see [“Handling Designs When Verification Is Incomplete”](#) on page 7-4.

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

7

Debugging Failed Design Verifications

This chapter describes procedures that help you find problem areas in a design that fails verification.

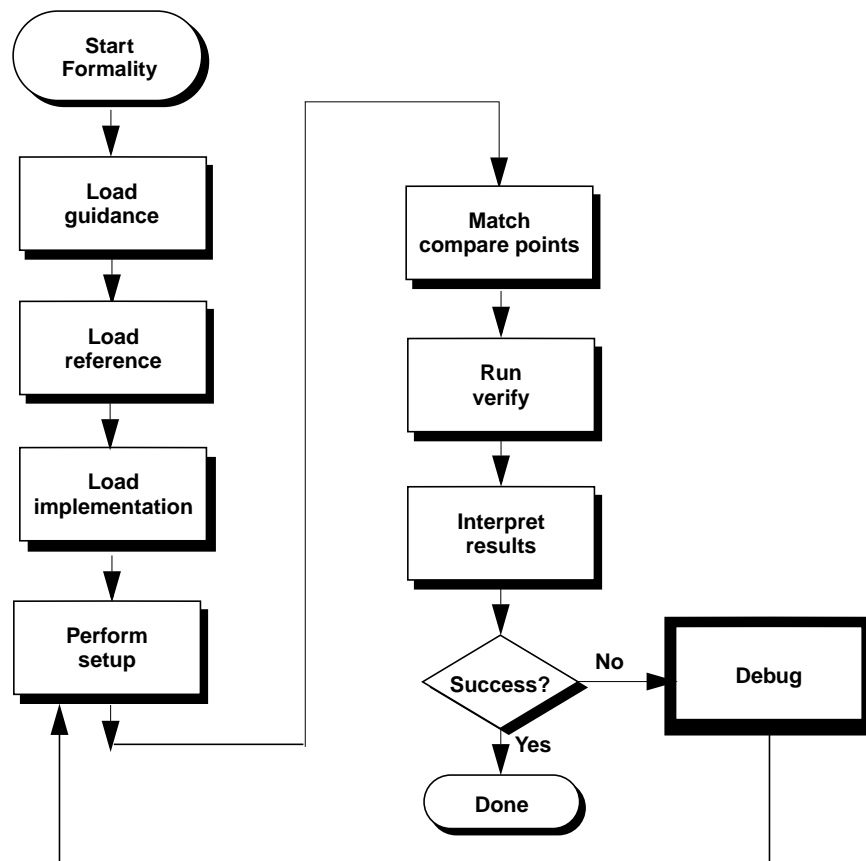
It contains the following sections:

- Debugging Process Flow
- Gathering Information
- Handling Designs When Verification Is Incomplete
- Determining Failure Causes
- Debugging by Using Diagnosis
- Debugging by Using Logic Cones
- Eliminating Setup Possibilities
- Working With Schematics

- Working With Logic Cones
- Working With Failing Patterns

This chapter's subject matter pertains to the box outlined in Figure 7-1.

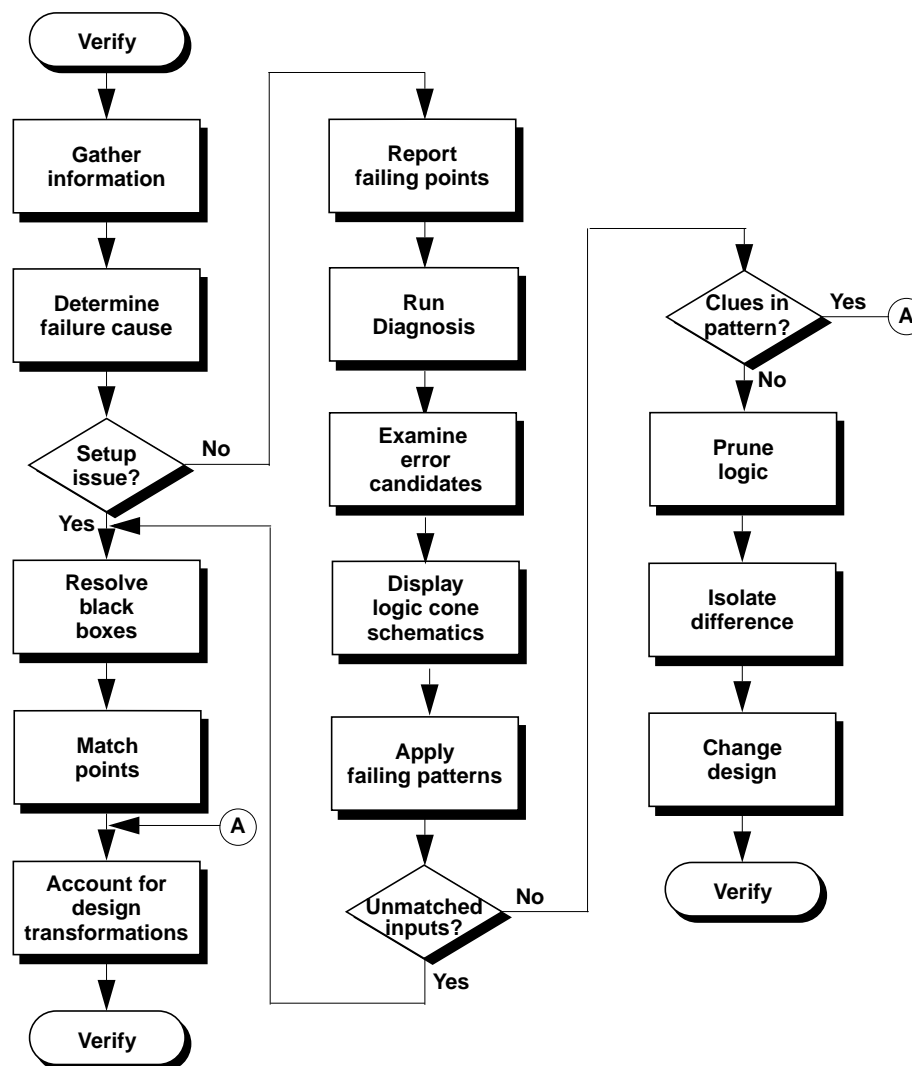
Figure 7-1 Design Verification Process Flow Overview



Debugging Process Flow

Figure 7-2 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 8, “Technology Library Verification.”](#)

Figure 7-2 Debugging 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 information sources are the following:

- The transcript window provides information about verification status, black box creation, and simulation/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.

Handling Designs When Verification Is Incomplete

Occasionally, Formality encounters a design that cannot be verified because it is particularly complex. For example, asynchronous state-holding loops can cause Formality to abort verification if you did not check for their existence prior to executing the `verify` command. For more information, see [“Eliminating Asynchronous State-Holding Loops” on page 5-10](#).

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 7-7](#), which describes what to do when verification finishes but fails.

Note:

Incomplete verifications can occur when Formality reaches a prespecified 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 6-8](#).
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. The `set_black_box` command allows you to specify the designs that you want to black box. For more information about this command, see the man page.

Alternatively, you can insert cutpoint black boxes to simplify hard-to-verify designs, as described in [“Working With Cutpoints” on page 5-12](#).

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 recommended incremental verification technique described in [Figure 1-1 on page 1-5](#).

7. Prior to running verification a second time, match by hand 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 successfully preverify any black box equivalent datapath blocks. For additional information, see the man page.

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. If 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 7-1](#).

Table 7-1 Determining 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 7-14](#) to help you fix them. You must fix setup problems and then reverify 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 specifying 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 6-3](#) 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 GUI's Formality 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 macrocells (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 handling black boxes in your design, see [“Working With Black Boxes” on page 5-15](#).

3. Check for incorrect environment variable settings, especially for the design transformations listed in [“Design Transformations” on page 7-28](#). To view a list of current variable settings, use the `printvar` command.

Debugging by 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, “Quick Start With Formality.”](#)

After you have run verification, debug your design by doing the following:

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 Diagnose.

Note:

After clicking Diagnose, 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 the recommended ones 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, 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 7-43](#).

Note:

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

Debugging by 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 show the entire set of failing input patterns, click the Show Patterns toolbar option 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, if you see two adjacent unmatched cone inputs (one in the references and one in the implementation design) that have opposite values on all patterns, these unmatched cone inputs should probably 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 7-14](#). 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 [“Working With Failing Patterns” on page 7-44](#).

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 [“Working With Logic Cones” on page 7-38](#).

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 prior to verification, see [“Design Transformations” on page 7-28](#).

4. Prune the logic cones and subcones, as needed, to better isolate the problem.

For more information, see [“Pruning Logic” on page 7-43](#).

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 further help you debug 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 7-7](#) 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
- Executing `report_unmatched -point_type bbox` command
- Executing the `report_black_boxes` command in the Formality shell or Formality prompt from within the GUI. For more information about this command, see the man pages.

For more information about black boxes, see [“Working With Black Boxes” on page 5-15](#).

Unmatched Points

As described in [“Debugging Unmatched Points” on page 6-8](#), you might need to manually match compare points by using the techniques described in this section. Normally, you do this during the compare point matching process, prior to 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.

Important:

Use caution when adding and removing 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, do one of the following:

fm_shell	GUI
Specify:	1. Click Match > Unmatched Points.
<code>set_user_match</code>	2. Select a point in the reference list.
<code>[-type ID_type]</code>	3. Select a point in the implementation list.
<code>[-inverted] [-noninverted]</code>	4. Select +, -, or ?.
<code>object_1 object_2 [...]</code>	5. Click the User Match Setup tab to view the list of user-specified matches.

Sometimes design objects of different types share the same name. If this is the case, change the `-type` option to the unique object type. For more information about the `set_user_match` command, see the man page.

You can set the `-inverted` or `-noninverted` 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 `-inverted` option matches the specified objects with inverted polarity; the `-noninverted` option matches the specified objects with noninverted 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] \
               $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
```

The `set_user_match` command allows you 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
```

You can use the `set_user_match` command to match these design objects as follows:

```
fm_shell (verify)> set_user_match ref:/WORK/CORE/carry_in \
```

```
impl:/WORK/CORE/cin
```

Removing User-Matched Compare Points. To unmatch objects previously matched by the `set_user_match` command, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>remove_user_match</code> <code>[-all] [-type type]</code> <code>instance_path</code>	<code>remove_user_match</code> <code>[-all] [-type type]</code> <code>instance_path</code>

This command accepts instance-based path names and object IDs. For more information about the `remove_user_match` command, see the man page.

Listing User-Matched Compare Points. You can generate a list of points matched by the `set_user_match` command. Do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>report_user_matches [-inverted </code> <code>-noninverted -unknown]</code>	<code>report_user_matches [-inverted </code> <code>-noninverted -unknown]</code>

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 [“Compare Rules” on page 1-28](#), 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.

Important:

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 allows you to affect many design objects during compare point matching. For example, suppose 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.

Defining Compare Rules. To create a compare rule, do one of the following:

fm_shell	GUI
Specify:	1. Click the Match > Compare Rule tab.
<pre>set_compare_rule -from search_pattern -to replace_pattern designID</pre>	2. Click Add, and then click the Reference or Implementation tab.
	3. Select a library, and select a design as needed.
	4. Type the initial search pattern in the Search value box, and type the replacement search pattern in the Replace value box.
	5. Select the object type: Any, Port, Cell, or Net.
	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.

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. Hence, 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:

```
fm_shell (verify)> set_compare_rule $ref \  
-from {_reg2\[ \([0-1]\) \] \[ \([0-1]\) \] $} \  
-to {\1_reg2\1_\2}
```

This rule transforms all objects in the reference design that follow the format `name_reg#[#][#]` to `name_reg##_#`, 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 unambiguously determine them to be arithmetic operators. 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:

```
fm_shell (verify)> set_compare_rule ref:/WORK/design_name \  
-from {bus\[ \([0-9]*\) \]} -to {bus\[ \([15-\1\) \]}
```

The “-” operator in the replace pattern means arithmetic minus.

Testing Compare Rules. You can test name translation rules on unmatched points or arbitrary user-defined names by doing one of the following:

fm_shell	GUI
<p>Specify:</p> <pre>test_compare_rule [-designID -r -i] -from search_pattern -to replace_pattern [-substring string] [-type type]</pre> <p>Or</p> <pre>test_compare_rule -from search_pattern -to replace_pattern -name list_of_names</pre>	<ol style="list-style-type: none"> 1. Click the Match > Compare Rule Setup tab. 2. Click Set. 3. Set the rule on a specific design by clicking either the Reference or Implementation tab. 4. Set the object name, and type the search pattern and replace pattern in their respective boxes. 5. Click the test button and select Test With Unmatched Points or Test With Specified Names. <ul style="list-style-type: none"> - If you select the Test With Unmatched Points tab, you can optionally type a substring that restricts the test to those unmatched points that contain the inputted substring. - 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, then click Add. 6. 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—0 for failure, 1 for success.

Removing Compare Rules. To remove all compare rules from a design, do one of the following:

fm_shell	GUI
Specify:	1. Click the Match > Compare Rules tab.
<code>remove_compare_rules [designID]</code>	2. Click Remove.
	3. Select a design, and then click OK.

Currently it is not possible to remove a single compare rule. For more information about the `remove_compare_rules` command, see the man page.

Listing Compare Rules. To track compare rules, you can generate reports that list them by doing one of the following:

fm_shell	GUI
Specify:	Click the Match > Compare Rules tab.
<code>report_compare_rules [designID]</code>	

Each line of output displays the search value followed by the replace value for the specified design. For more information about the `report_compare_rules` command, see the man page.

Matching With Name Subset

During subset matching, each name is viewed as a series of tokens, separated by characters in the `name_match_filter_chars` 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(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 in at least 90 percent of all names of a given type of object (as long as 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, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>rename_object</code> <code>-file file_name</code> <code>[-type object_type]</code> <code>[-shared_lib]</code> <code>[-container container_name]</code> <code>[-reverse] objectID</code> <code>[new_name]</code>	<code>rename_object</code> <code>-file file_name</code> <code>[-type object_type]</code> <code>[-shared_lib]</code> <code>[-container container_name]</code> <code>[-reverse] objectID</code> <code>[new_name]</code>

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. For more information about the `rename_object` command, see the man page.

Note:

To rename multiple design objects from a file, select the `-file` option. The file format should match that of the `report_names` command in Design Compiler.

The `rename_object` command also allows you 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/clock_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 (perhaps inadvertently).

You can also use the `change_names` 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 `change_names` command). To work around this problem, obtain the changed-names report from Design Compiler and supply it to Formality with the `rename_object` 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 5, “Preparing the Design for Verification,”](#) for the following common design transformations:

- Internal scan insertion on [page 5-38](#).
- Boundary scan on [page 5-39](#).

- Clock tree buffering on [page 5-41](#).
- Asynchronous bypass logic on [page 5-43](#).
- Clock gating on [page 5-45](#).
- Inversion push on [page 5-50](#).
- Reencoded finite state machines on [page 5-53](#).
- Retimed designs on [page 5-59](#).

Working With 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, do the following:

- Right-click in a design in any of the following report windows:

Verify

Match > Unmatched and Match > Unmatched

Debug > Failing Points, Debug > Passing Points, Debug > Aborted Points

- Choose View > View Reference Object or View > View Implementation Object.

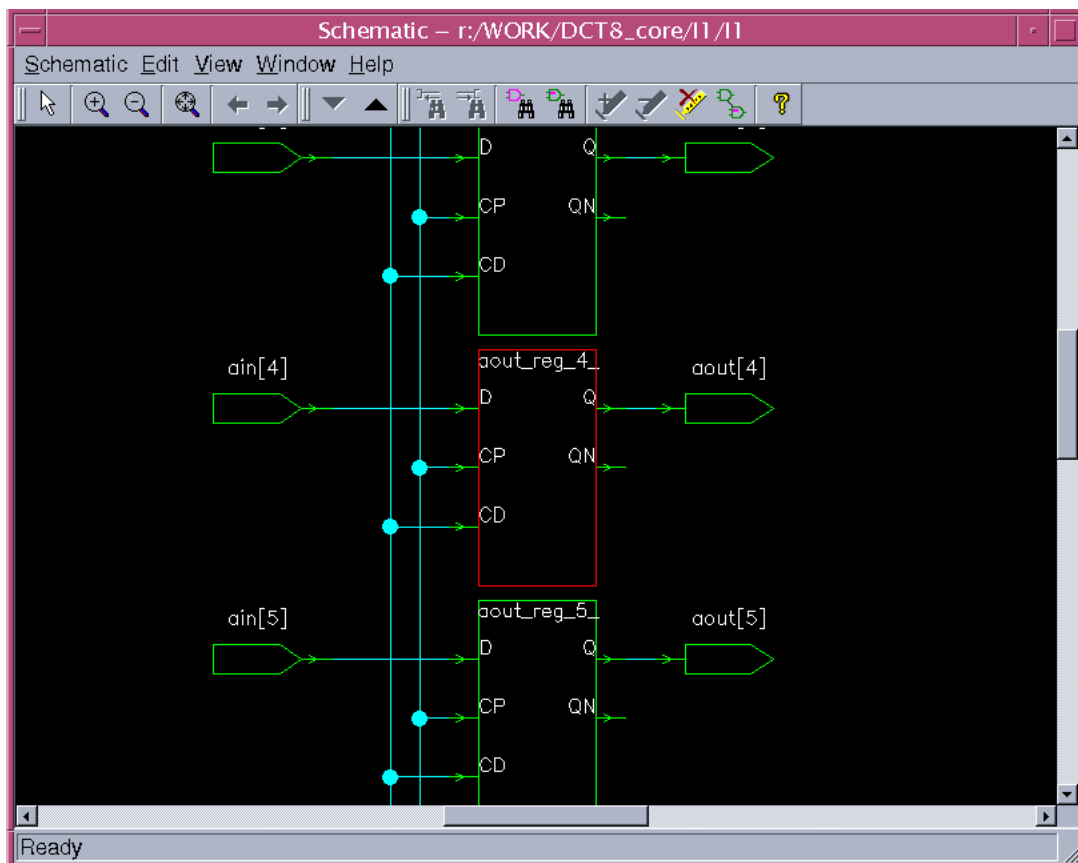
After you perform these steps, a schematic view window appears that shows the selected object in the context of its parent design. The object is highlighted and centered in the schematic view window.

From the schematic view window you can zoom in and zoom 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 > Preferences > 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 7-3](#) shows a schematic view window.


Figure 7-3 Schematic View Window



Some of the more important areas are the following:

Toolbar

The toolbar contains tools that act as shortcuts to some menu selections. The schematic viewer supports the following tool options:

-  Click to select particular sections of the design.



Click to increase the magnification applied to the schematic area by approximately two times (2X).



Click to decrease the magnification applied to the schematic area by approximately 2X.



Click to redraw the displayed schematic sheet so that all logic is viewable.



Click to view the previous view.



Click to view the next view.



Click to find the driver for the selected net.



Click to find the load on the selected net.



Click to find and zoom to the compare point in the schematic.



Click to display the object finder dialog box to find an object by name in the schematic.



Click to set highlighting on the selected objects.



Click to remove highlighting from the selected objects.



Click to clear highlighting from all objects.



Click to display the name visibility dialog box where you can control the visibility of the objects.



