

# **DFT Compiler**

## **User Guide: Scan**

---

Version B-2008.09-SP2, December 2008

### Comments?

Send comments on the documentation by going to <http://solvnet.synopsys.com>, then clicking “Enter a Call to the Support Center.”

**SYNOPSYS®**

# Copyright Notice and Proprietary Information

Copyright © 2008 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, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, Design Compiler, DesignWare, Formality, HDL Analyst, HSPICE, Identify, iN-Phase, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Synplicity, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, the Synplicity logo, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

## Trademarks (™)

AFGen, Apollo, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HAPS, HapsTrak, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance plus ASIC Prototyping System, HSIM, HSIM<sup>plus</sup>, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, System Compiler, System Designer, Taurus, TotalRecall, 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 . . . . .	xx
About This Guide . . . . .	xx
Customer Support . . . . .	xxii
<b>1. Key Design-for-Test (DFT) Flows and Methodologies</b>	
Design-for-Test (DFT) Flows in the Logical Domain . . . . .	1-2
Unmapped Design Flow . . . . .	1-2
Synthesizing Your Design . . . . .	1-4
Postprocessing Your Design . . . . .	1-5
Building Scan Chains . . . . .	1-6
Mapped Design Flow . . . . .	1-8
Reading In Your Design . . . . .	1-10
Performing Scan Replacement and Building Scan Chains . . . . .	1-11
Mapped Designs With Existing Scan Flow . . . . .	1-13
Reading In Your Design . . . . .	1-15
Checking Test Design Rules . . . . .	1-16
Designing Block by Block . . . . .	1-17
Controlling Scan Replacement During Scan Insertion . . . . .	1-17
Hierarchical Scan Synthesis Flow . . . . .	1-18
Introduction to Test Models . . . . .	1-21
Linking Test Models to Library Cells . . . . .	1-23
Checking If a Library Contains CTL Models . . . . .	1-24
Scan Assembly Using Test Models . . . . .	1-24
Saving Test Models for Subdesigns . . . . .	1-25
Using Test Models . . . . .	1-26

Reading Designs Into TetraMAX .....	1-28
Managing Test Models .....	1-28
Top-Level Integration .....	1-31
DFT Flows in Design Compiler Topographical Mode .....	1-32
Supported DFT Features .....	1-33
DFT Insertion Flow in Design Compiler Topographical Mode .....	1-33
Running DFT Insertion in Design Compiler Topographical Mode .....	1-34
Error Messages .....	1-34
Hierarchical Support in Design Compiler Topographical Mode .....	1-35
Top-Level Design Stitching Flow .....	1-35
Bottom-Up/Hierarchical Flow With Test Models .....	1-37
Scan Insertion Methodologies .....	1-41
Bottom-Up Scan Insertion .....	1-41
Top-Down Scan Insertion .....	1-43
DFT Compiler Default Scan Synthesis Approach .....	1-44
Scan Replacement .....	1-45
Scan Element Allocation .....	1-45
Test Signals .....	1-45
Pad Cells .....	1-46
Area and Timing Optimization .....	1-46
Getting the Best Results With Scan Design .....	1-47
DFT Compiler and Power Compiler Interoperability .....	1-48
Improving Testability in Clock Gating .....	1-48
Inserting Control Points in Control Clock Gating .....	1-49
Scan Enable Versus Test Mode .....	1-50
Inserting Observation Points to Control Clock Gating .....	1-52
Choosing a Depth for Observability Logic .....	1-54
Hooking Up Test Ports Through Hierarchies in Power Compiler .....	1-54
Power Compiler/DFT Compiler Interoperability Flows .....	1-57
Using test_mode With Power Compiler .....	1-57
Using Scan Enables With Power Compiler .....	1-58
Connecting Test Pins to Clock Gating Cells Using insert_dft .....	1-61
Design Requirements .....	1-62
Hookup Testport Connections .....	1-63
Design Rule Checking Changes .....	1-63
Specifying a Particular Signal as Test Pin When Automatically Connecting Test Ports to Clock-Gating Cells .....	1-64