Click to bring up help for the schematic view window.

Schematic area

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.

Formality displays the wire connections in different colors to represent the different coverage percentages of the error candidates. While debugging, concentrate on the nets with the highest coverage percentage.

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 Control-click. Next, click the Push Design toolbar option.
- To move up the hierarchy, select a port or corresponding net, then click the Pop Design toolbar option.

Finding a Particular Object

To find an object in the currently displayed design, do the following:

1. In the schematic view window, choose Edit > Find by Name. This opens the Find By Name dialog box, which lists the objects contained in the design.
2. In the top text box, select Cells, Ports, or Nets. Objects of the selected type are displayed in the list box, which you can scroll through.
3. Select an object from the list.
4. Click OK.

Formality zooms in on the object, putting it at the center of the view; the object is selected and highlighted in yellow.

Generating a List of Objects

Using the object finder, you can interact with a schematic through dynamic lists of drivers, loads, nets, cells, and ports. Click Find Driver, Find Load, Find X, or Find By Name on the 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. On the toolbar, click Find 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 Load, the GUI takes you directly to the load without bringing up the dialog box. This is also true when you are using Find 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 to quickly 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 the "Back to previous view" toolbar option to step back through views and the "Forward to next view" option to return.

To display the entire design, use the Zoom Full tool. There are four ways to invoke this tool:

- Choose View > Zoom Full.
- Right-click in the schematic window and choose Zoom Full.
- Click the Zoom Full toolbar option.
- Press the letter *f* on the keyboard.

Similarly, to zoom in or zoom out, choose View > Zoom In Tool or Zoom Out Tool, and click where you want the new view to be centered.

To repeatedly zoom into a design, 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, then click.
4. Keep clicking as needed, to zoom in further.

To repeatedly zoom out of a design, follow the same steps used to zoom into a design, except press the minus (–) key to activate the Zoom Out tool.

To quickly zoom in on a small area, 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, 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 Code

You can select an object from any schematic view (or logic cone view) and view its corresponding RTL source data. Source browsing works with all RTL and netlist source files.

To view RTL source code, do the following:

- In the schematic, select a design object, such as a net.
- Right-click and choose View Source.

A window opens with the RTL source code. The selected object is highlighted in red. The previous and next toolbar arrows allow you to cycle through instances of the selected object.

In addition, in any report window, you can right-click a design and then choose View Reference Source or View Implementation Source.

Working With Logic Cones

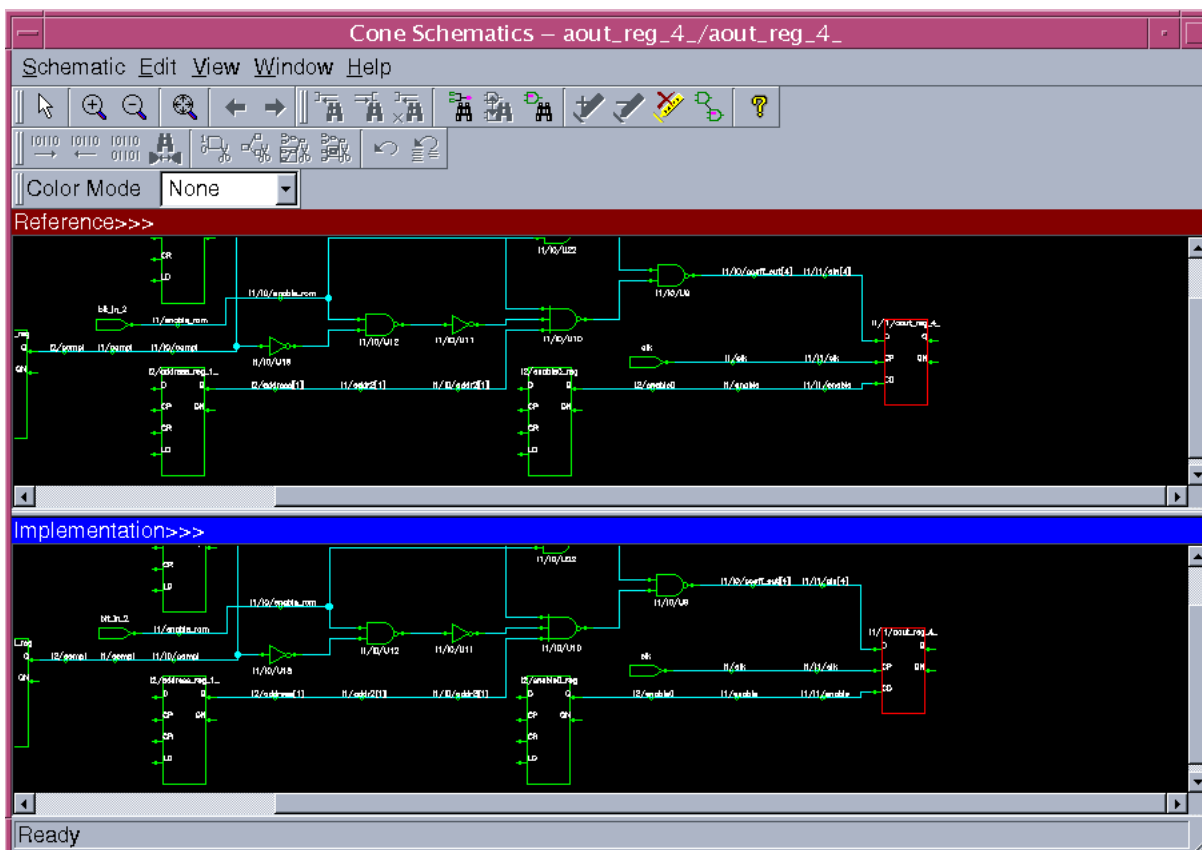
As described in step 2 of [“Debugging by Using Logic Cones” on page 7-11](#), you can open a logic cone view of a failing compare point to help you debug design nonequivalencies.

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 choose View Logic Cones.

A logic cone window appears, as shown in [Figure 7-4](#).

Figure 7-4 Logic Cone View Window



In the Logic Cone View window, the toolbar contains tool buttons that act as shortcuts to some menu selections. The schematic viewer supports the following tool buttons:



Click to select particular sections of the design.



Click to increase the magnification applied to the schematic area by approximately 2X.



Click to decrease the magnification applied to the schematic area by approximately 2X.



Click to redraw the displayed schematic sheet so that all logic is viewable.



Click to view the previous view.



Click to view the next view.



Click to find the driver for the selected net.



Click to find the load on the selected net.



Click to find all net sources of the selected net with logic value X.



Click to find and zoom to the compare point in the schematic.



Click to find the point in the opposing window that matches the point you selected in a double-cone schematic.



Click to display the object finder dialog box to find an object by name in the schematic.



Click to set highlighting on the selected objects.



Click to remove highlighting from the selected objects.



Click to clear highlighting from all objects.



Click to display the name visibility dialog box where you can control the visibility of the objects.



Click to bring up help for the logic cone view window.



Click to show the next pattern values on the schematic.



Click to show the previous pattern values on the schematic.



Click to open the patterns viewer window to show all patterns for the current compare point.



Click to bring up the matching analysis tool for this compare point.



Click to find the point in the opposing window that matches the point you selected in a double-cone schematic.



Click to remove the subcones of the selected net.



Click to isolate the subcones of the selected net.



Click to prune logic and isolate error candidates.



Click to undo the last edit cone operation.



Click to revert to the original cone before editing operations.

In the Logic Cone View window, the two schematics display the logic cones, one for the reference design and one for the implementation design. 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 mouse pointer on it.

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 an unmatched register, do the following:

1. From the cone schematic window, select the unmatched cone input register you want the schematic for.
2. Right-click to open the Context menu.
3. Choose View Logic Cones from this menu.

A single cone schematic window appears containing the cone of logic feeding the unmatched register.

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 by Using Diagnosis” on page 7-9](#).

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.

To better find differences in the full schematic, remove the non-controlling 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.

Sometimes it is useful to look at part of a logic cone. 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, do the following:

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. Click the Undo last cone edit toolbar to restore the subcone.

To isolate a subcone, do the following:

1. Click the net whose logic cone you want to isolate. The selected net is highlighted in yellow.
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.

Working With Failing Patterns

Formality keeps track of the set of input patterns, 0s and 1s only, that cause verification and diagnosis operations to fail. From the logic cone view window, you can simulate the logic by applying single vectors from this set.

By default, Formality uses up to 256 failing patterns to perform diagnosis. When more than one failing compare point exists, Formality selects and uses a set of failing pattern vectors from the patterns of all failing compare points.

A pattern is automatically applied to a displayed logic cone for both the implementation and the reference design. Formality applies the first pattern to both logic areas and displays the state values associated with the logic cones. When the displayed pattern is not helpful for debugging, you can experiment with other failing patterns.

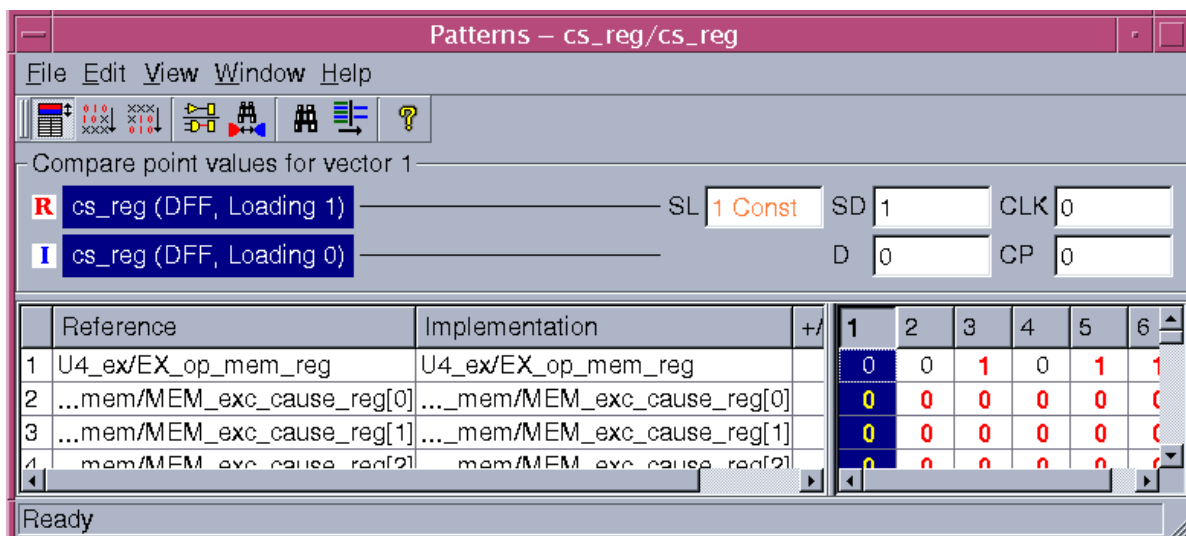
To show a pattern, do the following:

1. In the logic cone view, click the Show Patterns toolbar option.

The failing pattern view window appears. Inputs are those that normally would not make the compare point fail. [Figure 7-5](#) shows an example of the pattern view window.

The reference and implementation columns list the inputs to the failing logic cone. The highlighted column represents the failing pattern currently annotated on the logic cone. Patterns highlighted in red are required to cause a failure.

Figure 7-5 Failing Pattern View Window



Note:

Names shown in blue indicate that a constant 0 has been applied to those inputs. Names shown in orange indicate that a constant 1 has been applied to those inputs. You see the same color indicators in the cone schematic when you set the net coloring mode to view constants.

The toolbars are



Click to show or hide the compare point information.



Click to sort the failing patterns by most-required inputs to cause failure.



Click to sort the failing patterns by least-required inputs to cause failure.



Click to display the logic cone view for the selected input pair.



Click to show the matching tool that provides a report of the matched hierarchical pins and simulation values in the logic cone for the selected failing pattern.



Click to find an input in the reference or implementation list.



Click to filter the reference or implementation list.



Click to show help for the pattern view window.

2. To move forward through the current set of previously failing patterns, click the Next Pattern toolbar option. To move backward, use the Previous Pattern toolbar option. Each command updates the display with a new set of logic values.

Applying a pattern to the logic cone causes the logic area to be redrawn with the new states marked on each pin of each instance. The patterns are applied at the inputs.

You can move through patterns only when the failed verification or completed diagnosis operation has identified more than one failing pattern for the displayed logic cone. If only one pattern exists, the Next Pattern and Previous Pattern options in the View menu are inactive (dimmed).

Saving Failing Patterns

Formality retains the set of failing patterns for the most recently failed verification. You can save these patterns for use in simulation at a later time.

To save the current set of failing patterns, do one of the following:

fm_shell	GUI
Specify:	1. Choose File > Save Failing Patterns.
<code>write_failing_patterns</code> <code>[-diagnosis_patterns]</code> <code>[-verilog] [-replace]</code> <code>file_name</code>	2. In the “File name” box, type the file name. 3. Select Save Diagnosis Patterns Only (to save a diagnosis pattern subset) or Save Patterns in Verilog Format. 4. Click OK.

This command saves failing patterns to an external file (*.fpt). For more information about this command, see the man page.

Note:

Be sure that the verification section of the transcript shows that verification failed.

Running Previously Saved Failing Patterns

To simulate the implementation and reference designs with a set of previously saved failing patterns, do one of the following:

fm_shell	GUI
Specify:	At the Formality prompt, specify:
<code>simulate_patterns file</code>	<code>simulate_patterns file</code>

Note:

The file containing previous failing patterns must be readable by Formality, not saved as a text file. If you save the failing patterns in Verilog format, you cannot use the resulting file with the `simulate_patterns` command.

You can do two things with previously saved failing patterns:

- Simulate your top-level design by applying the entire set of patterns.
- Apply single patterns from the set to the design while viewing a particular logic cone.

When you run a set of previously failing patterns, Formality performs logic simulation on the implementation and reference designs. Using failing patterns as input vectors can help you determine whether changes you have made to the implementation design have fixed a problem found during verification.

After applying the patterns, Formality reports the success of the simulation by refreshing the verification section of the information bar. A report created at this point reflects the state of the completed simulation; it does not reflect the state of the most recently failed verification.

Note:

Passing simulation performed with a set of previously failing patterns is not sufficient to prove design equivalence. For functional equivalence to be proved, the implementation design must pass verification.

8

Technology Library Verification

Formality allows you to verify a reference design with an implementation design in the process described in Chapters 4 through 6. However, you can also compare technology (or cell) libraries, as described in this chapter.

In this chapter it is assumed that you understand Formality concepts and the general process for Formality design verification.

This chapter contains the following sections:

- [Overview](#)
- [Initializing Library Verification](#)
- [Loading the Reference Library](#)
- [Loading the Implementation Library](#)
- [Listing the Cells](#)

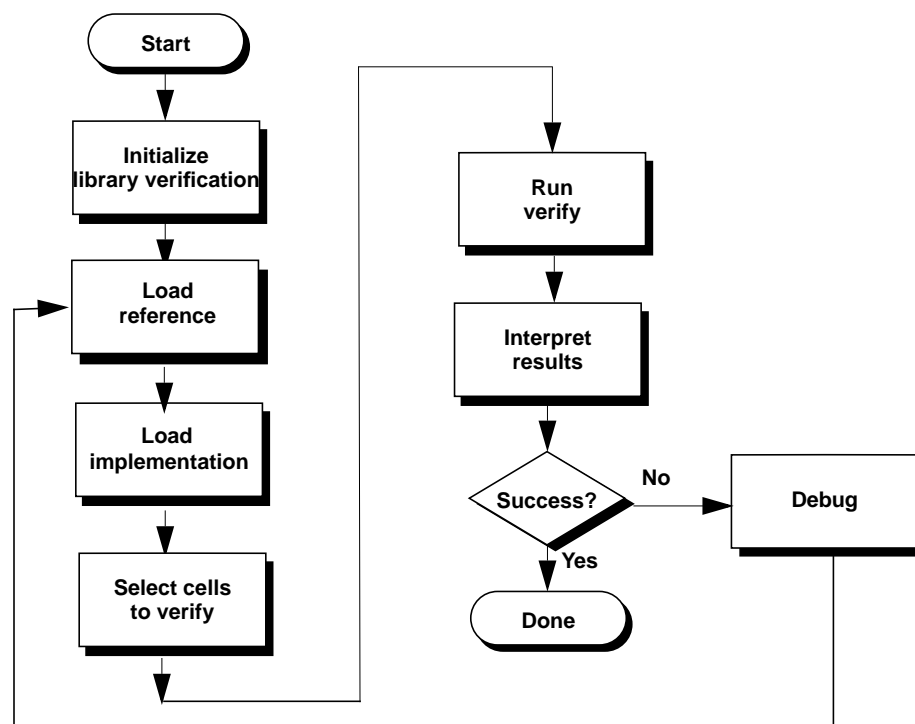
- [Specifying a Customized Cell List](#)
- [Elaborating Library Cells](#)
- [Performing Library Verification](#)
- [Reporting and Interpreting Verification Results](#)
- [Debugging Failed Library Cells](#)

For additional information about verifying libraries, see the *Formality ESP User Guide* in the ESP documentation suite.

Overview

Figure 8-1 shows the general flow for verifying two technology libraries. This chapter describes all the steps in the library verification process.

Figure 8-1 Library Verification Process Flow Overview



During technology library verification, Formality compares all the cells in a reference library to all the cells in an implementation library. The process allows you 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*.

Initializing Library Verification

To verify two libraries, you must first switch to the `library_verification` mode (as opposed to `setup`, `match`, `verify`, or `debug` mode). The GUI closes if it is running. Library verification is a command-line-driven process. Each time you enter (or leave) library verification mode, Formality empties the contents of the `r` and `i` containers in readiness for a new library (or design) verification session.

To enter library verification mode, specify the `library_verification` command, as follows:

```
fm_shell (setup)> library_verification argument
```

The `fm_shell` prompt changes to

```
fm_shell (library_setup)>
```

You can specify one of the following options for *argument*:

- `verilog_verilog`
- `db_db`

- `verilog_db`
- `db_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.

When you set this mode, Formality sets the following variable:

```
set verification_passing_mode "equality"
```

When you exit library verification mode, Formality sets the variable back to its default value, `"consistency"`.

Note:

Unsupported synthesis library formats must be translated by Library Compiler before being read into Formality.

Loading the Reference Library

As with the design verification process described in Chapter 4, you must specify the reference library prior to 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 [-tech] file_list
```

As described in the man pages, the `read_db` and `read_verilog` commands have several options that do not apply to library verification. Use the `read_verilog -tech` option 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 a `designID` argument. The `designID` 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 in the same manner 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 [-tech] file_list
```

Formality loads the reference library into the `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 prior to verification to confirm the matched and unmatched cells.

Specify the following command to return a list of library cells:

```
fm_shell (library_setup)> report_cell_list -reference | \
    -implementation | -verify | -matched | -unmatched | \
    -filter wildcard
```

You must specify one of the following options:

- `-reference`: Returns the cells contained in the reference library.
- `-implementation`: Returns the cells contained in the implementation library.
- `-verify`: Returns 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 8-8](#).
- `-matched`: Returns a list of reference and implementation cells that match by name.
- `-unmatched`: Returns 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, do the following:

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 with the `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:

```
fm_shell (library_setup)> select_cell_list [-file  
filename]\  
[-add cell_names] [-clear] [-remove cell_names]  
cell_names
```

You can use the following options as needed:

- `-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 wildcards 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 `verify` command. You might want to elaborate your library cells prior to verification to apply constraints to specific cells. To elaborate these library cells, run the `elaborate_library_cells` command.