Limitations .....	1-66
<b>2. Running RTL Test Design Rule Checking</b>	
Understanding the Flow .....	2-2
Specifying Setup Variables .....	2-3
Generating a Test Protocol .....	2-3
Defining a Test Protocol.....	2-3
Reading in an Initialization Protocol in STIL Format.....	2-4
Setting the Scan Style .....	2-7
Design Examples.....	2-8
Test Protocol Example 1.....	2-8
Test Protocol Example 2.....	2-9
Running RTL Test DRC .....	2-12
Understanding the Violations .....	2-13
Violations That Prevent Scan Insertion .....	2-13
Uncontrollable Clocks.....	2-13
Latches Enabled at Beginning of Clock Cycle .....	2-14
Asynchronous Control Pins in Active State .....	2-14
Violations That Prevent Data Capture .....	2-15
Clock Used As Data .....	2-15
Black Box Feeds Into Clock or Asynchronous Control .....	2-16
Source Register Launch Before Destination	
Register Capture .....	2-16
Registered Clock-Gating Circuitry .....	2-17
Three-State Contention .....	2-18
Clock Feeding Multiple Register Inputs .....	2-18
Violations That Reduce Fault Coverage .....	2-19
Combinational Feedback Loops .....	2-19
Clocks That Interact With Register Input .....	2-20
Multiple Clocks That Feed Into Latches and Flip-Flops .....	2-21
Black Boxes .....	2-22
<b>3. Running Test DRC Debugger</b>	
Starting and Exiting the Graphical User Interface.....	3-2
Exploring the Graphical User Interface.....	3-2
Logic Hierarchy View .....	3-4
Console Window .....	3-4

Command Line . . . . .	3-5
Viewing Man Pages . . . . .	3-5
Menus . . . . .	3-5
Checking Scan Test Design Rules . . . . .	3-5
Examining DRC Violations . . . . .	3-6
Viewing Test Protocols . . . . .	3-6
Viewing Design Violations. . . . .	3-6
Examining DRC Violations. . . . .	3-7
Inspecting DRC Violations . . . . .	3-8
Inspecting Static DRC Violations . . . . .	3-8
Viewing a Violation . . . . .	3-9
Viewing Multiple Violations . . . . .	3-12
Viewing CTL Models. . . . .	3-13
Inspecting Dynamic DRC Violations . . . . .	3-14
Commands Specific to the DFT GUI . . . . .	3-17
gui_inspectViolations . . . . .	3-17
guiWaveAddSignal . . . . .	3-18
guiViolationSchematicAddObjects . . . . .	3-19
<b>4. Performing Scan Replacement</b>	
Scan Replacement Flow . . . . .	4-3
Preparing for Scan Replacement . . . . .	4-4
Selecting a Scan Replacement Strategy . . . . .	4-4
Identifying Barriers to Scan Replacement . . . . .	4-5
Technology Library Does Not Contain Appropriate Scan Cells . . . . .	4-6
Unsupported Sequential Cells . . . . .	4-7
Attributes That Prevent Scan Replacement . . . . .	4-8
Invalid Clock Nets . . . . .	4-9
Invalid Asynchronous Pins . . . . .	4-11
Preventing Scan Replacement . . . . .	4-12
Specifying a Scan Style . . . . .	4-12
Types of Scan Styles . . . . .	4-12
Multiplexed Flip-Flop Scan Style. . . . .	4-13
Clocked Scan Style. . . . .	4-13
LSSD Scan Style . . . . .	4-13
Scan Style Considerations. . . . .	4-14
Setting the Scan Style . . . . .	4-15

Verifying Scan Equivalents in the Technology Library .....	4-16
Checking the Technology Library for Scan Cells .....	4-16
Checking for Scan Equivalents .....	4-17
Scan Cell Replacement Strategies .....	4-17
Specifying Scan Cells .....	4-18
Restricting the List of Available Scan Cells.....	4-18
Sample Scan Cell Replacement Strategies .....	4-18
Mapping Sequential Gates in Scan Replacement .....	4-19
Multibit Components .....	4-20
What Are Multibit Components?.....	4-21
How DFT Compiler Assimilates Multibit Components .....	4-21
Controlling Multibit Test Synthesis .....	4-22
Performing Multibit Component Scan Replacement.....	4-22
Disabling Multibit Component Support .....	4-22
Test-Ready Compilation .....	4-23
What Is Test-Ready Compile?.....	4-23
The Test-Ready Compile Flow .....	4-24
Preparing for Test-Ready Compile.....	4-25
Performing Test-Ready Compile in the Logical Domain .....	4-26
Controlling Test-Ready Compile .....	4-26
Comparing Default Compile and Test-Ready Compile .....	4-27
Complex Compile Strategies .....	4-30
Validating Your Netlist .....	4-31
Running the link Command .....	4-31
Running the check_design Command.....	4-32
Performing Constraint-Optimized Scan Insertion .....	4-32
Supported Scan States .....	4-32
Locating Scan Equivalents .....	4-33
Preparing for Constraint-Optimized Scan Insertion .....	4-35
Scan Insertion .....	4-35
Specification Phase .....	4-37
Preview .....	4-39
Synthesis .....	4-39
<b>5. Pre-Scan Test Design Rule Checking</b>	
Test DRC Basics .....	5-2
Test DRC Flow.....	5-2

Preparing Your Design .....	5-4
Creating the Test Protocol .....	5-5
Assigning a Known Logic State .....	5-5
Performing Test Design Rule Checking .....	5-5
Analyzing and Debugging Violations .....	5-6
Summary of Violations .....	5-6
Enhanced Reporting Capability .....	5-7
Test Design Rule Checking Messages .....	5-8
Test Design Rule Checking Message Generation .....	5-9
Understanding Test Design Rule Checking Messages.....	5-9
Effects of Violations on Scan Replacement.....	5-9
Viewing the Sequential Cell Summary.....	5-10
Classifying Sequential Cells .....	5-10
Sequential Cells With Violations .....	5-11
Cells With Scan Shift Violations .....	5-11
Black-Box Cells.....	5-12
Constant Value Cells .....	5-12
Sequential Cells Without Violations .....	5-12
Checking for Modeling Violations .....	5-12
Black-Box Cells .....	5-13
Correcting Black Box Cells .....	5-13
Unsupported Cells .....	5-14
Generic Cells .....	5-16
Scan Cell Equivalents .....	5-17
Scan Cell Equivalents and the <code>dont_touch</code> Attribute.....	5-17
Latches .....	5-18
Nonscan Latches .....	5-18
Setting Timing Attributes .....	5-18
Protocols for Common Design Timing Requirements .....	5-19
Strobe-Before-Clock Protocol.....	5-19
Strobe-After-Clock Protocol .....	5-19
Setting Timing Attributes .....	5-20
<code>test_default_period</code> Attribute.....	5-20
<code>test_default_delay</code> Variable.....	5-20
<code>test_default_bidiir_delay</code> Attribute .....	5-21
<code>test_default_strobe</code> Variable .....	5-22
<code>test_default_strobe_width</code> Variable.....	5-22
The Effect of Timing Attributes on Vector Formatting.....	5-24

Creating Test Protocols.....	5-25
Design Characteristics for Test Protocols .....	5-25
scan_style Attribute .....	5-25
signal_type Attributes .....	5-25
Clock Ports .....	5-26
Asynchronous Control Ports.....	5-26
Bidirectional Ports.....	5-26
STIL Test Protocol File Syntax.....	5-26
Defining the test_setup Macro .....	5-27
Defining Basic Signal Timing .....	5-27
Defining the load_unload Procedure .....	5-29
Defining the Shift Procedure.....	5-29
Defining an Initialization Protocol.....	5-30
Scan Shift and Parallel Cycles.....	5-32
Multiplexed Flip-Flop Scan Style.....	5-32
Clocked-Scan Scan Style .....	5-33
LSSD Scan Style .....	5-33
Examining a Test Protocol File .....	5-34
Updating a Protocol in a Scan Chain Inference Flow .....	5-36
Masking DRC Violations.....	5-36
Setting the Severity of DRC Violations .....	5-36
Resetting the Severity of DRC Violations .....	5-38
Reporting the Severity of DRC Violations .....	5-39

## **6. Architecting Your Test Design**

Configuring Your DFT Architecture.....	6-3
Defining Your Scan Architecture .....	6-3
Setting Design Constraints .....	6-4
Defining Constant Input Ports During Scan .....	6-4
Specifying Test Ports .....	6-4
Specifying Individual Scan Paths.....	6-5
Previewing Your Scan Design .....	6-6
Using preview_dft Versus report_scan_path .....	6-7
Architecting Scan Chains .....	6-7
Specifying a Scan Chain for the Current Design .....	6-8
Controlling the Scan Chain Length .....	6-8
Specifying Limits for Individual Scan Chain Length .....	6-8
Specifying the Global Scan Chain Exact Length .....	6-9