If you do not want to apply constraints to individual library cells, proceed directly to verification. For more information about using the `elaborate_library_cells` command, see the man page.

Performing Library Verification

Proceed to verification after optionally refining your cell list. As with the design verification process, specify the `verify` command:

```
fm_shell (library_setup)> verify
```

Formality performs compare point matching and verification for each cell-pair as described in [Chapter 6, “Compare Point Matching and Verification.”](#) 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 `verify` command has additional options that do not apply to library verification.

After verification, Formality outputs a transcribed summary 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 `verification_passing_mode` and `inversion_push` are set automatically.

```
#-----
# Sets the directories where Formality will search for files
#-----

set search_path "./db ./verilog/cells ./verilog_udp"
#-----
# Sets variables
#-----

set hdlin_unresolved_modules black_box
library_verification VERILOG_DB

#-----
# Reads into container 'r'
#-----
# Read UDP using -tech

read_verilog -r -tech {
UDP_encode.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
ao11A.v
ao11C.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 -v
report_cell_list -m
report_cell_list -u

#-----
# Verifies libraries
#-----

verify

#-----
# Reports on passing and full_addiling cells
#-----

report_status -p
report_status -f
report_status -a

#-----
# Exits
#-----

exit

```

Reporting and Interpreting Verification Results

After verification, use the following command to report the verification results:

```
fm_shell (library_setup)> report_status [-passing]
[-failing] \
```

`[-aborting]`

If you specify no arguments, Formality reports the passing, failing, and aborted cell counts. Use the options as follows:

- `-passing`: Returns a list of all passing cells.
- `-failing`: Returns a list of all failing cells.
- `-aborting`: Returns a list of all aborted cells.

During verification, Formality assigns one of three types of status messages for each library cell-pair it identifies:

Passing

A passing library cell-pair is one in which all its compare points are functionally equivalent.

Failing

A failing library cell-pair is one in which at least one compare point is not functionally equivalent.

Aborted

Verification stops. This occurs most often 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 aborted, and a cell can be aborted if Formality cannot determine whether one of its compare points passes or fails. Aborted points can 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 electrical data only from the last cell verified (which could be a passing cell). This command repopulates Formality with the verification data for the specified cell, which then enables you to debug the cell from within the same 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:

- *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, you can reverify it by customizing the cell list to include only the fixed cell and then specifying the `verify` command.

3. If further investigation is required to fix a failed cell, specify the following command:

```
fm_shell (library_setup)> write_library_debug_scripts \
    [-dir filename]
```

This command generates individual Tcl scripts for each failed cell and places them in the DEBUG directory unless you specify the `-dir` option. The DEBUG directory is located in the current working directory.

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 verification_passing_mode Equality
set verification_inversion_push true
```

```
set search_path "DEBUG"  
read_container -r lib_ref.fsc  
read_container -i lib_impl.fsc  
set_ref r:/*/mux21  
set_impl i:/*/mux21  
verify
```

4. Run one of the Tcl scripts and specify `start_gui` 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:
 - [“Debugging by Using Diagnosis” on page 7-9](#)
 - [“Working With Schematics” on page 7-29](#)
 - [“Working With Logic Cones” on page 7-38](#)
 - [“Working With Failing Patterns” on page 7-44](#)
5. 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.

A

Appendix A - 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 [“Formality Shell Environment” on page 3-6](#).

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 A-1 summarizes the components of the syntax.

Table A-1 Command Components

Component	Description
Command name	<p>If you enter an ambiguous command, Formality attempts to find the correct command.</p> <pre>fm_shell> report_p Error: ambiguous command "report_p" matched two commands: (report_parameters, report_passing_points) (CMD-006)</pre> <p>Formality lists as many as three of the ambiguous commands in its warning. To list the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern.</p> <pre>fm_shell> help report_p*</pre>
Options	<p>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 <i>-libname</i> option.</p> <pre>fm_shell> read_db -libname my_lib</pre> <p>Other options, such as <i>-help</i>, are Boolean options without arguments. You can abbreviate an option name to the shortest unambiguous (unique) string. For example, you can abbreviate <i>-libname</i> to <i>-lib</i>.</p>
Positional arguments	<p>Some Formality commands have positional (or unswitched) arguments. For example, in the <i>set_equivalence</i> command, the <i>object1</i> and <i>object2</i> arguments are positional.</p> <pre>fm_shell> set_equivalence object1 object2</pre>

Using Special Characters

The characters in [Table A-2](#) have special meaning for Tcl in certain contexts.

Table A-2 Special Characters

Character	Meaning
\$	Dereferences a Tcl variable.
()	Groups expressions.
[]	Denotes a nested command.
\	Indicates escape quoting.
" "	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 following command:

```
help
```

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 [“Creating Procedures” on page A-16](#).

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 A-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 the following:

- 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 A-3](#) summarizes several ways for using variables.

Table A-3 Examples of Using Variables

Task	Description
Setting variables	Use the <code>set</code> command to set variables. For compatibility with <code>dc_shell</code> and <code>pt_shell</code> , <code>fm_shell</code> also supports a limited version of the <code>a = b</code> syntax. For example, <code>set x 27</code> or <code>x = 27</code> <code>set y \$x</code> or <code>y = \$x</code>
Removing variables	Use the <code>unset</code> command to remove variables.
Referencing variables	Substitute the value of a variable into a command by dereferencing it with the dollar sign (<code>\$</code>), as in <code>echo \$flag</code> . In some cases, however, you must use the name of a value, such as <code>unset flag</code> , 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 $l1]] $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 conditionally start 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 `elseif` and `else` clauses:

```
if {$x == 0} {  
    echo "Equal"  
} elseif {$x > 0} {  
    echo "Greater"  
} else {  
    echo "Less"  
}
```

In this example, notice that the `else` and `elseif` 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 `elseif` 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
}
```

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]"
}
```

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 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.

Programming Default Values for Arguments

To set up a default value for an argument, you must locate the argument in a sublist that contains two elements: the name of the argument and the default value.

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, 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]"  
}
```

B

Appendix B - Formality Library Support

This appendix provides information about verifying and modifying technology libraries to make them suitable for Formality to perform equivalence checking.

This appendix includes the following sections:

- [Overview](#)
- [Supported Library Formats](#)
- [Updating Synthesis Libraries](#)
- [Library Enhancement and Generation Process](#)
- [Library Verification Process](#)
- [Library Loading Order](#)

Overview

Formality compares two versions of a design for functional equivalence. For this process to succeed, both versions of the design must have functional descriptions for all blocks of logic to be verified. These basic functions are defined by a library. The design libraries must describe the functions of all cells that are used in a design during the synthesis, test, and layout stages of the design flow.

If you are an ASIC vendor, you need to create technology libraries and provide different representations of these libraries to enable your customers to use Synopsys tools. At a minimum, you must provide a library in Synopsys database (.db) format so that customers can synthesize designs, and a Verilog library so that customers can simulate designs with VCS.

There are some important library implementation issues to consider that affect the usability, performance, and capacity of Formality for the end user. This appendix provides guidelines to ensure that the basic requirements of functional modeling are met. It describes the various library formats that Formality supports and tells you how to determine whether any enhancements are required for the library and format you choose. It also provides suggestions to help you enhance your library, guidelines on verifying these enhancements, and information about packaging your library for proper use by customers.

Supported Library Formats

Formality works with the same libraries and design databases as Design Compiler. Users of synthesis-based design flows can adopt Formality with little or no effort. Formality can also accept library information in the form of Verilog simulation libraries, synthesizable RTL, and gate-level netlists.

Synopsys Synthesis Libraries

To read cell definition information contained in Synopsys synthesis libraries (in .db format), use the `read_db` command. Reading information in this format is the fastest way to adopt Formality into your flow. Formality can use the same synthesis libraries that you deliver to customers. However, some enhancements might be required, as determined on a case-by-case basis. For information about such enhancements, see [“Updating Synthesis Libraries” on page B-4](#).

Verilog Simulation Libraries

To read cell definition information contained in Verilog simulation library files, use the `read_simulation_library` or `read_verilog -vcs -y` command, also known as the Verilog library reader. The library reader extracts the pertinent information from the Verilog library to determine the gate-level behavior of the design, and generates a netlist that represents the functionality of the Verilog library cells.

Synthesizable RTL

To read cell definition information from library cells modeled with a synthesizable subset Verilog or VHDL, use the `read_verilog` or `read_vhdl` command. Reading information in this form is typically done for an incremental library. You can model cells by using RTL descriptions of the functional behavior and deliver such a collection of cells to customers either as RTL source code or as .db output from Design Compiler.

Cells modeled with RTL can be inefficient for Formality to process, possibly causing a performance or capacity penalty. To avoid these effects, you can map the RTL to the GTECH technology and save the optimized .db description.

Gate-Level Netlists

As an ASIC vendor, you have the option to provide the functional information in the form of one or more gate-level netlists. You can provide such netlists in .db, .v, or .vhd, format.

Updating Synthesis Libraries

If you choose to use a synthesis library, this section will help you determine whether any enhancements are required to support Formality.

Typically, all vendor libraries have some black boxes in their synthesis libraries. There are two reasons for these black boxes:

- The current modeling capabilities of Library Compiler are insufficient to capture the functional information.

- The modeling capability exists, but the library provider did not model some cells because some Synopsys tools do not use these models and treat them as black boxes anyway.

These black boxes usually need to be updated for Formality because Formality needs functional models to perform verification of any cells or modules that undergo any change in a design flow. However, there might be some cells for which black box descriptions are acceptable.

To determine whether any additional work is required to support Formality, use the flowchart shown in [Figure B-1](#). If you determine that cells need to be updated, use the flowchart shown in [Figure B-2](#) to identify the required enhancements.

Figure B-1 Library Verification Flow

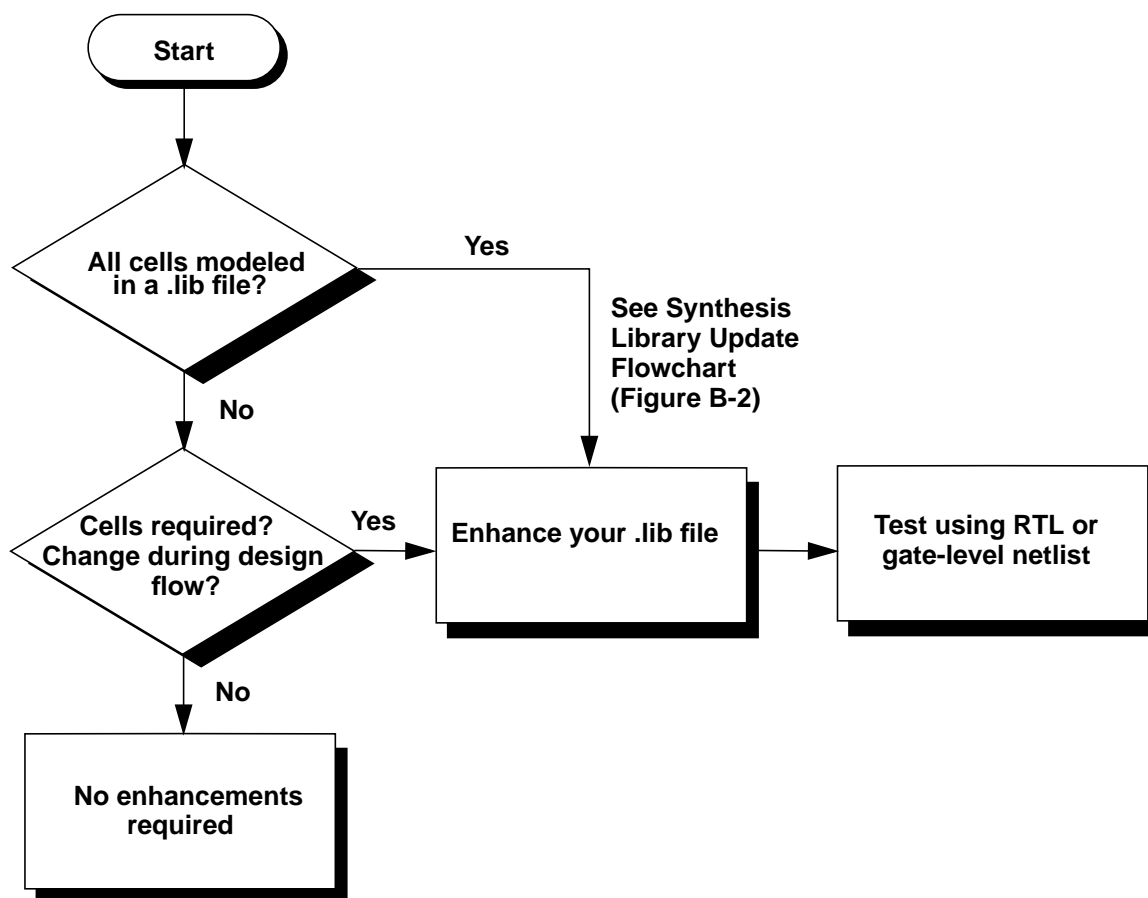
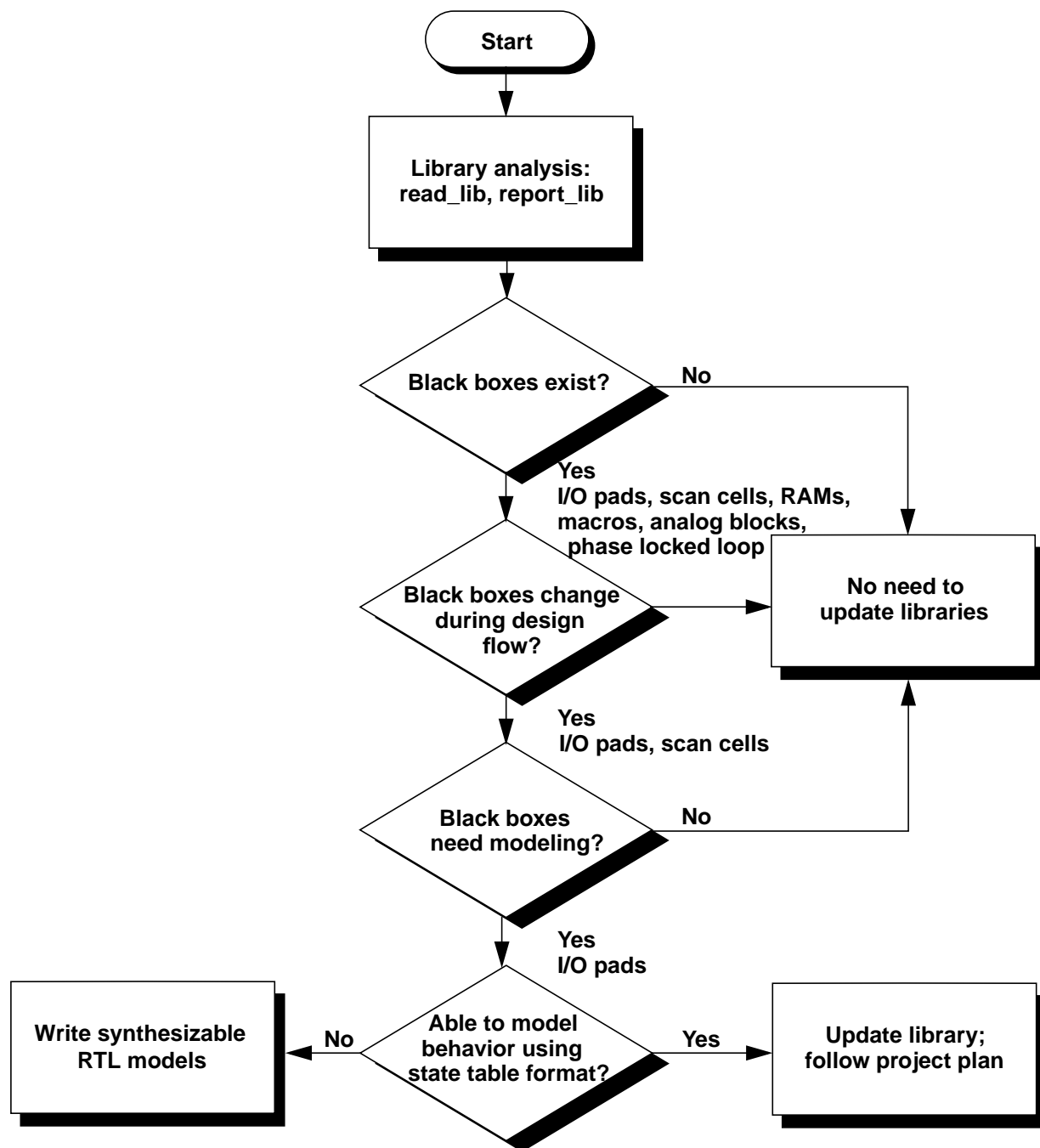


Figure B-2 Synthesis Library Update Flowchart



In [Figure B-2](#), these are the basic steps to take to update a synthesis library:

1. Identify cells that do not need to be modeled to support Formality. A cell that is added by the user during the design flow and remains unchanged for the duration of the flow does not need a functional model. Examples of this type of cell include analog cells such as phase-locked loops, analog I/O port drivers/receivers, hard macros or cores such as controllers and other intellectual property, and highly regular structures such as RAMs and ROMs. Formality can successfully handle these cells as black boxes.
2. Identify black box cells that need to be modeled. Most existing .db libraries have all the important cells modeled already. Important cells to model include registers, latches, scannable versions of registers and latches, all combinational cells, and I/O pad cells. The cells most likely to lack the necessary functional description are the scan elements and I/O pads. Note that Formality handles soft macro cells like any other piece of the design.
3. Identify cells that are missing in the synthesis (.db) library. If there are cells that the tool requires and they are not present in your synthesis .db library, you must go through your regular library development flow to add these cells.

Library Enhancement and Generation Process

The library enhancement and verification process depends on whether you choose to use a synthesis (.db) library or Verilog simulation libraries. Using a synthesis library is the preferred

method. In that case, see [“Using Synthesis Libraries” on page B-9](#). If you choose to use Verilog simulation libraries instead, see [“Using Verilog Libraries” on page B-9](#).

After you generate or update the libraries, you need to verify them for functional accuracy as described in [Chapter 8, “Technology Library Verification.”](#)

Using Synthesis Libraries

If you use a synthesis library with Formality, it is necessary to define the functionality for some cells. Most of the missing cells in a typical synthesis library can be modeled by using Library Compiler, which reads an ASCII description of the cell timing and functional characteristics and generates a .db file that can be read into Design Compiler.

The “state table” format in Library Compiler is useful for modeling many kinds of sequential cells because it provides an intuitive way to describe complex sequential behavior. You can use this format to convert a state table in a databook into a library cell description. The Library Compiler tools reads the truth table for a cell and translates it into a netlist. For more information, see the Library Compiler documentation.