Specifying the Global Scan Chain Length Limit . . . . .	6-9
Determining the Scan Chain Count . . . . .	6-10
Balancing Scan Chains . . . . .	6-11
Multiple Clock Domains . . . . .	6-11
Multibit Components and Scan Chains. . . . .	6-14
Controlling the Routing Order . . . . .	6-15
Routing Scan Chains and Global Signals . . . . .	6-17
Rerouting Scan Chains . . . . .	6-17
Stitching Scan Chains Without Optimization . . . . .	6-18
Specifying a Stitch-Only Design . . . . .	6-18
Mapping the Replacement of Nonscan Cells to Scan Cells . . . . .	6-18
Conditions Under Which Scan Cells Are Excluded or Nonscan Cells Become Scan Cells . . . . .	6-20
Using Existing Subdesign Scan Chains. . . . .	6-22
Uniquifying Your Design. . . . .	6-24
Reporting Scan Path Information on the Current Design . . . . .	6-25
Architecting Scan Signals . . . . .	6-25
Specifying Scan Signals for the Current Design . . . . .	6-26
Selecting Test Ports . . . . .	6-31
Sharing Scan-In Pins With Multiple Scan Chains. . . . .	6-31
Sharing a Scan Input With a Functional Port . . . . .	6-32
Sharing a Scan Output With a Functional Port. . . . .	6-32
Associating Scan Enable Ports With Multiple Scan Chains . . . . .	6-33
Using Dedicated Scan Output Ports. . . . .	6-34
Suppressing Replacement of Sequential Cells . . . . .	6-34
In Logical Scan Synthesis. . . . .	6-35
Changing the Scan State of a Design . . . . .	6-35
Removing Scan Specifications . . . . .	6-36
Keeping Specifications Consistent. . . . .	6-37
Synthesizing Three-State Disabling Logic . . . . .	6-37
Configuring Three-State Buses . . . . .	6-40
Configuring External Three-State Buses . . . . .	6-40
Configuring Internal Three-State Buses . . . . .	6-41
Overriding Global Three-State Bus Configuration Settings . . . . .	6-41
Disabling Three-State Buses and Bidirectional Ports . . . . .	6-41
Handling Bidirectional Ports. . . . .	6-42
Setting Individual Bidirectional Port Behavior. . . . .	6-42
Fixed Direction Bidirectional Ports . . . . .	6-43

Using Scan Lock-Up Elements .....	6-43
Assigning Test Port Attributes .....	6-46
Architecting Test Clocks .....	6-46
Setting Test Clocks .....	6-47
Specifying Clock Timing Attributes .....	6-47
The Waveform Section .....	6-47
Specifying the Clock Period .....	6-47
Multiplexed Flip-Flop Design Example .....	6-48
Handling Multiple Clock Designs .....	6-49
Internal Clocks .....	6-49
Assigning Scan Chains to Specific Clocks .....	6-53
Requirements for Valid Scan Chain Ordering .....	6-53
Using Multiple Master Clocks in LSSD Designs .....	6-55
Dedicated Test Clocks for Each Clock Domain .....	6-55
Controlling LSSD Slave Clock Routing .....	6-56
Modifying Your Scan Architecture .....	6-59
Post-Scan Test Design Rule Checking .....	6-60
Preparing for Test Design Rule Checking After Scan Insertion .....	6-60
Checking for Topological Violations .....	6-61
Checking for Scan Connectivity Violations .....	6-62
Scan Chain Extraction .....	6-62
Causes of Common Violations .....	6-62
Ability to Load Data Into Scan Cells .....	6-63
Incomplete Test Configuration .....	6-63
Invalid Clock Logic .....	6-64
Incorrect Clock Timing Relationship .....	6-67
Nonscan Sequential Cells .....	6-69
Ability to Capture Data Into Scan Cells .....	6-70
Clock Driving Data .....	6-71
Untestable Functional Path .....	6-72
Uncontrollable Asynchronous Pins .....	6-72
<b>7. Advanced DFT Architecture Methodologies</b>	
Performing Scan Extraction .....	7-2
Inserting Observe Test Points .....	7-3
Understanding Observe Points .....	7-5
Reading In Your Netlist and Configuring for Scan .....	7-5

Configuring for Observe Points .....	7-6
Enabling Observe Point Analysis .....	7-6
Defining an Observe Clock .....	7-6
Defining a Test Mode .....	7-7
Selecting and Implementing Observe Test Logic .....	7-7
Previewing Scan and Observe Test Point Logic .....	7-13
Inserting Observe Test Logic With Scan Chains .....	7-13
 Using AutoFix .....	7-14
Understanding the Flow .....	7-14
When to Use AutoFix .....	7-16
Configuring AutoFix .....	7-19
Enabling Test Point Utilities .....	7-19
Specifying Test Point Signals .....	7-19
Specifying AutoFix Behavior .....	7-19
Previewing and Inserting Scan Chains and Test Points .....	7-21
AutoFix Script Example .....	7-21
Top-Down and Bottom-Up Design Flows .....	7-22
Top-Down Example .....	7-22
Bottom-Up Example .....	7-23
 Implementing User-Defined Test Points .....	7-26
Types of User-Defined Test Points .....	7-26
Force Test Points .....	7-26
Control Test Points .....	7-28
Observe Test Points .....	7-30
Test Point Options .....	7-31
User-Defined Test Points Example .....	7-32
 Pipelining Scan Enable Architecture .....	7-34
Using Multimode Scan Architecture .....	7-36
Reconfiguring Scan Chains .....	7-36
Defining a Test Mode .....	7-37
Defining the Name of a Test Mode .....	7-37
Defining the Usage of a Test Mode .....	7-38
Defining the Encoding of a Test Mode .....	7-39
Defining the Encoding for All Specified Test Modes .....	7-42
Examining the Encoding in the Preview Report .....	7-44
Assigning Scan Specifications to a Test Mode .....	7-44
Assigning Common Scan Specifications to All Test Modes .....	7-45

Assigning Common Scan Specifications to a Specific Test Mode .....	7-47
Multimode Scan Insertion Script Examples .....	7-48
Multivoltage Support.....	7-55
Configuring Scan Insertion for Multivolts .....	7-56
Conflicting Voltage Mixing Specifications .....	7-56
Configuring Scan Insertion for Multiple Power Domains .....	7-56
Conflicting Power Domain Mixing Specifications .....	7-57
Generating a SCANDEF File in the Presence of Isolation Cells.....	7-57
Scan Extraction Flows in the Presence of Isolation Cells.....	7-57
Mixture of Multivolts and Multiple Power Domain Specifications .....	7-57
Reusing a Multivoltage Cell .....	7-59
Level Shifter as Part of Scan Path .....	7-59
Isolation Cell as Part of Scan Path .....	7-61
DFT Considerations for Low-Power Design Flows.....	7-63
Previewing a Multivoltage Scan Chain.....	7-65
Running in UPF Mode .....	7-66
Limitations .....	7-67
Power-Aware Functional Output Gating .....	7-68
Internal Pins Flow .....	7-70
Understanding the Architecture .....	7-71
DFT Commands .....	7-71
Enabling the Internal Pins Flow .....	7-72
Specifying Hookup Pins .....	7-72
Scan Insertion Flow .....	7-73
Mixing Ports and Internal Pins.....	7-75
Limitations .....	7-76
Support for Implicit Scan Chain Elements .....	7-76
Command Usage .....	7-76
Usage Scenario .....	7-77
Example Script.....	7-79
Example Protocol.....	7-80
Limitations .....	7-81
Creating Scan Groups .....	7-81
Configuring Scan Grouping .....	7-82
Creating Scan Groups .....	7-82
Removing Scan Groups .....	7-83

Integrating an Existing Scan Chain Into a Scan Group .....	7-83
Reporting Scan Groups .....	7-84
Scan Group Flows .....	7-85
Known Limitations .....	7-85
Identification of Shift Registers .....	7-85
<b>8. Wrapping Cores</b>	
Core Wrapping Commands .....	8-2
Wrapper Cells .....	8-2
Test Wrapper Operation.....	8-4
Activating the SAFE Mode .....	8-7
Test Wrapper Control Interface .....	8-8
Test Wrapper Cell Interface .....	8-8
Dedicated Wrapper Cell .....	8-9
Sharing Existing Design Registers for Wrapping .....	8-10
Delay Test Wrapper Insertion .....	8-14
Delay Wrapper Test Modes.....	8-15
Separate Input and Output Wrapper Chains and Control .....	8-16
Wrapping Three-State and Bidirectional Ports .....	8-17
Specifying Multiple Wrapper Modes .....	8-18
Test Wrapper Exceptions.....	8-18
Specifying Wrapper Chains .....	8-19
Specifying Multiple Wrapper Chains.....	8-19
Specifying Wrapper Cell Order.....	8-20
Specifying Wrapper Chain Ports.....	8-20
Specifying Wrapper Chain Length .....	8-21
Specifying Input-Only or Output-Only Wrapper Chains .....	8-21
Controlling Wrapper Chains with Scan Inputs, Scan Outputs, and Shift Enables .....	8-22
Automatically Create Additional Wrapper Modes With Separate Wrapper Chains.....	8-22
Core Wrapping Flows .....	8-22
Scan-Stitched Core Flow.....	8-22
Core Creation Flow .....	8-26
Core Wrapping Scripts .....	8-29
Core Wrapping With a Dedicated Wrapper .....	8-29
Core Wrapping With a Shared Wrapper .....	8-31