Using Verilog Libraries

The `read_simulation_library` and `read_verilog -vcs -y` commands in Formality, also known as the Verilog library reader, provides a way to directly read in Verilog simulation libraries to support equivalence checking. It supports both gate-level and user-defined primitive (UDP) Verilog models, but not RTL constructs.

To learn more about the `read_simulation_library` and `read_verilog -vcs -y` commands, use the `man` command in Formality. For additional information, see [“Verilog Simulation Data” on page 4-18](#).

The `read_simulation_library` command has an option, `-write_verilog`, that is useful to silicon vendors and others who create technology libraries. As part of the library qualification process, the silicon vendor should verify that Formality can translate the models correctly. The `-write_verilog` option provides a way to check the translation process.

Using the `read_simulation_library` command with the `-write_verilog` option reads in the Verilog simulation library and writes out a Verilog file containing all the models that were read in. The ordering of I/O ports in the Verilog output file is compatible with the original simulation model for all processed modules, allowing the silicon vendor to use this file in the silicon vendor’s existing test bench to verify its correctness.

Note that Verilog written out is not the same as the Verilog read in. The command reads in the Verilog simulation models and converts this information into the internal Formality database. Then it converts this database information into Verilog and writes the results to a file called `lib_name_fm_verilog_out.v`, where `lib_name` is the technology library name specified in the command (or the default name, `TECH_WORK`). Silicon vendor testing of the Verilog output file is the most complete way to verify that Formality can translate the Verilog simulation libraries correctly.

If you are a silicon vendor using this technique, be aware that the simulation models generated by Formality are not complete for simulation purposes. In particular, note the following points:

- The simulation models contain no timing information. Therefore, the results from simulation test benches should not depend upon details of the timing in the original, golden Verilog model. For example, be careful about changing two asynchronous control pins at the same time.
- The simulation models accurately model the response of the cell to input conditions of logic 1, logic 0, and logic Z (where appropriate). This meets the requirements of Formality for functional static verification. However, the models are not intended to exactly reproduce the handling of Xs on inputs or bidirectionals. In some cases, the models can be more pessimistic than the original, golden Verilog model in handling such Xs. ASIC vendors should consider this when selecting a quality assurance (QA) testbench.

These points do not mean that Formality builds incorrect models. Formality builds models that are accurate for functional static verification, not necessarily for simulation.

If you are a silicon vendor and you want to provide the customer with an encrypted version of the Verilog library, use the `-save_binary` option in conjunction with the `-write_verilog` option. Then the command writes the Verilog information in binary format, using the file name *lib_name_fm_verilog_out.bin*. The customer can read in Verilog files saved in this format by using the `read_simulation_library` command. However, the `-write_verilog` option is disabled for this format.

Library Verification Process

After you decide on the library format and make any required enhancements, you can use any one or a combination of the verification flows described in this section. The library verification process is normally a vendor-specific flow that verifies the generated model against a known golden model.

The main goals of the verification process are to ensure that the generated functionality is accurate and complete, and to ensure that the generated model is understood correctly by Formality.

This section recommends generic flows that you can customize for your specific methodology. The following recommendations are guidelines only. After you decide on a flow, you should update your internal test benches so that the process is automated.

The following methods are explained in this appendix:

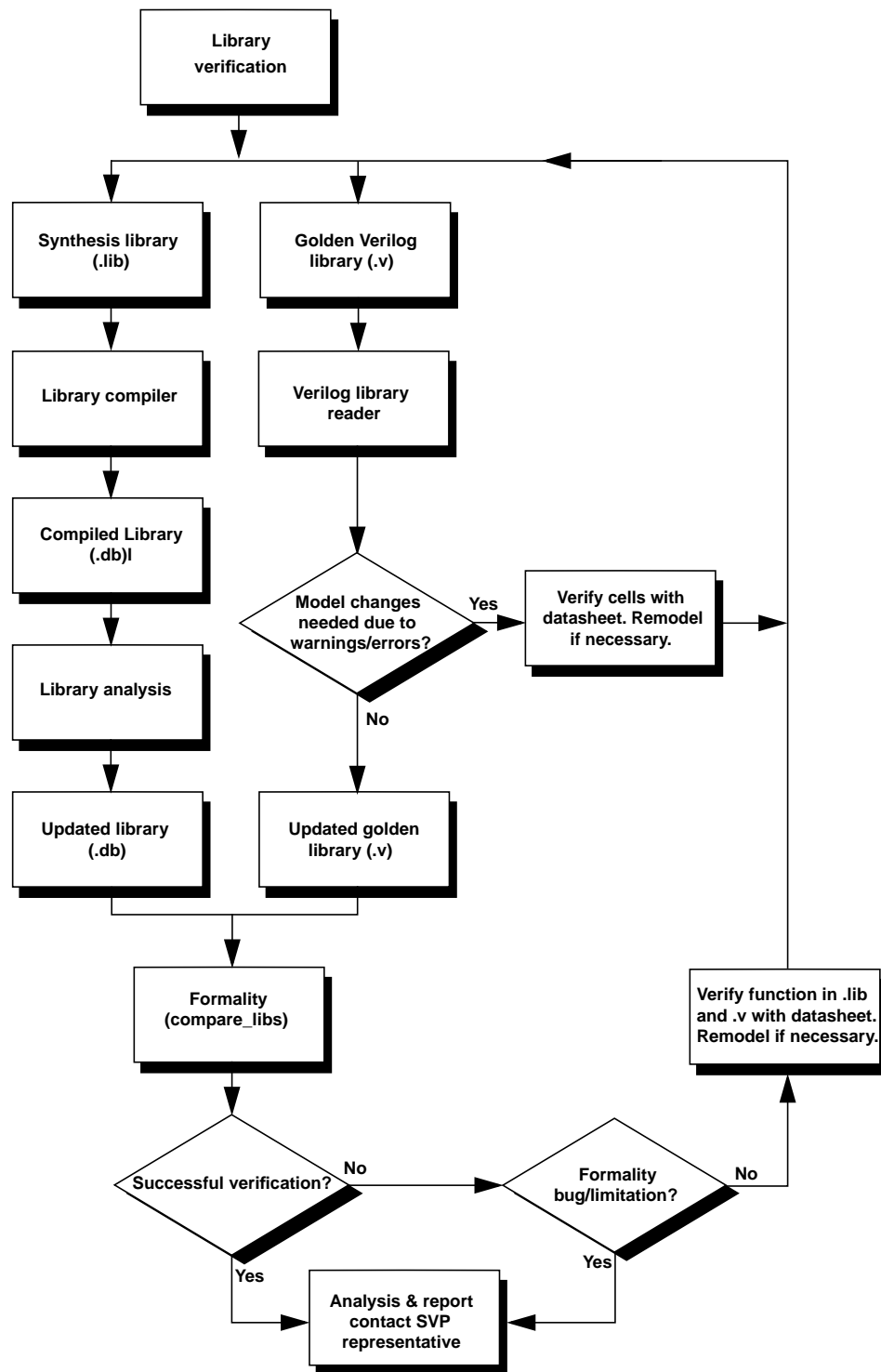
- Verifying the synthesis (.db) libraries against the simulation (.v) libraries. This flow allows you to compare a synthesis model versus an available simulation model. This flow also determines whether Formality interprets a given model correctly. For more information, see “[Verify Synthesis and Simulation Libraries](#).”
- Verifying that the Verilog library reader generates the correct output. This flow compares the simulation results of the golden Verilog libraries with the Verilog libraries produced by Formality.
- Verifying existing designs with the Verilog library reader.

These flows identify any problems with library support for Formality, irrespective of the library format you use.

Verify Synthesis and Simulation Libraries

This section describes a typical flow for functional verification of libraries in Library Compiler (.db) format and Verilog library format (.v). You can use Formality for functional verification of the Verilog library with the synthesis library. The flowchart shown in [Figure B-3](#) describes a typical flow for library verification.

Figure B-3 Synthesis Versus Simulation Library Verification



The process shown in [Figure B-3](#) can be summarized as follows:

1. Generate a synthesis database (.db) from the synthesis library (.lib).
2. Identify any problems in Verilog library (.v) generated by the Verilog library reader.
3. Use Formality to verify that the two libraries are equivalent.

Library Verification Using Formality

Before the verification process, make sure that neither the technology library nor the translated Verilog library have unintentional black boxes.

You can use the library verification mode to compare the two libraries. For information on verifying libraries and the process in which you run the library verification mode, see [Chapter 8](#), “Technology Library Verification.”

Library Loading Order

Formality has the ability to load and manage multiple definitions of a cell, such as synthesis .db files, simulation .db files, and Verilog/VHDL netlists. The order 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, it is recommended that you 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 proper 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, it is recommended that you specify the order in which the libraries are to be loaded.

Augmenting a Synthesis (.db) Library

If you want to use your current synthesis .db database and you do not want to update your .lib database, you can use Verilog simulation libraries to augment the .db library. The recommended flow in this case is

1. Load the existing synthesis .db files.
2. Optionally load the Verilog simulation library .v files to describe the black boxes after step 1.
3. Optionally load the netlist or RTL descriptions for the black boxes after step 2.

Augmenting a Simulation (.v) Library

If you want to use your Verilog simulation libraries and you do not want to use your .lib-based .db database, you first need to verify that the Verilog library reader reads all the cells and Formality interprets

them correctly. You then need to decide whether you want to ship the translated .db or ship the Verilog simulation libraries. The recommended flow in this case is

1. Load the Verilog-based .v files.
2. Optionally load synthesis-based .db files to describe the black boxes after step 1.
3. Optionally load the netlist or RTL descriptions for black boxes after step 2.

C

Appendix C - Reference Lists

This appendix provides reference tables that describe Formality shell variables and commands.

This appendix includes the following sections:

- [Shell Variables](#)
- [Shell Commands](#)

Shell Variables

[Table C-1](#) lists the Tcl variables you can use to condition the environment from the Formality shell.

Table C-1 Shell Variables

Variable	Description
architecture_selection_precedence string	Informs the arithmetic generator which architecture selection method takes precedence. Change only during setup mode.
bus_dimension_separator_style string	Sets the VHDL/Verilog read option that controls the bus dimension separator style used with multidimensional buses.
bus_extraction_style string	Sets the VHDL/Verilog read option that controls the bus extraction style.
bus_naming_style string	Sets the VHDL/Verilog read option that controls the bus naming style.
bus_range_separator_style string	Sets the VHDL/Verilog read option that controls the bus range separator style.
synopsys_auto_setup	Sets certain conservative Formality default variables to values that better conform to Design Compiler behavior. Certain guide commands are also accepted, instead of ignored. Set this variable before reading the automated setup file (.svf).
diagnosis_enable_error_isolation true false	Enables the precise diagnosis engine in Formality that enables you to manually run diagnosis.
diagnosis_enable_find _matching_candidates true false	Determines if the diagnose command identifies matching candidates in the other design.
diagnosis_pattern_limit integer	Sets the maximum number of failing patterns that Formality uses during diagnosis.
distributed_64bit_mode true false	Determines which type of server executable to use on 64-bit capable machines.
distributed_verification_mode string	Allows Formality to perform distributed verification using the verify command. Formality is in distributed mode when this variable is set to enable and there are valid servers in the server list. Default is enabled.

Table C-1 Shell Variables (Continued)

Variable	Description
distributed_work_directory string	Sets the directory that is used for the communication between the master and the servers.
dw_foundation_threshold integer	Defines the input width where Formality switches from generating nbw to wall architectures for multiplier instances. Available only during the setup mode.
enable_multiplier_generation true false	Enables Formality to generate multiplier architectures based on user directives. Available only during setup mode.
gui_report_length_limit integer	Specifies a GUI report size limit.
hdlin_auto_netlist true false	Enables automatic Formality to use the Verilog netlist reader by the <code>read_verilog</code> command.
hdlin_auto_top true false	Enables automatic detection and linking of the top-level design when using the read commands. Unable to select a top-level design if there are multiple candidates.
hdlin_disable_tetramax_define true false	Controls whether the Verilog TetraMAX macro is automatically defined for all <code>read_simulation_library</code> commands.
hdlin_do_inout_port_fixup true false	Controls whether DDX-7 error messages are generated at link time.
hdlin_dwroot string	Specifies the path name to the DesignWare synthesis root.
hdlin_dyn_array_bnd_check string	Controls whether logic is added to check the validity of array indices.
hdlin_enable_verilog_assert true false	Controls whether SystemVerilog ASSERT (and related) statements in Verilog 95 and 2001 source files are treated as errors (default) or ignored.
hdlin_error_on_mismatch_message true false	Works with the <code>hdlin_warn_on_mismatch_message</code> variable to control the severity of messages alerting you to possible mismatches between simulation and synthesis.
hdlin_ignore_builtin true false	Controls whether to ignore or not ignore the <code>built_in</code> directive in VHDL files.

Table C-1 Shell Variables (Continued)

Variable	Description
hdlin_ignore_dc_script true false	Controls whether to ignore or not ignore the <code>dc_script_begin</code> and <code>dc_script_end</code> pragma directives in Verilog or VHDL files.
hdlin_ignore_full_case true false	Controls whether to ignore or not ignore the <code>full_case</code> directive in Verilog files.
hdlin_ignore_label true false	Controls whether to ignore or not ignore the label directive in VHDL files.
hdlin_ignore_label_applies_to true false	Controls whether to ignore or not ignore the <code>label_applies_to</code> directive in Verilog or VHDL files.
hdlin_ignore_map_to_entity true false	Controls whether to ignore or not ignore the <code>map_to_entity</code> directive in VHDL files.
hdlin_ignore_map_to_module true false	Controls whether to ignore or not ignore the <code>map_to_module</code> attribute in Verilog or VHDL files.
hdlin_ignore_map_to_operator true false	Controls whether to ignore or not ignore the <code>map_to_operator</code> directive in Verilog or VHDL files.
hdlin_ignore_parallel_case true false	Controls whether to ignore or not ignore the <code>parallel_case</code> directive in Verilog files.
hdlin_ignore_resolution_method true false	Controls whether to ignore or not ignore the <code>resolution_method</code> directive in VHDL files.
hdlin_ignore_synthesis true false	Controls whether to ignore or not ignore the <code>synthesis_off</code> directive in VHDL files.
hdlin_ignore_translate true false	Controls whether to ignore or not ignore the <code>translate_off</code> and <code>translate_on</code> directives in Verilog files.
hdlin_infer_mux default none all	Specifies how to control the MUX_OP inference for your Verilog or VHDL description.
hdlin_interface_only design	Enables the specified reader to produce interface-only designs.
hdlin_library_directory string	Designates all designs contained within the specified directories as TECHNOLOGY designs.
hdlin_library_enhanced_analysis true false	Enables extra processing of library cells, such as Verilog switch primitives.
hdlin_library_file string	Designates all designs contained within the specified files as TECHNOLOGY files.

Table C-1 Shell Variables (Continued)

Variable	Description
hdlin_library_report_summary true false	Controls whether a warning summary is produced for Verilog library files.
hdlin_multiplier_architecture none csa nbw wall dw_foundation	Defines the architecture generated for multiplier or DW02_multtp instances. You must change this variable before reading a design file to affect results.
hdlin_normalize_blackbox_busses true false	Controls how Formality names black box busses during the link operation.
hdlin_sv_port_name_style complex vector	Controls the naming of ports that are of a complex data type, such as multi-dimensional arrays (MDA), structs, and unions, in a SystemVerilog design.
hdlin_unresolved_modules black_box error	Controls black box creation for Verilog and VHDL descriptions of references that are unresolved during set_top.
hdlin_verilog_95 true false	Controls the default Verilog language interpretation of the Formality RTL reader.
hdlin_verilog_wired_net_interpretation simulation synthesis	Specifies whether non-wire nets are resolved using a simulation or a synthesis interpretation.
hdlin_vhdl_87 true false	Controls the default VHDL language interpretation of the Formality RTL reader.
hdlin_vhdl_auto_file_order true false	Enables the read_vhdl command to automatically order the specified files.
hdlin_vhdl_forgen_inst_naming mode0 mode1 mode2	Specifies the scheme that should be used to name component instances within VHDL for -generate statements.
hdlin_vhdl_fsm_encoding binary 1hot	Specifies the FSM encoding for your VHDL description.
hdlin_vhdl_integer_range_constraint true false	Controls whether Formality adds logic to constrain the value of integer ports and registers to match the integer range constraint specified in the VHDL.
hdlin_vhdl_others_covers_extra_states true false	Controls if others clauses are used to determine how unreachable states are handled.
hdlin_vhdl_presto_naming true false	Controls the generation of operator names inferred from VHDL.

Table C-1 Shell Variables (Continued)

Variable	Description
hdlin_vhdl_presto_shift_div true false	Controls the implementation of a divide by a constant power of two.
hdlin_vhdl_strict_libs true false	Controls whether a strict mapping of VHDL libraries is used during synthesis.
hdlin_vhdl_use_87_concat true false	Controls whether the IEEE Std 1076-1987 style concatenations are used in the IEEE Std 1076-1993 VHDL code.
hdlin_warn_on_mismatch_message message_id_list	Works with the <code>hdlin_error_on_mismatch_message</code> variable to control the types of messages displayed when Formality detects a mismatch between simulation and synthesis.
impl	Indicates the current implementation design.
message_level_mode error warning info	Sets the message severity threshold that Formality uses during verification.
mw_logic0_net string	Specifies the name of the Milkyway ground net.
mw_logic1_net string	Specifies the name of the Milkyway power net.
name_match all none port cell	Controls whether compare point matching uses object names or relies solely on function and topology to match compare points.
name_match_allow_subset_match strict any none	Specifies whether to use subset (token)-based name matching, and if so, what method to use.
name_match_based_on_nets true false	Controls whether compare point matching is based on net names.
name_match_filter_chars char_list	Specifies the characters that should be ignored when the name matching filter is used.
name_match_flattened_hierarchy_ separator_style char	Specifies the separator used in path names Formality creates when it flattens a design during hierarchical verification.
name_match_multibit_register_ reverse_order true false	Specifies to reverse the bit order of the bits of a multibit register.
name_match_net	Specifies whether name matching attempts to reference nets to implementation nets.
name_match_pin_net	Specifies whether name matching attempts to match hierarchical pins to nets

Table C-1 Shell Variables (Continued)

Variable	Description
name_match_use_filter true false	Specifies whether the built-in name matching filter should be used.
ref	Indicates the current reference design.
schematic_expand_logic_cone true auto false	Specifies the schematic view of the logic cone display the internals of techlib cells and DesignWare components.
search_path dirlist	Specifies directories searched for design and library files specified without directory names.
sh_arch	Indicates the current system architecture of the machine you are using.
sh_continue_on_error true false	Controls whether processing continues when errors occur.
sh_enable_page_mode true false	Specifies to display long reports one screen at a time.
sh_product_version value	Specifies the version of the application.
sh_source_uses_search_path true false	Causes the source command to use the search_path variable to search for files.
signature_analysis_allow_net_match	Specifies whether signature analysis utilizes net-based matching methods.
signature_analysis_allow_subset_match	Specifies whether signature analysis uses subset matching methods.
signature_analysis_match_blackbox_input true false	Specifies whether signature analysis attempts to match previously unmatched black box inputs.
signature_analysis_match_blackbox_output true false	Specifies whether signature analysis attempts to match previously unmatched black box outputs.
signature_analysis_match_compare_points true false	Specifies whether signature analysis attempts to match compare points.
signature_analysis_match_datapath true false	Controls whether to automatically attempt to match previously unmatched datapath blocks and their pins during signature analysis.
signature_analysis_match_hierarchy true false	Controls whether to automatically attempt to match previously unmatched hierarchy blocks and their pins during signature analysis.
signature_analysis_match_net true false	Controls whether signature analysis attempts to match reference nets to implementation nets.

Table C-1 Shell Variables (Continued)

Variable	Description
signature_analysis_match_pin_net true false	Controls whether signature analysis attempts to match hierarchical pins to nets.
signature_analysis_match_primary_input true false	Controls whether signature analysis attempts to match previously unmatched primary inputs or not.
signature_analysis_match_primary_output true false	Controls whether signature analysis attempts to match previously unmatched primary outputs or not.
signature_analysis_match_compare_points true false	Controls whether signature analysis attempts to match compare points.
svf_datapath true false	Controls whether Formality processes the transformations found in the automated setup file.
synopsys_root pathname	Sets the value of the \$SYNOPSYS environment variable.
verification_assume_reg_init none conservative liberal liberal0 liberal1	Controls the assumptions that Formality makes about the initial state of registers.
verification_asynch_bypass true false	Controls the verification of registers with asynchronous controls that operate by creating a combinational “bypass” around the register against registers with asynchronous controls that operate only by setting the register state.
verification_auto_loop_break true false	Performs simple loop breaking by cutting the nets.
verification_blackbox_match_mode any identity	Defines how Formality matches comparable black boxes during verification.
verification_clock_gate_hold_mode none low high any	Specifies whether Formality should allow the next state of a flip-flop whose clock is gated to be considered equivalent to that of a flip-flop whose clock is not gated.
verification_constant_prop_mode all user none	Specifies how Formality propagates constants during linking from the top level to a lower level when verifying a lower-level block.
verification_datapath_effort_level low medium high unlimited automatic	Defines the effort level Formality applies during datapath block verification.

Table C-1 Shell Variables (Continued)

Variable	Description
Verification_effort_level	Specifies the effort level to use when verifying two designs.
verification_failing_point_limit value	Specifies the number of failing compare points identified before Formality halts the verification process.
verification_ignore_unmatched_implementation_blackbox_input true false	Defines whether Formality allows unmatched implementation black box input pins in a succeeding verification.
verification_incremental_mode true false	Controls whether the verify command runs verification incrementally.
verification_inversion_push true false	Defines whether Formality attempts to correct failing verifications by checking for cases where data inversion has been moved across register boundaries.
verification_match_undriven_signals true false	Specifies whether Formality should identify and match undriven signals.
verification_merge_duplicated_registers true false	Specifies whether Formality should identify and merge duplicated registers.
verification_parameter_checking true false	Enables parameter checking for technology library register cells and black box cells.
verification_partition_timeout_limit	Causes Formality to terminate processing of any partition after the time-out limit is reached.
verification_passing_mode consistency equality	Sets the verification mode.
verification_progress_report_interval integer	Specifies the interval between progress reports during verification, in minutes.
verification_propagate_const_reg_x true false	Specifies how Formality propagates don't-care states through reconvergent-fanout-dependent constant-disabled registers.
verification_set_undriven_signals X Z 0 1 PI	Specifies how Formality treats undriven nets and pins during verification.
verification_status	Returns the status of the most recent verification, if any.
verification_super_low_effort_first_pass true false	Controls whether Formality makes a preliminary super-low-effort verification attempt on all compare points.

Table C-1 Shell Variables (Continued)

Variable	Description
verification_timeout_limit value	Specifies a wall-clock time limit for how long Formality spends in verification.
verification_use_partial_modeled_cells true false	Directs Formality to use the partial modeled cells behavior as the function of test cells, if no other function is present.
verification_verify_unread_ compare_points true false	Controls whether or not Formality verifies unread compare points.
verification_verify_unread_tech_cell_pins true false	Allows individual unread input pins of tech library cells in the reference design to be matched with the corresponding cells in the implementation design. and treated as primary outputs for verification.

Shell Commands

[Table C-2](#) lists the shell commands that you can invoke from within Formality.

Table C-2 Shell Commands

Command	Description
add_distributed_processors [machine machine...] [-lsf] -nservers -options	Sets up a distributed processing environment and allows distributed equivalence checking verification.
alias [name] [def]	Creates a pseudo-command that expands to one or more words, or lists current alias definitions.
cputime	Returns the CPU time used by the Formality shell.
create_constraint_type type_name designID	Creates a named user-defined (arbitrary) constraint type.
create_container containerID	Creates a new container.
create_cutpoint_blackbox	This command is obsolete with the 2004.06 release. Use the <code>set_cutpoint</code> command instead.
current_container [containerID]	Establishes or reports the current container.

Table C-2 Shell Commands (Continued)

Command	Description
current_design [designID]	Establishes or reports the current design.
debug_library_cell [cell_name]	Runs a diagnosis on the most recent verification.
diagnose [-all failing_points_list] [-effort_level low medium high] [-pattern_limit limit] [-r]	Runs a diagnosis on the most recent verification.
echo [-n] [largument...]	Echoes arguments to standard output.
elaborate_library_cells	Resolves cell references before verifying the library.
error_info	Prints extended information about errors from last command.
exit	Exits the Formality shell.
find_cells [-of_object objectID] [-library_cells -nolibrary_cells] [-type ID_type] [cellID_list]	Lists cells in the current design.
find_designs [object_list] [-passing] [-failing] [-aborted] [-not_verified]	Returns a list of designs.
find_nets [object_list] [-hierarchy] find_nets [-of_object objectID] [-type ID_type]	Returns a list of nets.
find_pins [object_list] [-in] [-out] [-inout] find_pins [-of_object objectID] [-in] [-out] [-inout] [-hierarchy] [-type ID_type]	Returns a list of pins.
find_ports [object_list] [-in] [-out] [-inout] find_ports [-of_object objectID] [-in] [-out] [-inout] [-type ID_type]	Returns a list of ports.
find_references [design_list] [-black_box] [-hierarchy] find_references [-of_object objectID]	Returns a list of designs instantiated within the specified designs.
find_svf_operation [-command] [-status]	Returns a list of automated setup file operation IDs.

Table C-2 Shell Commands (Continued)

Command	Description
<code>get_unix_variable variable_name</code>	Returns the value of a UNIX environment variable.
<code>group cell_list -design_name name [-cell_name name]</code>	Creates a new level of hierarchy.
<code>guide</code>	Puts Formality into guide mode, supporting guidance for all the automated setup file features.
<code>guide_architecture_db [-file <i>filename</i>] [libraries]</code>	Guides Formality on the association of a .db file with an architectural implementation.
<code>guide_architecture_netlist [-file <i>filename</i>] [libraries]</code>	Guides Formality on the association of a netlist file with an architectural implementation.
<code>guide_boundary [[<i>-body body_name</i> operand resource]</code>	Guides Formality in specifying required boundary information needed to verify the netlist represented in the <code>guide_datapath</code> command.
<code>guide_boundary_netlist [[<i>-file netlist_file_name</i> <i>library_name</i>]</code>	Guides Formality in defining the boundary netlist associated with the <code>guide_boundary</code> command.
<code>guide_change_names -design designName [-instance instanceName] [changeBlock]</code>	Guides Formality on changes in the naming of design objects.
<code>guide_constraints -body <i>name</i> [constants] [equivalence]</code>	Guides Formality in defining the boundary netlist associated with the <code>guide_boundary</code> command.
<code>guide_datapath [-design <i>designname</i>] [-datapath <i>datapathname</i>] [-body <i>netlist_design_name</i>]</code>	Guides Formality with identifying a datapath subdesign.
<code>guide_fsm_reencoding -design designName -previous_state_vector prevList -current_state_vector currList -state_reencoding stateList</code>	Guides Formality on the changes in the encoding of the finite-state machine.

Table C-2 *Shell Commands (Continued)*

Command	Description
guide_group -design <i>designName</i> [-instance <i>instanceName</i>] [-cells <i>cellList</i>] -new_design <i>newDesignName</i> -new_instance <i>newInstanceName</i> [groupBlock]	Guides Formality on the hierarchy created around design objects.
guide_inv_push -design <i>designName</i> -register <i>registerName</i>	Guides Formality on a register that has a logical inversion push across it. Register inversions are created during synthesis when phase inversion is turned on.
guide_mc -design <i>designName</i>	Guides Formality on identifying a subdesign generated by Module Compiler.
guide_multiplier -design <i>designName</i> [-instance <i>instanceName</i>] -arch <i>arch</i> -body <i>fileName</i>	Guides Formality on identifying a subdesign as a multiplier with a specific architecture.
guide_reg_constant [-design <i>designName</i>] <i>instanceName</i> <i>constantVal</i>	Guides Formality on the architecture of the constant register information stored in the automated setup file (.svf).
guide_reg_duplication [-design <i>designName</i>] -from <i>fromReg</i> -to <i>toList</i>	Guides Formality on the architecture of the duplicate register information stored in the automated setup file (.svf).
guide_reg_encoding [-design <i>designName</i>] -from <i>fromStyle</i> -to <i>toStyle</i> {bit:bit[:bit]*}	Guides Formality on identifying the registers whose encoding has changed.
guide_reg_merging [-design <i>designName</i>] -from <i>fromReg</i> -to <i>toReg</i>	Guides Formality on the architecture of the merged register information stored in the automated setup file (.svf).

Table C-2 Shell Commands (Continued)

Command	Description
guide_replace [-body <i>name</i>] <i>input</i> <i>output</i> <i>pre_graph</i> <i>post_graph</i>	Guides Formality on the transformation that takes place during datapath optimizations.
guide_retiming [-design <i>designName</i>] -direction <i>direction</i> -libCell <i>cellName</i> -input <i>cellInput</i> -output <i>cellOutput</i> -resetStateDontCare	Guides Formality on an atomic register move over a library cell or net fork. It records the library cell, register names before and after the move, and direction of the move.
guide_retiming_decompose [-design <i>designName</i>] -preName <i>previousName</i> -postName <i>changedName</i> -classChanged	Guides Formality that a synchronous enable register in the reference has been decomposed (changed to a D flip-flop with a feedback MUX) in the implementation.
guide_retiming_finished	Guides Formality that the current group of retiming guidance commands are finished.
guide_retiming_multibit [-design <i>designName</i>] -multiName <i>previousName</i> -splitNames <i>newName</i>	Guides Formality that a multibit register has been split into individual one-bit registers.
guide_transformation -design <i>designName</i> -type <i>type</i> -input <i>inputList</i> -output <i>outputList</i> [-control <i>controlList</i>] [-virtual <i>virtualList</i>] [-pre_resource <i>preResourceList</i>] [-pre_assign <i>preAssignList</i>] [-post_resource <i>post_ResourceList</i>] [-post_assign <i>postAssignList</i>] [-datapath <i>datapathList</i>]	Guides Formality on the structure of datapath transformations.

Table C-2 Shell Commands (Continued)

Command	Description
guide_ungroup -design designName [-instance instanceName] [-cells cellList] [-all] [-small] [-flatten] [-start level] [ungroupBlock	Guides Formality on removing hierarchy from design objects.
guide_uniquify -design designName [uniquifyBlock]	Guides Formality creating unique instances for design objects.
guide_ununiquify -design designName [ununiquifyBlock]	Integrates uniquified instances back to their original designs.
help [command]	Returns information about Formality shell commands.
history [-h] [-r] [args...]	Displays or modifies the commands recorded in the history list.
library_verification mode	Transfers Formality from one of the design verification modes to technology library verification mode.
list_libraries	Lists technology libraries currently loaded.
list_key_bindings	Displays all the key bindings and edit mode of the current shell session.
man [command]	Displays man pages for Formality shell commands.
match [-datapath] [-hierarchy]	Matches compare points.
memory	Reports memory used by the Formality shell.
printenv variable_name	Prints the values of environment variables.
printvar pattern	Prints the values of one or more variables.
proc_args proc_name	Displays the formal parameters of a procedure.
proc_body proc_name	Displays the body of a procedure.
quit	Exits the Formality shell.

Table C-2 Shell Commands (Continued)

Command	Description
read_container [-r -i -container containerID] [-replace] file	Reads a previously written container into the Formality environment.
read_db [-r -i -container containerID] [-libname library_name] [-technology_library] [-merge] [-replace_black_box] file_list	Reads Synopsys .db designs or technology (cell) libraries.
read_ddc [-r -i -container containerID] [-libname library_name] [-technology_library] file_list	Reads designs from a .ddc database.
read_fsm_states file [designID]	Reads finite state machine (FSM) states.
read_milkyway [-r -i -container <i>containerID</i>] [-libname <i>library_name</i>] [-technology_library] version <i>version_number</i> -cell_name <i>mw_cell_name</i> <i>mw_db_path</i>	Reads designs from a Milkyway design library.
read_simulation_library [-r -i -container containerID] [-libname library_name] [-work_library library_name] [-skip_unused_udps] [-write_verilog] [-save_binary] [-merge] [-replace_black_box] [-halt_on_error] file_list	Reads a Verilog simulation library into the Formality environment.
read_sverilog [-container <i>container_name</i>] [-r -i] [-libname <i>libname</i>] [-technology_library] [-work_library <i>libname</i>] [-vcs "VCS options"] [-define <i>defines</i>] [-f <i>filename</i>] [-file_names]	Reads one or more SystemVerilog design files or technology (cell) libraries into the Formality environment.

Table C-2 *Shell Commands (Continued)*

Command	Description
read_verilog [-r -i -container containerID] [-libname library_name] [-work_library library_name] [-netlist] [-technology_library] [-f vcs_option_file] [-define] [-vcs "VCS options"] [-01 -95] file_list	Reads one or more Verilog design files or technology (cell) libraries into the Formality environment.
read_vhdl [-r -i -container containerID] [-libname library_name] [-work_library library_name] [-technology_library] [-87 -93] file_list	Reads one or more VHDL files into the Formality environment.
redirect [-append] file_name { command } string file_name string command	Redirects the output of a command to a file.
remove_black_box [design_list] -all	Undoes the set_black_box command.
remove_compare_rules [designID]	Removes all user-defined compare rules.
remove_constant -all [-type ID_type] instance_path ...	Removes specified or all user-defined constants.
remove_constraint constraint_name	Removes external constraints from the control points of a design.
remove_constraint_type type_name	Removes named user-defined external constraint types created using the create_constraint_type command.
remove_container container_list -all	Removes one or more containers from the Formality environment.
remove_cutpoint object_list [-type objectID_type] -all	Removes the specified or all user-defined cutpoints. Formality accepts this command only in setup mode.
remove_design [-hierarchy] [-shared_lib] design_list	Removes designs from the Formality environment and replaces them with a black box.
remove_design_library library_list -all	Removes one or more design libraries from the Formality environment.

Table C-2 Shell Commands (Continued)

Command	Description
remove_distributed_processors machine machine ... -all	Removes servers for the distributed processing that were added using add_distributed_processors.
remove_dont_verify_point [-type ID_type] [object_1 [object_2] ...] [-all]	Undoes the set_dont_verify_point command for the specified objects.
remove_equivalence [-type ID_type] item1 item2 -all	Removes one or all user-defined equivalences.
remove_guidance [-id <name>] -design <design_name> { <cell_name>:<new_design_name> ...}	Removes all automated setup file (.svf) information that was previously entered through guidance commands or through the set_svf command.
remove_inv_push [-shared_lib] objectIDlist -all	Removes user-defined inversion push.
remove_library library_list -all	Removes one or more technology libraries from the Formality environment.
remove_object objectID [-shared_lib] [-type ID_type]	Removes one or more technology libraries from the Formality environment.
remove_parameters parameter_list [designID_list ...] -all_designs	Removes user-defined, design-specific parameters.
remove_resistive_drivers	Removes pull-ups, pulldowns, and bus holders.
remove_user_match [-all] [-type type] [instance_path]	Un-matches objects previously matched by the set_user_match command.
rename_object objectID -file filename [new_name] [-type ID_type] [-shared_lib] [-reverse] [-container containerID]	Renames one or more design objects.
report_aborted_points [-compare_rule] [-loop] [-hard] [-unverified] [-substring string] [-point_type point_type] [-status status]	Produces information about compare points not verified.
report_architecture [-set_architecture] [-hdlin_multiplier_architecture] -fm_pragma -all ObjectID]	Displays the architecture used to implement the specified ObjectID as well as what caused the selection of that ObjectID.

Table C-2 *Shell Commands (Continued)*

Command	Description
report_black_boxes [design_list -r -i -con containerID] [-all -unresolved -interface_only -empty -set_black_box]	Lists all the black boxes in a design.
report_cell_list [-r -i] [-matched] [-unmatched] [-verify] [-passing] [-failing] [-aborting] [-filter wildcard_pattern]	Reports library cells depending on the option specified. This command is available only in the library verification mode.
report_compare_rules [designID]	Produces information about user-defined compare rules.
report_constants [objectID ...]	Produces information about user-defined constants.
report_constraint [constraint_name] [-long]	Provides information about constraints.
report_constraint_type [type_name] [-long]	Provides information about constraint types.
report_containers	Produces a list of containers.
report_cutpoints	Reports all user-specified cutpoints.
report_design_libraries [item_list]	Produces information about design libraries.
report_designs [item_list]	Produces information about designs.
report_diagnosed_matching_region	Reports information about matching regions for the most recent diagnosis. Matching regions are drivers in the matched design and are identified in the nondiagnosed design.
report_distributed_processors	Reports all servers currently in the list of distributed servers and identifies the distributed working directory.
report_dont_verify_points	Reports compare points disabled by the <code>set_dont_verify_points</code> command.
report_environment	Produces information about user-defined parameters that affect the verification and simulation environment. This command becomes obsolete in a future release. It is recommended that you use the <code>printvar Tcl</code> command instead of using this command.

Table C-2 Shell Commands (Continued)

Command	Description
report_equivalences	Produces information about user-defined equivalences.
report_error_candidates [-match] [-expand]	Produces information about error candidates for the most recent verification.
report_failing_points [-compare_rule] [-matched] [-unmatched] [-substring string] [-point_type point_type] [-status status]	Produces information about compare points that fail verification.
report_fsm [designID -name FSM_name]	Produces information about state encoding.
report_guidance [-datapath [-long]] [-to filename]	Reports to the transcript, or optionally to a file, all automated setup file (.svf) information that was previously entered through guidance commands or through the SVF command.
report_hdlin_mismatches [-summary -verbose] [-reference] [-implementation] [-container container_name]	Reports and summarizes the RTL simulation/synthesis mismatches encountered during design linking.
report_hierarchy [designID]	Produces information about the hierarchy of a design.
report_inv_push [designID]	Produces information about user-defined inversion push.
report_libraries [-short] [library_list]	Produces information about technology libraries.
report_loops [-ref -impl] [-limit N] [-unfold]	Reports loops or portions of loops in either the reference or implementation.
report_matched_points [-compare_rule] [-datapath] [-substring string] [-point_type point_type] [-status status] [-except_status status] [-method matching_method] [-last] [[-type ID_type] objectID...]	Reports all design objects (including compare points) matched by the match command.