Core Wrapping With Dedicated Delay Wrapper .....	8-32
Core Wrapping With Shared Delay Wrapper .....	8-33
<b>9. On-Chip Clocking Support</b>	
Background .....	9-3
Supported Flows .....	9-4
Definitions .....	9-4
Capabilities .....	9-5
Limitations .....	9-5
Design Flows .....	9-6
Logical Representation of an OCC Controller .....	9-10
Basic Processes .....	9-12
Setting Up Your Environment .....	9-12
Enabling On-Chip Clocking Support .....	9-12
OCC Supported Flows .....	9-12
OCC and Clock Chain Synthesis Insertion Flow .....	9-12
Defining Clocks .....	9-13
Defining Global Signals .....	9-15
Configuring the OCC Controller .....	9-16
Specifying Scan Configuration .....	9-17
Sample Script .....	9-17
User-Defined Instantiated Clock Controller and Clock Chain Insertion Flow .....	9-18
Defining Clocks .....	9-19
Defining Global Signals .....	9-21
Specifying Clock Chains .....	9-22
Specifying Scan Configuration .....	9-23
Sample Script .....	9-23
Reporting Clock Controller Information .....	9-25
OCC and Clock Chain Synthesis Insertion Flow .....	9-25
User-Defined Clock Controller and Clock Chain Insertion Flow .....	9-25
DRC Support .....	9-26
Enabling the OCC Controller Bypass Configuration .....	9-27
Example Configurations on a Design .....	9-27
Example 1 .....	9-28

Example 2 .....	9-28
Example 3 .....	9-29
Waveform and Capture Cycle Example .....	9-30
<b>10. Exporting to Other Tools</b>	
Verifying DFT Inserted Designs for Functionality .....	10-2
SVF File Generation .....	10-2
Test Information Passed to the SVF File .....	10-3
Sample Script .....	10-3
Formality Tool Limitations .....	10-4
Exporting a Design to TetraMAX ATPG .....	10-4
Before Exporting Your Design .....	10-4
Support for DFT Compiler Commands in TetraMAX ATPG .....	10-5
Creating Generic Capture Procedures .....	10-5
Exporting Your Design to TetraMAX ATPG .....	10-14
SCANDEF-Based Reordering Flow .....	10-16
Overview .....	10-16
Generation of a SCANDEF File .....	10-17
Reading and Compiling the Design .....	10-17
Specifying the Scan Configuration .....	10-17
Writing Out the SCANDEF File .....	10-17
Generating a SCANDEF File for Typical Flows .....	10-18
Generating SCANDEF Files for Hierarchical Flows .....	10-18
Hierarchical SCANDEF Flow Support .....	10-19
DFT Commands .....	10-19
Hierarchical SCANDEF Flows .....	10-20
Impact of DFT Configuration Specification on SCANDEF File Generation .....	10-24
Case 1: .....	10-24
Case 2: .....	10-24
Case 3: .....	10-24
Case 4: .....	10-25
Case 5: .....	10-25
Case 6: .....	10-25
Case 7: .....	10-26
Case 8: .....	10-26
Support for Other DFT Features .....	10-27
Limitations With SCANDEF Generation .....	10-27

Exporting of SCANDEF Files to Third-Party Tools. . . . . 10-27

**Index**



# Preface

---

This preface includes the following sections:

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

---

## What's New in This Release

For information about new features, enhancements, and changes, along with known problems and limitations and resolved Synopsys Technical Action Requests (STARs), is available in the *DFT Compiler Release Notes* in SolvNet.

To access the *DFT Compiler Release Notes*,

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

<http://solvnet.synopsys.com/DownloadCenter>

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the registration instruction.

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

---

## About This Guide

The *DFT Compiler User Guide: Scan* describes the processes and flows for using scan technology in DFT Compiler.

---

### Audience

This manual is intended for ASIC design engineers who have some experience with testability concepts and strategies and for test and design-for-test engineers who want to understand how basic test automation concepts and practices relate to DFT Compiler.

---

### Related Publications

For additional information about DFT Compiler, see Documentation on the Web, which is available through SolvNet at the following address:

<http://solvnet.synopsys.com/DocsOnWeb>.

You might want to refer to the documentation for the following Synopsys products:

- Design Compiler
- BSD Compiler
- TetraMAX

---

## 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.)
<b>Courier bold</b>	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 low   medium   high (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 an 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 to the Support Center.”