Table C-2 Shell Commands (Continued)

Command	Description
report_parameters [design_list]	Produces information about user-defined parameters.
report_passing_points [-compare_rule] [-substring string] [-point_type point_type] [-status status]	Produces information about compare points that passed the most recent verification.
report_status [-pass] [-fail] [-abort]	Reports the current status of verification.
report_svf_operation [-guide] [-message] [-summary] [-operationID_list]	Reports information about specified operations.
report_truth_table [-display fanin] [-fanin {list of signals}] [-constraints {signal=[0/1] }+] [-nb_lins int] [-max_line int] [-max_fanin int]	Generates and prints a truth table for a given signal.
report_unmatched_points [-compare_rule] [-datapath] [-substring string] [-point_type point_type] [-reference] [-implementation] [-status status] [-except_status status] [[-type ID_type] objectID ...]	Reports points that have not been matched.
report_unverified_points [-compare_rule] [-substring <i>string</i>] [-point_type <i>point_type</i>] [-cause <i>cause</i>] [-status <i>status</i>] [-inputs <i>input_type</i>]	Generates information about compare points that you have verified.
report_user_matches [-inverted -noninverted -unknown]	Generates a list of points matched by the <code>set_user_match</code> command.
report_vhdl [-switches] [-name] [configuration] [-entity] [-package]	Produces a list of VHDL configurations, entities, architectures, and associated generics and their default values.
restore_session file	Restores a previously saved Formality session.

Table C-2 Shell Commands (Continued)

Command	Description
save_session [-replace] filename	Saves the current Formality session, including a design matched state.
select_cell_list [-cellNames/wildcards] [-filename] [-add cellNames/wildcards] [-remove cellNames/wildcards] [-clear]	Selects library cells depending on the option specified.
set_architecture [ObjectID] [csa nbw wall]	Instructs Formality to specify the architecture for multiplier ObjectID or for a DesignWare partial product multiplier (DW02_multp) ObjectID. This command is available only during setup mode
set_black_box [designID_list]	Specifies to treat the specified design object as a black box for verification.
set_compare_rule [designID] -from search_pattern -to replace_pattern [-type ID_type] -file filename	Adds a name-matching rule that Formality applies to a design before creating compare points.
set_constant [-type ID_type] instance_path state	Creates a user-defined constant by setting the logic state of a design object to 0 or 1.
set_constraint type_name control_point_list [designID] [-name constraint_name] [-map mapping_list]	Creates an external constraint on a design.
set_cutpoint [-type ID_type] objectID	Specifies that the given hierarchical pin or net is a hard cutpoint that should be verified independently and can be used as a free variable for verification of downstream compare points.
set_direction [-type ID_type] objectID direction [-shared_lib]	Defines port or pin directions.
set_dont_verify_point [-type ID_type] -directly_undriven_output [objectID_list]	Prevents Formality from checking for design equivalence between two objects that constitute a compare point.
set_equivalence [-type ID_type] [-propagate] [-inverted] objectID_1 objectID_2	Declares that two nets or ports have equivalent functions.

Table C-2 Shell Commands (Continued)

Command	Description
set_fsm_encoding encoding_list [designID]	Enumerates FSM state names and their encodings.
set_fsm_state_vector flip-flop_list [designID]	Names state vector flip-flops in an FSM.
set_implementation_design [designID]	Establishes the implementation design.
set_inv_push [-shared_lib] objectIDlist	Adds a sequential object to the list of objects through which Formality transports inverters for verification.
set_parameters [-flatten] [-resolution function] [-retimed] [designID]	Establishes verification parameters for a specific design.
set_power_gating_style [-hld_blocks name] -type type	Sets the <code>power_gating_style</code> attribute on designs or HDL blocks, specifying the kind of retention register cells expected.
set_reference_design [designID]	Establishes the reference design.
set_svf [-append] [-ordered] [-extenstion name] [filedirname]	Specifies the automated setup file (.svf) name (setup verification for Formality).
set_top [-vhdl_arch architecture_name] [designID -auto] [-parameter value]	Sets and links the top-level reference or implementation design.
set_unix_variable variable_name new_value	Sets the value of a UNIX environment variable.
set_user_match [-type ID_type] [-inverted] [-noninverted] object_1 object_2	Forces an object in the reference design to match an object in the implementation design during compare point matching.
setup	Reverts a design from a MATCHED to a SETUP state.
sh [args]	Executes a command in a child process.
simulate_patterns [-no_link] file	Simulates the implementation and reference designs by using previously failing patterns as input vectors.
source [-echo] [-verbose] file	Reads a file and evaluates it as a Tcl script.
start_gui	Invokes the Formality GUI.
stop_gui	Exits the Formality GUI.

Table C-2 Shell Commands (Continued)

Command	Description
test_compare_rule [-designID -r -i] -from search_pattern -to search_pattern -name name [...] [-substring string]	Tests a name matching rule on current unmatched points or user-specified names.
translate_instance_pathname [-type ID_type] pathname	Translates an instance-based path name to a Formality designID or objectID.
unalias pattern	Removes one or more aliases.
undo_match [-all]	Undoes the results of the match command.
ungroup cell_list -all [-prefix prefix_name] [-flatten] [-simple_names]	Removes a level of hierarchy.
uniquify [designID]	Creates unique design names for multiply instantiated designs in hierarchical designs.
verify [designID_1 designID_2] [[-inverted]] [-type ID_type] objectID_1 objectID_2] [[-constant0 constant1] [-type ID_type] objectID] [-restart -incremental] [-level integer]]	Causes Formality to prove or disprove functional equivalence given two designs or two comparable design objects.
which filename_list	Locates a file and displays its path name.
write_container [-r -i -container container_name] [-replace] [-quiet] filename	Saves the information in the current or specified container to a file.
write_failing_patterns [-diagnosis_patterns] [-verilog] [-replace] filename	Saves failing patterns from the most recent diagnosis.

Table C-2 Shell Commands (Continued)

Command	Description
write_hierarchical_verification_script [-replace] [-noconstant] [-noequivalence] [-match type] [-save_directory pathname] [-save_file_limit integer] [-save_time_limit integer] [-level integer] <i>filename</i>	Creates a hierarchical verification Tcl script that may be subsequently executed. You can use it to isolate verification issues to hierarchical subblocks.
write_library_debug_scripts [-dir <i>directory_name</i>]	Debugs failing cells from library verification mode in the standard Formality design format.

Index

Symbols

\$isolate() function 5-64
\$power() function 5-64
\$retain() function 5-64
<< operator 6-32
> operator 3-13
>> operator 3-13

A

aborted compare points
 defined 6-35
 during verification 6-36
add_distributed_processors command 6-25,
 C-10
alias command 3-15, C-10
aliases 3-14
AND resolution function 5-7
application commands A-3
architecture_selection_precedence variable
 C-2
arguments
 designID 4-21
 positional A-4
 programming default values A-17
 varying the number of A-17
arithmetic generator

 creating multiplier architectures 5-67
ASIC libraries 1-16
ASIC verification flow diagram 1-7
asynchronous bypass logic 5-43
asynchronous state-holding loops 5-10
attributes, black box 5-19
automated setup file 1-17
 creating 5-72
 reading 5-73
 reading multiple 5-74
automatic
 creating compare points 1-26
 creating containers 1-30

B

batch mode
 controlling verification 6-33
 interrupting 6-33
 overview 6-31
 preparing for 6-31
 running 6-32
 scripts 6-31
batch script 1-18, 3-3
black box
 attributes 5-19
 controlling 5-16
 creating cutpoints 5-13

- debugging 7-14
- identity check 3-25, 5-17, 5-20
- loading design interfaces 5-17
- locating 7-8
- marking designs 5-18
- overview 5-15
- redefining 5-21
- reporting 5-19
- resolution function 5-7, 5-8
- setting pins and port directions 5-21
- unresolved 5-21
- verifying 5-15, 7-8
- boundary scan 5-39
- built-in commands A-6
- bus holders 5-16
- bus naming
 - changing the style 4-10
 - mapping 1-29
 - VHDL and Verilog design styles 4-9
- bus_dimension_separator_style variable 4-11, C-2
- bus_extraction_style variable C-2
- bus_naming_style variable 4-10, C-2
- bus_range_separator_style variable C-2

C

- cell
 - defined 1-29
 - library verification 8-1
 - listing 3-16
- change_names command 7-28
- clock gating 5-45
- clock tree buffering 5-41
- codes for messages 3-24
- collapse_all_cg_cells value 5-48
- color-coding
 - error candidates 7-33
 - schematic area 7-33
- combinational design changes 5-38
 - boundary scan 5-39

- clock tree buffering 5-41
 - internal scan insertion 5-38
- command alias 3-14
- command log file 3-27
- command names, syntax A-4
- command results, returning A-5
- command shortcuts 3-12
- command-line editing 3-9
- command-line interface (see fm_shell)
- commands
 - add_distributed_processors 6-25, C-10
 - alias 3-15, C-10
 - application, using A-3
 - built-in A-6
 - case-sensitivity 3-7
 - change_names 7-28
 - commenting A-5
 - cputime 3-22, C-10
 - create_constraint_type 5-33, C-10
 - create_container 3-25, 4-30, 5-6, 5-20, 5-21, C-10
 - create_cutpoint_blackbox 7-48, C-10
 - current_container C-10
 - current_design C-11
 - debug_library_cell 8-14, C-11
 - diagnose C-11
 - echo 3-14, 3-29, C-11
 - elaborate_library_cells 8-9, C-11
 - entering commands 3-7
 - error_info C-11
 - exit C-10, C-11
 - find_cells 3-16, C-10, C-11
 - find_designs C-11
 - find_nets 3-16, C-11
 - find_pins 3-16, C-11
 - find_ports 3-16, C-11
 - find_references 3-16, C-11
 - flow control, Tcl A-13
 - fm_shell 1-23, 3-3
 - get_unix_variable C-12
 - getting syntax information 3-21, 3-22

- group C-12
- guide C-10, C-12
- guide_architecture_db C-12
- guide_architecture_netlist C-10, C-12
- guide_change_names C-10, C-12
- guide_datapath C-10, C-12
- guide_fsm_reencoding C-12
- guide_group C-13
- guide_multiplier C-13
- guide_reg_constant C-13
- guide_reg_duplication C-13
- guide_reg_merging C-13, C-14
- guide_transformation C-14
- guide_ungroup C-15
- guide_uniquify C-15
- guide_ununiquify C-15
- help 3-21, A-7, C-15
- help option 3-21, 3-22
- history 3-10, C-15
- interrupting 3-22
- library_verification 8-4, C-15
- line breaks 3-7
- list_key_bindings 3-10, C-15
- list_libraries C-15
- man 3-21, 3-22, C-15
- match 6-5, C-15
- memory C-15
- multiline shell commands 3-7
- nesting A-5
- positional arguments A-4
- printenv C-15
- printvar C-15
- proc_args C-15
- proc_body C-15
- procedures A-7
- puts 3-14
- quit C-15
- read_container 5-88, C-16
- read_db 4-15, 8-5, 8-6, B-3, C-16
- read_ddc 4-24, C-16
- read_fsm_states 5-55, C-16
- read_milkyway 4-23

- read_simulation_library 4-19, B-3, C-16
- read_verilog 4-17, 8-5, 8-6, C-16, C-17
- read_vhdl 4-17, C-17
- recalling 3-12
- redirect 3-13, C-17
- remove_black_box C-17
- remove_compare_rules 6-6, 7-24, C-17
- remove_constant 5-24, C-17
- remove_constraint 5-34, C-17
- remove_constraint_type 5-34, C-17
- remove_container 5-84, C-17
- remove_cutpoint 5-14
- remove_cutpoint_blackbox C-17
- remove_design 5-82, C-17
- remove_design_library 5-83, C-17
- remove_distributed_processors 6-27, C-18
- remove_dont_verify_point 6-23, C-18
- remove_equivalence 5-26, 5-27, C-18
- remove_guidance C-18
- remove_inv_push C-18
- remove_library 5-83, 5-84, C-18
- remove_parameters C-18
- remove_resistive_drivers C-18
- remove_user_match 6-6, 7-18, C-18
- rename_object 7-27, 7-28, C-18
- report_aborted_points 6-34, C-18
- report_architecture 5-71, C-18
- report_black_boxes 5-19, C-19
- report_cell_list 8-7, C-19
- report_compare_rules 7-24, C-19
- report_constants 5-24, C-19
- report_constraint 5-35, C-19
- report_constraint_type 5-35, C-19
- report_containers C-19
- report_cutpoints 5-14, C-19
- report_design_libraries C-19
- report_designs C-19
- report Diagnosed_matching_region C-19
- report_distributed_processors 6-26, C-19
- report_dont_verify_points 6-23, C-19
- report_environment C-19
- report_equivalences C-20

- report_error_candidates C-20
- report_failing_points 6-34, C-20
- report_fsm 5-57, C-20
- report_guidance C-20
- report_hdlin_mismatches C-20
- report_hierarchy C-20
- report_inv_push C-20
- report_libraries C-20
- report_loops 5-11, C-20
- report_matched_points 6-7
- report_parameters C-21
- report_passing_points C-21
- report_power_gating 5-67
- report_status 8-12, C-21
- report_truth_table 8-14, C-21
- report_unmatched_inputs 7-8
- report_unmatched_points 6-6, C-20, C-21
- report_unverified_points C-21
- report_user_matches 7-18, C-21
- restore_session 5-89, C-21
- returning results 3-7
- save_session 5-87, C-22
- select_cell_list 8-8, C-22
- set 3-28
- set_architecture 5-69, C-22
- set_black_box 5-18, C-22
- set_compare_rule 7-20, C-22
- set_constant 5-23, C-22
- set_constraint 5-32, C-22
- set_cutpoint 5-13, C-22
- set_direction C-22
- set_dont_verify_point 6-23, C-22
- set_equivalence 5-25, 5-26, C-22
- set_fsm_encoding 5-56, C-23
- set_fsm_state_vector 5-56, C-23
- set_implementation_design C-23
- set_inv_push 5-51, 5-52, C-23
- set_parameters 5-62, C-23
- set_power_gating_style 5-67, C-23
- set_reference_design C-23
- set_svf 5-72, C-23
- set_top 4-24, C-23
- set_unix_variable C-23
- set_user_match 5-13, 5-41, 7-16, C-23
- setup 6-5, C-23
- sh C-23
- simulate_patterns C-23
- source 3-26, C-23
- special characters A-5
- start_gui C-23
- stop_gui C-23
- Tcl syntax A-1
- test_compare_rule 1-29, 7-23
- translate_instance_pathname C-24
- unalias 3-15, C-24
- undo_match 6-7, C-24
- ungroup C-24
- uniquify C-24
- verify 6-19, 6-21, 6-34, 8-9, C-24
- which C-24
- write_container 5-86, C-24
- write_failing_patterns 7-47, C-24
- write_hierarchical_verification_script 6-30, C-25
- write_library_debug_scripts 8-15, C-25
- commenting commands A-5
- compare points
 - aborted 6-35, 6-36
 - automatic creation of 1-26
 - debugging 7-15
 - debugging unmatched 6-9
 - defining your own 1-27
 - exact-name matching 6-12
 - example 1-27
 - failing 6-35
 - listing user-matched points 7-18
 - mapping names between 1-25, 1-27
 - matched state 6-3
 - matching 6-5
 - flow 6-3
 - related variables 6-12
 - techniques 1-27, 6-10
 - name filtering 6-14
 - name-based matching 6-10

- net-name matching 6-18
- non-name-based matching 6-11
- objects used to create 1-25
- overview 1-25
- passing 6-35
- removing 7-18
- removing from verification set 6-22
- restart matching 6-7
- revert to NOT_RUN state 6-7
- signature analysis 6-16
- status messages 6-35
- topological equivalence 6-15
- undoing match command 6-7
- unmatched, debugging techniques 6-8
- unmatched, reporting 6-6
- unverified 6-35
- verifying single 6-19, 6-21
- compare rules 7-19
 - checking syntax 1-29
 - defining 7-20
 - listing 7-24
 - mapping object names 1-28
 - overview 1-28
 - removing 7-24
 - testing 7-23
- complete verification 1-26
- concepts
 - black boxes 5-15
 - compare points 1-25
 - compare rules 1-28
 - constants 5-22
 - containers 1-29
 - current container 4-31
 - cutpoints 5-12
 - design equivalence 1-31
 - design libraries 1-19
 - design objects 1-29
 - designs 4-8
 - don't care information 5-3
 - external constraints 5-30
 - FSMs 5-53
 - implementation design 1-34
 - libraries 1-19
 - logic cones 1-33
 - LSSD cell 5-62
 - reference design 1-34
 - resolution functions 5-7
 - signature analysis 6-16
 - verification 6-19
- consensus, resolution function 5-7
- consistency, defined 1-12
- console window
 - Formality prompt 3-18, 3-19
 - toolbar 7-31
 - transcript area 3-19
- constants
 - defining 5-22
 - listing 5-24
 - overview 5-22
 - propagating 5-22, 5-37
 - removing 5-24
 - types 5-22
 - user-defined 5-22
- constraint module 5-33
- containers
 - automatic creation 1-30
 - contents 1-29
 - creating 1-30, 4-30
 - current 1-31, 4-31
 - listing 4-31
 - managing 4-29
 - naming 4-30
 - overview 1-29
 - reading data into 1-31
 - removing 5-84
 - restoring 5-88
 - saving 1-18, 1-21, 5-86
 - setting up 4-29
- control flow commands, Tcl A-13
- control statements 6-33
- Control-c interrupt 3-22, 6-29, 6-33
- controlling
 - black boxes 5-16

- verification runtimes 6-24
- copying text
 - from shell window 3-18, 3-19
 - from transcript area 3-19
- coverage percentage 7-33
- cputime command 3-22, C-10
- create_constraint_type command 5-33, C-10
- create_container command 3-25, 4-30, 5-6, 5-20, 5-21, C-10
- create_cutpoint_blackbox command 7-48, C-10
- creating
 - automated setup files 5-72
 - constraint types 5-33
 - containers 1-30, 4-30
 - cutpoints 5-13
 - don't care cells 5-3
 - multiplier architectures 5-67
 - procedures A-16
 - tutorial directories 2-2
- .cshrc 2-2
- current
 - container 1-31, 4-31
 - design 4-31
- current_container command C-10
- current_design command C-11
- cutpoint black boxes 5-13
 - removing 5-14
 - reporting 5-14

D

- data
 - ASIC libraries 1-16
 - automated setup files 1-17
 - containers 1-18, 1-21, 1-29
 - failing patterns 1-21, 7-47, 7-48
 - .fpt files 1-21, 7-47
 - .fsc files 1-18, 1-21, 5-86
 - .fss files 1-18, 1-21, 5-87
 - input file types 1-15

- output file types 1-20
- reading
 - Synopsys database files (.db) 4-9
 - Verilog files 4-9
 - VHDL files 4-9
- removing 5-82
- restoring 5-88, 5-89
- saving 1-18, 1-21, 5-85
- Synopsys database files (.db) 1-16, 4-9
- Verilog files 1-16, 1-17, 4-9
- Verilog simulation library files 1-17
- VHDL files 1-17, 4-9
- .db files 1-16
- .ddc databases
 - designs from 4-22
 - reading designs 4-24
- .ddc files 1-16, 4-8, 4-22
- debug_library_cell command 8-14, C-11
- debugging
 - black boxes 7-14
 - compare points 7-15
 - determining failure causes 7-7
 - eliminating setup possibilities 7-14
 - failing patterns 7-46
 - gathering information 7-4
 - library cells 8-14
 - matching with compare rules 7-19
 - renaming objects 7-27
 - setting compare points to match 7-15
 - strategies 7-4
 - subset matching 7-24
 - unmatched compare points 6-9
 - using diagnosis 7-9
 - using logic cones 7-11
 - working with subcones 7-43
- defining
 - compare points 1-27
 - constants 5-22
 - FSM states 5-54
- deleting
 - containers 5-84

- design libraries 5-83
- designs 5-82
- technology libraries 5-83
- dereferencing variables A-5
- design equivalence 5-25
 - overview 1-31
- design flow, methodology 1-3, 1-4
- design libraries 1-19
 - default name 4-22
 - library ID 1-20
 - reading 1-20
 - removing 5-83
 - restoring 1-18, 5-88, 5-89
 - saving 5-86
 - viewing 1-20
- design management 1-12
- design objects
 - finding 7-34
 - generating lists 7-35
 - overview 1-29
 - renaming 7-26
 - unmatched 1-27
 - used in compare point creation 1-25
- designID, argument 4-21
- designs
 - black box 5-85
 - bus naming, VHDL and Verilog 4-9
 - constants 5-22
 - current 4-31
 - designID 4-21
 - failing patterns, simulating 7-48
 - flattened 5-36
 - hierarchy separator style 5-36
 - implementation 1-34, 1-35
 - inserting cutpoints 5-13
 - linking (with set_top) 4-14
 - locating problem areas 7-4
 - marking as black boxes 5-18
 - overview 4-8
 - propagating constants 5-22
 - reference 1-34, 1-35
 - removing 5-82
 - restoring 5-88
 - retiming 5-59
 - retiming using Design Compiler 5-59
 - retiming with other tools 5-62
 - saving 5-86, 5-87
 - setting the top-level design 4-27
 - setting up 5-1
 - Synopsys database (.db) 1-16, 4-8
 - Verilog 1-16, 1-17, 4-9
 - VHDL 1-17, 4-9
- DesignWare
 - reading in instantiated 4-12
 - root directory 2-8
 - supported components 4-11
- diagnose command C-11
- diagnosis
 - debugging with 7-9
 - interrupting 3-22
 - overview 1-13
- diagnosis_enable_error_isolation variable C-2
- diagnosis_enable_find_matching_candidates variable C-2
- diagnosis_pattern_limit variable C-2
- dimension separator styles 4-9
- directives in VHDL and Verilog ignored/used 4-12
- directory
 - creating for tutorial 2-2
 - work 1-22
- distributed verification process 6-25
 - setting up environment 6-25
 - verifying environment 6-27
- distributed_64bit_mode variable C-2
- distributed_verification_mode variable C-2
- distributed_work_directory variable C-3
- don't care cells
 - creating 5-3
- don't care information
 - overview 5-3
 - verification modes 1-31

dw_foundation_threshold variable C-3

E

- echo command 3-14, 3-29, C-11
- editing, command line 3-9
- elaborate_library_cells command 8-9, C-11
- enable_multilplier_generation variable C-3
- enable_multiplier_generation variable 5-69
- enable_power_gating variable 5-67
- equality, verifying 1-12
- equivalences 5-25
 - defining 5-26
 - listing user-defined 5-27
 - removing 5-26
- error candidates
 - color-coding 7-33
 - coverage percentage 7-33
- error messages 3-23, 3-24
- error_info command C-11
- escape quoting A-5
- exact-name compare point matching 6-12
- examples
 - bus holder 5-16
 - compare point, creation 1-27
 - logic cone 1-34
 - multiply-driven nets 5-8
 - resolution function 5-8
 - schematic view window 7-30
- exit command C-10, C-11
- expressions
 - evaluation, Tcl A-12
 - grouping A-5
- external constraints
 - creating constraint types 5-33
 - overview 5-30
 - removing 5-34
 - removing constraint type 5-34
 - reporting 5-35
 - reporting constraint types 5-35

- setting 5-32
- types 5-31
- user-defined 5-33

F

- failed verification 6-36
- failing compare points, defined 6-35
- failing patterns
 - applying from logic cone window 7-46
 - applying in the logic cone view window 7-38
 - coverage percentage 7-33
 - default number 7-44
 - file 1-21
 - limiting 7-44
 - saving 7-47
 - simulating 7-48
- features, Formality 1-3
- files
 - ASIC libraries 1-16
 - automated setup 1-17
 - batch script 1-18, 6-31
 - black boxing 5-85
 - command log 3-27
 - .ddc 4-8, 4-22
 - failing patterns 1-21, 7-47
 - fm_shell_command log 1-23
 - fm_shell_command.log 1-22
 - formality.log 1-22, 7-4
 - format types supported 1-15
 - .fpt type 1-21, 7-47
 - .fsc type 1-18, 1-21, 5-86
 - .fss type 1-18, 1-21, 5-87
 - input 1-15
 - log 1-23
 - name mapping 1-18
 - output 1-20
 - reading automated setup 5-73
 - removing 5-82
 - search path 3-28
 - session log 3-27

- state files for FSMs 1-17, 5-55
- Synopsys database (.db) 1-16, 4-8
- .synopsys_fm.setup 1-24
- Verilog 1-16, 1-17, 4-9
- Verilog simulation library 1-17
- VHDL 1-17, 4-9
- find_cells command 3-16, C-10, C-11
- find_designs command C-11
- find_nets command 3-16, C-11
- find_pins command 3-16, C-11
- find_ports command 3-16, C-11
- find_references command 3-16, C-11
- find_svf_operation command 5-78, C-11
- finding
 - design objects 7-34
 - lists of design objects 7-35
 - unmatched black boxes 7-8
- finite state machines (FSMs)
 - defining states individually 5-54, 5-56
 - listing state encoding information 5-57
 - overview 5-53
 - preparing for verification 5-54
 - state files 1-17, 5-55
- flattening designs
 - constant propagation 5-37
 - during verification 5-36
 - separator style 5-36
- fm_shell 1-12
 - getting help 3-20
 - listing commands 3-21
 - starting 3-3
- fm_shell command 1-23, 3-3
 - f option 3-3, 3-4, 6-32
 - gui option 3-4
 - no_init option 3-4
 - syntax A-3
 - version option 3-4
 - within GUI 3-18
- fm_shell_command.log file 1-22
- for loops, Tcl flow control A-14

- foreach command, Tcl flow control A-15
- Formality
 - generated file names, controlling 1-23
 - introduction 1-2
 - library support B-1
 - log files 1-22, 7-4
 - supported library formats B-3
- formality_svf directory 5-75
- formality.log file 1-22, 7-4
- format, of files 1-15
- .fpt files 1-21, 7-47
- .fsc files 1-18, 1-21, 5-86
- FSM reencoding 5-54
- .fss files 1-18, 1-21, 5-87

G

- gate-level netlists B-4
- gate-to-gate verification 1-10
- get_unix_variable command C-12
- golden design 1-2, 1-34
- graphical user interface, see GUI 1-11
- group command C-12
- grouping expressions A-5
- grouping words, Tcl commands A-6
- GUI
 - current container 4-31
 - displaying during startup 3-4
 - logic cone view window 7-38
 - overview 1-11, 3-17
 - starting 3-5
- gui_report_lentgh_limit variable C-3
- guide command C-10, C-12
- guide_architecture_db command C-12
- guide_architecture_netlist command C-10, C-12
- guide_change_names command C-10, C-12
- guide_datapath command C-10, C-12
- guide_fsm_reencoding command C-12
- guide_group command C-13

- guide_inv_push variable C-13
- guide_multiplier command C-13
- guide_reg_constant command C-13
- guide_reg_duplication command C-13
- guide_reg_encoding variable C-10, C-13
- guide_reg_merging command C-13, C-14
- guide_retiming variable C-14
- guide_retiming_decompose variable C-14
- guide_retiming_finished variable C-14
- guide_retiming_multibit variable C-14
- guide_transformation command C-14
- guide_ungroup command C-15
- guide_uniquify command C-15
- guide_ununiquify command C-15

H

- hdlin_auto_netlist variable 4-22, C-3
- hdlin_auto_top variable 4-27, C-3
- hdlin_disable_tetramax_define variable C-3
- hdlin_do_inout_port_fixup variable C-3
- hdlin_dwroot variable 4-12, C-3
- hdlin_dyn_array_bnd_check variable C-3
- hdlin_enable_verilog_assert variable C-3
- hdlin_error_on_mismatch_message variable C-2, C-3
- hdlin_ignore_builtin variable C-2, C-3
- hdlin_ignore_dc_script variable C-4
- hdlin_ignore_full_case variable 4-13, C-4
- hdlin_ignore_label variable C-4
- hdlin_ignore_label_applies_to variable C-4
- hdlin_ignore_map_to_entity variable C-4
- hdlin_ignore_map_to_module variable C-4
- hdlin_ignore_map_to_operator variable C-2, C-4
- hdlin_ignore_parallel_case variable 4-13, C-4
- hdlin_ignore_resolution_method variable C-4
- hdlin_ignore_synthesis variable 4-14, C-4
- hdlin_ignore_translate variable 4-14, C-4

- hdlin_infer_multibit variable 5-5
- hdlin_infer_mux variable C-4
- hdlin_interface_only variable C-4
- hdlin_interface_only variables 5-17
- hdlin_library_directory variable 4-17, C-4
- hdlin_library_enhanced_analysis variable C-4
- hdlin_library_file variable 4-17, C-4
- hdlin_library_report_summary variable C-5
- hdlin_multiplier_architecture variable 5-68, C-5
- hdlin_normalize_blackbox_busses variable C-2, C-5
- hdlin_synroot variable C-5
- hdlin_unresolved_modules variable C-5
- hdlin_verilog_95 variable C-5
- hdlin_verilog_wired_net_interpretation variable C-5
- hdlin_vhdl_87 variable C-5
- hdlin_vhdl_auto_file_order variable C-5
- hdlin_vhdl_forgen_inst_naming variable C-5
- hdlin_vhdl_fsm_encoding variable C-5
- hdlin_vhdl_integer_range_constraint variable C-5
- hdlin_vhdl_others_covers_extra_states variable C-5
- hdlin_vhdl_presto_naming variable C-5
- hdlin_vhdl_presto_shift_div variable C-6
- hdlin_vhdl_strict_libs variable 4-21, C-6
- hdlin_vhdl_use_87_concat variable C-6
- hdlin_warn_on_mismatch_message variable C-6
- help
 - command 3-21, A-7, C-15
 - fm_shell commands 3-20
- help, options 3-21
- hierarchical designs 5-35
 - GUI representation 4-21
 - hierarchy separator style 5-36
 - propagating constants 5-37
 - storing 1-30
 - traversing 7-30, 7-33

- hierarchical separator character, defining 5-36
- hierarchical verification 6-29
- history command 3-10, C-15

I

- identity check, black boxes 3-25, 5-17, 5-20
- if command, Tcl flow control A-13
- impl variable C-6
- implementation design 1-2
 - establishing 1-35
 - overview 1-34
 - restoring 5-89
- implementation libraries 8-6
- inconclusive verification status 6-36
- input
 - file types 1-15
 - redirecting during batch jobs 6-32
- inserting
 - cutpoints 5-13
- install directory 2-2
- installation 2-2
- interface
 - command line 1-12
- interfaces
 - GUI 1-11, 3-17
- internal scan insertion 5-38
- interpreting verification results 6-33
- interrupting
 - batch mode verification 6-33
 - diagnosis 3-22
 - fm_shell commands 3-22
 - verification 6-29
- introduction to Formality 1-1
- inversion push 5-50
 - environmental 5-52
 - instance-based 5-51
- invoking
 - fm_shell 2-3, 3-1, 3-3
 - GUI 3-5

- isolating
 - design problems 7-1
 - subcones 7-44

L

- libraries
 - design 1-19
 - DesignWare 4-11
 - functional verification B-13
 - loading order B-15
 - support B-1
 - supported formats B-3
 - Synopsys synthesis B-3
 - synthesis, updating B-4
 - synthesis, using B-9
 - technology 1-19, 1-20
 - verification process B-12
 - Verilog simulation 4-19, B-3
 - Verilog, using B-9
- library files
 - Verilog simulation 1-17
- library ID 1-20
- library verification 8-1
 - debugging process 8-9, 8-14
 - implementation library 8-6
 - initializing 8-4
 - process flow 8-3
 - reference library 8-5
 - reporting library cells 8-7
 - reporting status 8-12
 - sample Tcl script 8-10
 - specifying the cells to verify 8-8
 - status messages 8-13
 - supported formats 8-3
 - truth tables 8-14
 - verifying the libraries 8-9
 - versus design verification 8-3, 8-6, 8-10
- library_verification command 8-4, C-15
- limiting
 - failing patterns 7-44
 - messages 3-25

- linking designs 4-14
- list_key_bindings command 3-10, C-15
- list_libraries command C-15
- listing
 - constants 5-24
 - design objects 7-35
 - fm_shell commands 3-21
 - previously entered commands 3-10
- lists in Tcl A-8
- loading design interfaces, black boxes 5-17
- locating
 - design objects 7-34
 - design problems 7-4
 - lists of design objects 7-35
 - problems 5-22
 - unmatched black boxes 7-8
- log file 3-27
- log files 1-22, 1-23
 - fm_shell_command.log 1-22
 - formality.log 1-22, 7-4
- logic cone view window
 - applying failed patterns 7-38
 - overview 7-38
 - removing non-controlling logic 7-43
 - subcones 7-43
- logic cone, diagnose 7-11
- logic cones 7-45
 - originating point 1-33
 - overview 1-33
 - termination points 1-33
 - viewing 7-38
- loops, Tcl A-14
- LSSD cell, defined 5-62

M

- man command 3-21, 3-22, C-15
- man page overview 3-21
- managing, black boxes 5-16
- mapping names

- buses 1-29, 4-10
- compare points 1-25, 1-27
- compare rules 1-28
- design objects 7-26
- file used for 1-18
- marking designs as black boxes 5-18
- match command 6-5, C-15
- matched verification status 6-3
- matching compare points 6-5
- memory command C-15
- message thresholds, setting 3-25
- message_level_mode variable C-6
- messages
 - codes 3-24
 - error 3-23
 - limiting 3-25
 - setting threshold 3-25
 - severity 3-23
 - syntax 3-23
 - types 3-25
- methodology, design flow 1-3, 1-4
- Milkyway databases
 - allowed formats 4-8
 - reading 4-22
- Milkyway databases, reading 4-23
- multibit support 5-5
- mw_logic0_net variable 4-23, C-6
- mw_logic1_net variable 4-23, C-6

N

- name filtering compare point matching 6-14
- name mapping see mapping names
- name_match variable 6-12, 6-13, C-6
- name_match_allow_subset_match variable 6-12, 7-25, C-6
- name_match_based_on_nets variable 6-12, 6-18, C-6
- name_match_filter_chars variable 6-12, 6-14, 7-24, C-6

- name_match_flattend_hierarchy_separator_style variable C-6
- name_match_flattened_hierarchy_separator_style variable 6-12
- name_match_multibit_register_reverse_order variable 6-12, 6-13, C-6
- name_match_net variable C-6
- name_match_pin_net variable C-6
- name_match_use_filter variable 6-12, 6-14, 7-25, C-7
- name_matched_flattened_hierarchy_separator_style variables 5-36
- nesting commands A-5, A-11
- net-name compare point matching 6-18
- nets
 - constant value 5-22
 - listing 3-16
 - setting to a constant 5-23
 - with multiple drivers 5-6

O

- options, syntax A-4
- OR resolution function 5-7
- output
 - file types 1-20
 - redirecting 3-13
- overview
 - black boxes 5-15
 - compare points 1-25
 - compare rules 1-28
 - constants 5-22
 - containers 1-29
 - design equivalence 1-31
 - design libraries 1-19
 - design objects 1-29
 - designs 4-8
 - don't care information 5-3
 - external constraints 5-30
 - finite state machines (FSMs) 5-53
 - Formality 1-2

- implementation design 1-34
- libraries 1-19
- library support B-2
- library verification 8-3
- logic cones 1-33
- man pages 3-21
- reference design 1-34
- reports 1-21
- resolution functions 5-7
- technology libraries 1-19
- verification 6-19
- overwriting file names 1-23

P

- parameters
 - automatically flattening designs 5-36
 - bus naming 1-29, 4-10
 - constant propagation 5-37
 - design 5-1
 - hierarchical separator style 5-36
 - identity check, black boxes 3-25, 5-20
 - message threshold 3-25
 - multiply-driven net resolution 5-8
 - restoring 5-89
 - saving 5-86, 5-87
- passing compare points 6-35
- path, setting 2-2
- pins
 - defining direction 5-21
 - listing 3-16
- ports
 - constant value 5-22
 - defining direction 5-21
 - direction, black boxes 5-16
 - listing 3-16
 - setting to a constant 5-23
- positional arguments with, commands A-4
- previous session, sourcing 3-27
- printenv command C-15
- printing

- schematics 7-37
- transcript area 3-18
- printvar command C-15
- problem areas, see troubleshooting
- proc_args command C-15
- proc_body command C-15
- procedures
 - creating A-16
 - default A-7
- process flow, general 1-13
- propagating constants 5-22, 5-37
- puts built-in command 3-14

Q

- quick-start tutorial 2-1
- quit command C-15
- quotation marks, using A-6
- quotes, using A-6

R

- read_container command 5-88, C-16
- read_db command 4-15, 8-5, 8-6, B-3, C-16
- read_ddc command 4-24, C-16
- read_fsm_states command 5-55, C-16
- read_milkyway command 4-23
- read_simulation_library command 4-19, B-3, C-16
- read_verilog command 4-17, 8-5, 8-6, C-16, C-17
- read_vhdl command 4-17, C-17
- reading
 - containers 5-88
 - .ddc databases 4-22
 - FSM states 5-55
 - multiple automated setup files 5-74
 - session data 5-89
- redefining
 - black boxes 5-21

- redirect command 3-13, C-17
- redirecting
 - input 6-32
 - output 3-13
- ref variable C-7
- reference design 1-2
 - establishing 1-35
 - overview 1-34
- reference libraries 8-5
- references, design objects 3-16
- regression testing 1-35
- remove_black_box command C-17
- remove_compare_rules command 6-6, 7-24, C-17
- remove_constant command 5-24, C-17
 - type option 5-24
- remove_constraint command 5-34, C-17
- remove_constraint_type command 5-34, C-17
- remove_container command 5-84, C-17
- remove_cutpoint command 5-14
- remove_cutpoint_blackbox command C-17
- remove_design command 5-82, C-17
- remove_design_library command 5-83, C-17
- remove_distributed_processors command 6-27, C-18
- remove_dont_verify_point command 6-23, C-18
- remove_equivalence command 5-26, 5-27, C-18
- remove_guidance command C-18
- remove_inv_push command C-18
- remove_library command 5-83, 5-84, C-18
- remove_parameters command C-18
- remove_resistive_drivers command C-18
- remove_user_match command 6-6, 7-18, C-18
- removing
 - containers 5-84
 - design libraries 5-83
 - designs 5-82
 - subcones 7-43