To access SolvNet:

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 top-right menu bar or in the footer.

---

### **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), and 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](mailto:support_center@synopsys.com) from within North America.
  - Find other local support center e-mail addresses at [http://www.synopsys.com/support/support\\_ctr](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](http://www.synopsys.com/support/support_ctr).

# 1

## Key Design-for-Test (DFT) Flows and Methodologies

---

This chapter describes the key flows, methodologies, processes, and default behavior to consider when planning your design-for-test (DFT) project. For a complete introduction to DFT processes and flows, see Chapter 2 of the *DFT Overview User Guide*.

This chapter includes the following sections:

- [Design-for-Test \(DFT\) Flows in the Logical Domain](#)
- [Hierarchical Scan Synthesis Flow](#)
- [DFT Flows in Design Compiler Topographical Mode](#)
- [Scan Insertion Methodologies](#)
- [DFT Compiler Default Scan Synthesis Approach](#)
- [Getting the Best Results With Scan Design](#)
- [DFT Compiler and Power Compiler Interoperability](#)

---

## Design-for-Test (DFT) Flows in the Logical Domain

This section introduces the recommended DFT flows for simple designs in the logical domain. The flow you use depends on the state of the design before you start using DFT Compiler, as detailed in the following sections:

- [Unmapped Design Flow](#)
  - [Mapped Design Flow](#)
  - [Mapped Designs With Existing Scan Flow](#)
  - [Designing Block by Block](#)
  - [Controlling Scan Replacement During Scan Insertion](#)
- 

### Unmapped Design Flow

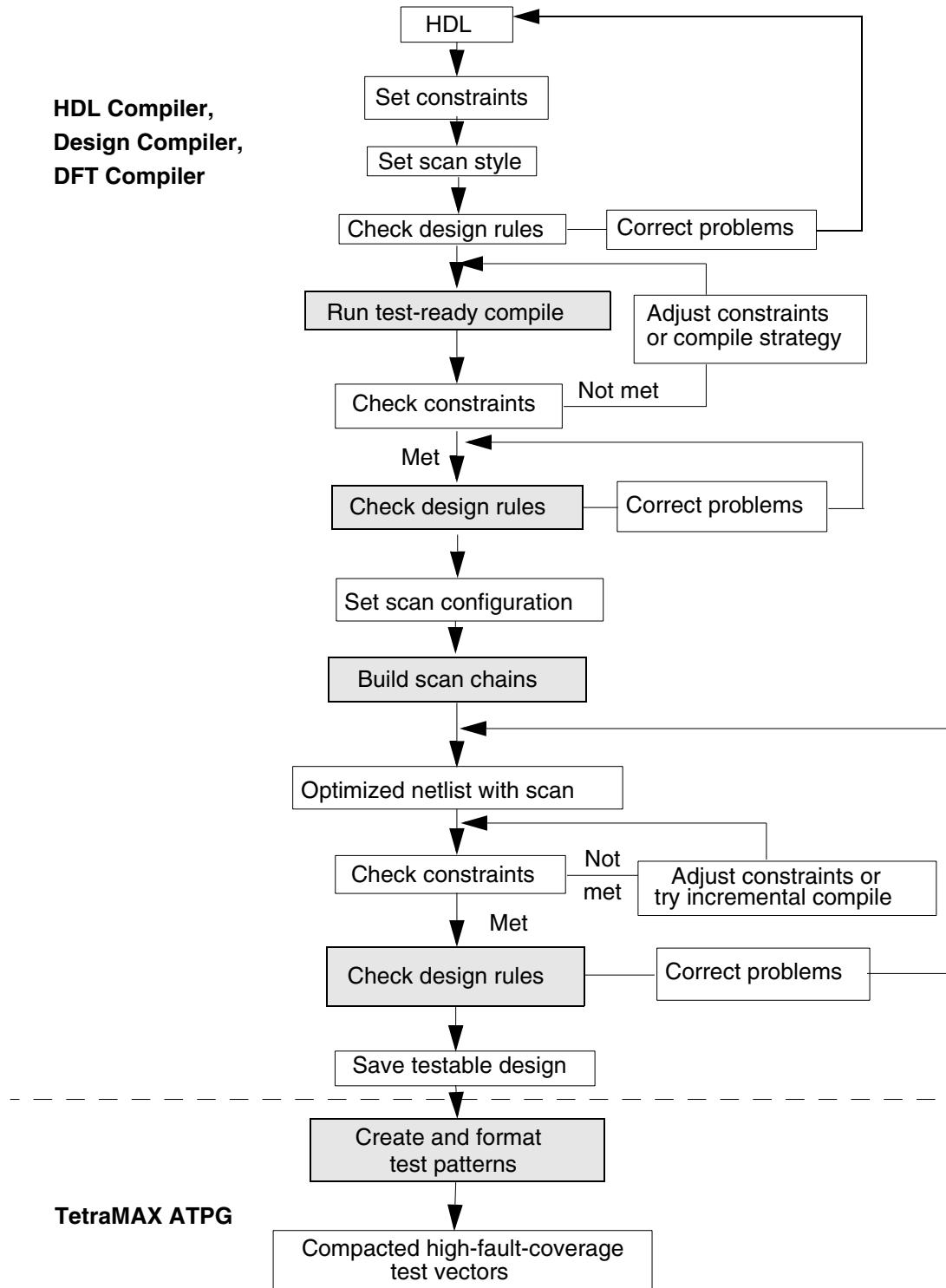
This flow applies to unmapped designs. Using the command sequence provided in this section, you can take a design from an HDL-level circuit description that does not contain existing scans to a fully optimized design with internal scan circuitry. The sample command sequence applies to full-scan and partial-scan designs. The basic process consists of