- technology libraries 5-83
- rename_object command 7-27, 7-28, C-18
- renaming design objects 7-26
- report_aborted_points command 6-34, C-18
- report_architecture command 5-71, C-18
- report_black_boxes command 5-19, C-19
- report_cell_list command 8-7, C-19
- report_compare_rules command 7-24, C-19
- report_constants command 5-24, C-19
- report_constraint command 5-35, C-19
- report_constraint_type command 5-35, C-19
- report_containers command C-19
- report_cutpoints command 5-14, C-19
- report_design_libraries command C-19
- report_designs command C-19
- report_diagnosed_matching_region command C-19
- report_distributed_processors command 6-26, C-19
- report_dont_verify_points command 6-23, C-19
- report_environment command C-19
- report_equivalences command C-20
- report_error_candidates command C-20
- report_failing_points command 6-34, C-20
- report_fsm command 5-57, C-20
- report_guidance command 5-79, C-20
- report_hdlin_mismatches command C-20
- report_hierarchy command C-20
- report_inv_push command C-20
- report_libraries command C-20
- report_loops command 5-11, C-20
- report_parameters command C-21
- report_passing_points command C-21
- report_power_gating command 5-67
- report_status command 8-12, C-21
- report_svf_operation command 5-79
- report_truth_table command 8-14, C-21
- report_unmatched_inputs command 7-8
- report_unmatched_points command 6-6, 6-7, C-20, C-21
- report_unverified_points command C-21
- report_user_matches command 7-18, C-21
- reporting
 - black boxes 7-8
- reporting black boxes 5-19
- reports 7-4
 - compare rules 7-24
 - constants, user-defined 5-24
 - containers 4-31
 - equivalences, user-defined 5-27
 - finite state machine (FSMs) information 5-57
 - library cells 8-7
 - library verification results 8-12
 - overview 1-13, 1-21
 - truth tables 8-14
- requirements, Formality 1-8
- resolution functions
 - multiply-driven nets 5-7
 - overview 5-7
- resolving
 - multiply-driven nets 5-7
 - nets with multiple drivers 5-6
- restore_session command 5-89, C-21
- restoring
 - data 1-18, 5-88, 5-89
 - parameters 5-89
 - session 1-18, 5-89
- results 6-36
- retimed designs, working with 5-59
- retiming designs
 - using Design Compiler 5-59
 - using other tools 5-62
 - verifying with Formality 5-61
- returning shell command results A-5
- rigid quoting A-5
- root directory 1-24
- RTL B-4
 - designs 1-9

RTL-to-gates verification 1-9

RTL-to-RTL verification 1-9

S

save_session command 5-87, C-22

saving

- containers 1-18, 1-21, 5-86

- data 5-85

- failing patterns 7-47

- parameters 5-86, 5-87

- session 1-21, 5-87

schematic view window

- example 7-30

- zooming 7-36

schematic_expand_logic_cone variable C-7

schematics

- printing 7-37

- viewing logic cones 7-38

script file

- batch jobs 6-31

- sourcing 3-26, 6-32

- tasks 3-26

search path

- examining 3-29

- files 3-28

search_path variable 3-28, C-7

select_cell_list command 8-8, C-22

separating list items, Tcl commands A-9

separator character, bus names 4-11, 4-12

sequential design changes 5-42

- asynchronous bypass logic 5-43

- clock gating 5-45

- inversion push 5-50

session data, restoring 5-89

set command 3-28

set_architecture command 5-69, C-22

set_black_box command 5-18, C-22

set_compare_rule command 7-20, C-22

set_constant command 5-23, C-22

- type option 5-23

set_constraint command 5-32, C-22

set_cutpoint command 5-13, C-22

set_direction command C-22

set_dont_verify –directly_undriven_output
command 4-5

set_dont_verify_point command 6-23, C-22

set_equivalence command 5-25, 5-26, C-22

set_fsm_encoding command 5-56, C-23

set_fsm_state_vector command 5-56, C-23

set_implementation_design command C-23

set_inv_push command 5-51, 5-52, C-23

set_parameters command 5-62, C-23

set_power_gating_style command 5-67, C-23

set_reference_design command C-23

set_svf command 5-72, C-23

set_top command 4-24, C-23

- conditions 4-25

set_unix_variable command C-23

set_user_match command 5-13, 5-41, 7-16,
C-23

setting

- implementation design 1-35

- message thresholds 3-25

- pins and port directions 5-21

- reference design 1-35

- top-level design 4-27

setup command 6-5, C-23

setup files 1-24

Setup-free Flow 4-5

severity rating for messages 3-23, 3-25

sh command C-23

sh_arch variable C-7

sh_continue_on_error variable C-7

sh_enable_line_editing variable 3-10

sh_enable_page_mode variable C-7

sh_line_editing_mode variable 3-10

sh_product_version variable C-7

- sh_source_uses_search_path variable 3-28, C-7
- shell interface, starting 3-3
- shell window
 - copying text 3-18, 3-19
- signature analysis 6-16
- signature_analysis_allow_net_match variable C-2, C-7
- signature_analysis_allow_subset_match variable C-7
- signature_analysis_match_blackbox_input variable C-7
- signature_analysis_match_blackbox_output variable C-7
- signature_analysis_match_compare_point variable C-7
- signature_analysis_match_compare_points variable C-8
- signature_analysis_match_datapath variable C-7
- signature_analysis_match_hierarchy variable C-7
- signature_analysis_match_net variable C-7
- signature_analysis_match_primary_input variable 6-12, C-2, C-8
- signature_analysis_match_primary_output variable 6-12, 6-17, C-8
- signature_analysis_matching variable 6-12, 6-17
- simulate_patterns command C-23
- simulating previously failing patterns 7-48
- single-state holding elements 5-62
- source command C-23
 - batch jobs 6-32
 - echo option 3-26
 - file search exception 3-28
 - syntax 3-26
 - verbose option 3-26
- sourcing
 - previous sessions 3-27
 - script files 3-26, 6-32

- special characters
 - Tcl A-5
- start_gui command C-23
- starting
 - fm_shell 2-3, 3-1, 3-3
 - GUI 3-5
- state files for FSMs 1-17, 5-55
- stop_gui command C-23
- subcones 7-43, 7-44
- succeeded verification 6-36
- supported library formats B-3
- svf_datapath variable C-8
- svf_ignore_unqualified_fsm_information variable 5-54
- switch command, Tcl flow control A-16
- Synopsys database files (.db) 1-16, 4-8, 4-9
- Synopsys synthesis libraries B-3
- synopsys_auto_setup mode 2-6, 4-3
- synopsys_auto_setup variable C-2
- .synopsys_fm.setup file 1-24
- synopsys_root variable C-8
- syntax
 - fm_shell commands A-4
 - procedures A-7
- synthesis libraries, updating B-4

T

Tcl

- arguments, varying the number of A-17
- break command A-15
- commands that support lists A-8
- continue command A-15
- control flow commands A-13
- expression evaluation A-12
- for loops A-14
- foreach command A-15
- grouping words A-6
- lists A-8
- nesting commands A-11

- overview A-1
- quoting values A-6
- separating list items 3-8, A-9
- special characters A-5
- switch command A-16
- user-defined variables A-10
- while command A-14
- Tcl variables
 - distributed_64bit_mode 6-27
 - distributed_verification_mode 6-27
 - distributed_work_directory 6-25
 - sh_source_uses_search_path 3-28
- TECH_WORK library 4-15
- technology libraries 1-19
 - default name 4-15
 - library ID 1-20
 - reading 1-20, 4-7
 - removing 5-83
 - restoring 1-18, 5-88, 5-89
 - saving 5-86, 5-87
 - shared 1-19, 4-30, 4-31
 - unshared 1-19, 4-30
 - verifying 8-1
 - viewing 1-20
- terminating loops, Tcl A-15
- test_compare_rule command 1-29, 7-23
- thresholds, message level 3-25
- toolbar, console window 7-31
- top-level designs 4-14
- topological equivalence 6-15
- transcript area
 - copying text 3-19
 - printing 3-18
- transcript window 7-4
- translate_instance_pathname command C-24
- traversing hierarchical designs 7-30, 7-33
- troubleshooting
 - asynchronous state-holding loops 5-10
 - black boxes 7-8, 7-14
 - determining failure cause 7-4, 7-7
 - eliminating setup possibilities 7-14
 - extraneous bus drivers 5-16
 - failed verifications 7-1
 - failing patterns 7-46
 - failing patterns, simulating 7-48
 - gathering information 7-4
 - incomplete verification 7-4
 - locating problems 5-22
 - logic cones, viewing 1-33
 - matching with compare rules 7-19
 - problem areas, locating 7-4
 - renaming objects 7-27
 - setting compare points to match 7-15
 - subset matching 7-24
 - unmatched compare points 6-9
 - using diagnosis 7-9
 - using logic cones 7-11
 - working with subcones 7-43
- truth table 8-14
- tutorial 2-1
 - directory 2-3
- tutorial directories
 - creating 2-2

U

- unalias command 3-15, C-24
- undo_match command 6-7, C-24
- ungroup command C-24
- uniquify command C-24
- unmatched compare points 6-6
- unverified compare points, defined 6-35
- updating, synthesis libraries B-4
- usage model diagram 1-5
- user-defined
 - compare points 1-27
 - constants 5-22
 - removing 5-24
 - reporting 5-24
 - equivalences 5-25
 - variables, Tcl A-10
- user-defined equivalence, removing 5-26

user-specified file names 1-23
/usr/synopsys root directory 1-24

V

values, quoting A-6

variables

- architecture_selection_precedence C-2
- bus_dimension_separator_style 4-11, C-2
- bus_extraction_style C-2
- bus_naming_style 4-10, C-2
- bus_range_separator_style C-2
- dereferencing A-5
- diagnosis_enable_error_isolation C-2
- diagnosis_enable_find_matching_candidates C-2
- diagnosis_pattern_limit C-2
- distributed_64bit_mode C-2
- distributed_verification_mode C-2
- distributed_work_directory C-3
- dw_foundation_threshold C-3
- enable_multiplier_generation 5-69, C-3
- enable_power_gating 5-67
- get_unix_variable C-12
- gui_report_length_limit C-3
- guide_inv_push C-13
- guide_reg_encoding C-10, C-13
- guide_retiming C-14
- guide_retiming_decompose C-14
- guide_retiming_finished C-14
- guide_retiming_multibit C-14
- hdlin_auto_netlist 4-22, C-3
- hdlin_auto_top 4-27, C-3
- hdlin_disable_tetramax_define C-3
- hdlin_do_inout_port_fixup C-3
- hdlin_dwroot 4-12, C-3
- hdlin_dyn_array_bnd_check C-3
- hdlin_enable_verilog_assert C-3
- hdlin_error_on_mismatch_message C-2, C-3
- hdlin_ignore_builtin C-2, C-3
- hdlin_ignore_dc_script C-4

- hdlin_ignore_full_case 4-13, C-4
- hdlin_ignore_label C-4
- hdlin_ignore_label_applies_to C-4
- hdlin_ignore_map_to_entity C-4
- hdlin_ignore_map_to_module C-4
- hdlin_ignore_map_to_operator C-2, C-4
- hdlin_ignore_parallel_case 4-13, C-4
- hdlin_ignore_resolution_method C-4
- hdlin_ignore_synthesis 4-14, C-4
- hdlin_ignore_translate 4-14, C-4
- hdlin_infer_multibit 5-5
- hdlin_infer_mux C-4
- hdlin_interface_only 5-17, C-4
- hdlin_library_directory 4-17, C-4
- hdlin_library_enhanced_analysis C-4
- hdlin_library_file 4-17, C-4
- hdlin_library_report_summar C-5
- hdlin_multiplier architecture 5-68
- hdlin_multiplier_architecture C-5
- hdlin_normalize_blackbox_busses C-2, C-5
- hdlin_synroot C-5
- hdlin_unresolved_modules C-5
- hdlin_verilog_95 C-5
- hdlin_verilog_wired_net_interpretation C-5
- hdlin_vhdl_87 C-5
- hdlin_vhdl_auto_file_order C-5
- hdlin_vhdl_forgen_inst_naming C-5
- hdlin_vhdl_fsm_encoding C-5
- hdlin_vhdl_integer_range_constraint C-5
- hdlin_vhdl_others_covers_extra_states C-5
- hdlin_vhdl_presto_naming C-5
- hdlin_vhdl_presto_shift_div C-6
- hdlin_vhdl_strict_libs 4-21, C-6
- hdlin_vhdl_use_87_concat C-6
- hdlin_warn_on_mismatch_message C-6
- impl C-6
- message_level_mode C-6
- mw_logic0_net 4-23, C-6
- mw_logic1_net 4-23, C-6
- name_match 6-12, 6-13, C-6
- name_match_allow_subset_match 6-12, 7-25, C-6

name_match_based_on_nets 6-12, 6-18, C-6
 name_match_filter_chars 6-12, 6-14, 7-24, C-6
 name_match_flattened_hierarchy_separator_style 6-12, C-6
 name_match_multibit_register_reverse_order 6-12, 6-13, C-6
 name_match_net C-6
 name_match_pin_net C-6
 name_match_use_filter 6-14, 7-25, C-7
 name_match_user_filter 6-12
 name_matched_flattened_hierarchy_separator_style 5-36
 ref C-7
 schematic_expand_logic_cone C-7
 search_path 3-28, C-7
 sh_arch C-7
 sh_continue_on_error C-7
 sh_enable_line_editing 3-10
 sh_enable_page_mode C-7
 sh_line_editing_mode 3-10
 sh_product_version C-7
 sh_source_uses_search_path 3-28, C-7
 signature_analysis_allow_net_match C-2, C-7
 signature_analysis_allow_subset_match C-7
 signature_analysis_match_blackbox_input C-7
 signature_analysis_match_blackbox_output C-7
 signature_analysis_match_compare_point C-7
 signature_analysis_match_compare_points C-8
 signature_analysis_match_datapath C-7
 signature_analysis_match_hierarchy C-7
 signature_analysis_match_net C-7
 signature_analysis_match_primary_input 6-12, C-2, C-8
 signature_analysis_match_primary_output 6-12, 6-17, C-8

signature_analysis_matching 6-12, 6-17
 svf_datapath C-8
 svf_ignore_unqualified_fsm_information 5-54
 synopsys_root C-8
 verification_assume_reg_init C-8
 verification_async_bypass 5-44, C-8
 verification_auto_loop_break C-8
 verification_blackbox_match_mode 6-12, C-8
 verification_clock_gate_hold_mode 5-47, 5-50, C-8
 verification_constant_prop_mode 5-37, C-8
 verification_datapath_effort_level C-8
 verification_effort_level C-9
 verification_failing_point_limit C-9
 verification_ignore_unmatched_implementation_blackbox_input C-9
 verification_incremental_mode 6-20, C-9
 verification_inversion_push 5-53, C-9
 verification_match_undriven_signals C-9
 verification_merge_duplicated_registers C-9
 verification_parameter_checking C-9
 verification_partition_timeout_limit C-9
 verification_passing_mode C-9
 verification_progress_report_interval C-9
 verification_propagate_const_reg_x C-9
 verification_set_undriven_signals C-9
 verification_status C-9
 verification_super_low_effort_first_pass 6-20, C-9
 verification_timeout_limit 6-24, C-10
 verification_use_partial_modeled_cells C-10
 verification_verify_unread_compare_points C-10
 verification_verify_unread_tech_cells C-10
 verification
 batch mode 6-31
 boundary scan 5-39
 cell libraries 8-1
 clock tree buffering 5-41
 compare point matching 6-10

- complete 1-26
- consistency 1-12, 1-31
- constant propagation 5-22, 5-37
- controlling 6-33
- controlling runtimes 6-24
- CPU time 1-35
- debugging failed 7-1
- design equality 1-12, 1-31, 5-4
- establishing environment 5-1
- failed 7-7
- failed status 6-36
- finite state machines 5-54
- flattened designs 5-36
- gates-to-gates 1-10
- getting ready 4-1
- hierarchical 6-29
- hierarchical designs 5-35, 5-36
- inconclusive status 6-36
- inserting cutpoints 5-12
- internal scan insertion 5-38
- interrupting 6-29
- library process B-12
- LSSD cells 5-62
- mode 1-32
- overview 1-12, 6-19
- performing 6-1
- problem areas, locating 7-4
- removing matched compare points 6-22
- reporting progress 6-33
- reporting results 6-34
- restoring the state of 5-89
- results 6-33
- RTL-to-gate 1-9
- RTL-to-RTL 1-9
- sequential design changes 5-42
- setting external constraints 5-30
- single compare point 6-19, 6-21
- starting 6-19
- status messages 6-35, 6-36
- succeeded status 6-36
- technology libraries 8-1
- transformed designs 5-38

- troubleshooting 7-4
- using diagnosis 7-9
- using logic cones 7-11
- viewing results 6-35
- verification_assume_reg_init variable C-8
- verification_asynch_bypass variable 5-44, C-8
- verification_auto_loop_break variable C-8
- verification_blackbox_match_mode variable 6-12, C-8
- verification_clock_gate_hold_mode variable 5-47, 5-50, C-8
- verification_constant_prop_mode variable 5-37, C-8
- verification_datapath_effort_level variable C-8
- verification_effort_level variable C-9
- verification_failing_point_limit variable C-9
- verification_ignore_unmatched_implementation_blackbox_input variable C-9
- verification_incremental_mode variable 6-20, C-9
- verification_inversion_push variable 5-53, C-9
- verification_match_undriven_signals variable C-9
- verification_merge_duplicated_registers variable C-9
- verification_parameter_checking variable C-9
- verification_partition_timeout_limit variable C-9
- verification_passing_mode variable C-9
- verification_progress_report_interval variable C-9
- verification_propagate_const_reg_x variable C-9
- verification_set_undriven_signals variable C-9
- verification_status variable C-9
- verification_super_low_effort_first_pass variable 6-20, C-9
- verification_timeout_limit variable 6-24, C-10
- verification_use_partial_modeled_cells variable C-10

- verification_verify_directly_undriven_output variable 4-5
- verification_verify_unread_compare_points variable C-10
- verification_verify_unread_tech_cells variable C-10
- verify command 6-19, 6-21, 6-34, 8-9, C-24
- verifying
 - black box behavior 5-15
 - black boxes 5-15
 - designs 6-19
 - marking design as black box 5-18
 - single compare point 6-21
- Verilog files 1-16, 1-17, 4-9
 - naming buses 4-9
- Verilog simulation, library files 1-17, 4-19, B-3
- version, displaying during startup 3-4
- VHDL files 1-17, 4-9
- viewing
 - design libraries 1-20
- viewing logic cones 7-38

W

- weak quoting A-5
- which command C-24
- wildcard characters 3-20, 3-21
- windows, managing 3-17
- WORK design library 4-22
- work directory 1-22
- write_container command 5-86, C-24
- write_failing_patterns command 7-47, C-24
- write_hierarchical_verification_script command 6-30, C-25
- write_library_debug_scripts command 8-15, C-25
- writing container to disk 5-86

Z

- zooming in and out 7-36