- [Synthesizing Your Design](#)
- [Postprocessing Your Design](#)
- [Building Scan Chains](#)
- [Exporting a Design to TetraMAX ATPG](#)

For complete information about how to pass your design to TetraMAX ATPG to generate test patterns, see [Chapter 10, “Exporting to Other Tools.”](#) Also, see the *TetraMAX ATPG User Guide* for information about generating test patterns, using TetraMAX after a DFT Compiler flow.

[Figure 1-1](#) shows a typical unmapped design flow that uses the HDL Compiler and DFT Compiler tools. At the end of the flow, the TetraMAX ATPG tool produces a set of high fault-coverage test vectors that you can readily adapt to your target tester.

Figure 1-1 Typical Flat Design Flow From an Unmapped Design



## Synthesizing Your Design

To synthesize your design, perform the following steps:

1. Select the target technology library and the link technology library.

```
dc_shell> set target_library asic_vendor.ddc
dc_shell> set link_library \
           [list "*" asic_vendor.db]
```

2. Set up for design rule checking on the RTL source.

```
dc_shell> set hdlin_enable_rtldrc_info true
```

For more information on setting up RTL design rule checking (DRC), see “[Specifying Setup Variables](#)” on page 2-3.

3. Use one of the following commands to read in an HDL circuit description:

```
dc_shell> read_file -format verilog design_name.v
dc_shell> read_file -format vhdl design_name.vhd
```

4. Explicitly link the design:

```
dc_shell> link
```

If DFT Compiler is unable to resolve any references, you must provide the missing designs before proceeding. See your Design Compiler documentation for details.

5. Select the design constraints. In this example, the design is constrained to be no larger than 1,000 vendor units in area and runs with a 20-ns clock period. Enter the following commands:

```
dc_shell> set_max_area 1000
dc_shell> create_clock clock_port -period 20 \
           -waveform [list 10 15]
```

For additional related information on setting design constraints, see [Chapter 6, “Architecting Your Test Design.”](#)

6. Set the test timing variables to the values required by your ASIC vendor. If you are using TetraMAX to generate test patterns and your vendor does not have specific requirements, the following settings produce the best results:

```
dc_shell> set test_default_delay 0
dc_shell> set test_default_bidir_delay 0
dc_shell> set test_default_strobe 40
dc_shell> set test_default_period 100
```

7. Select the scan style if you did not previously set the `test_default_scan_style` variable in the `.synopsys_dc.setup` file. In this example, the scan style is multiplexed flip-flop.

Enter the command

```
dc_shell> set test_default_scan_style \
           multiplexed_flip_flop
```

You can use the `set_scan_configuration -style` command instead.

For more information on setting the scan style, see “[Setting the Scan Style](#)” on page 2-7.

8. Define clocks and asynchronous set and reset signals in your design, and then generate a test protocol.

```
dc_shell> create_test_protocol -infer_clock \
           -infer_async
```

For more information on generating a test protocol, see “[Defining an Initialization Protocol](#)” on page 5-30.

9. Check test design rules in the RTL source file, using RTL Test DRC.

```
dc_shell> dft_drc
```

For more information, see “[Understanding the Violations](#)” on page 2-13.

10. Synthesize a design that meets the constraints you set, and map your circuit description to cells from the target technology library.

```
dc_shell> compile -scan
```

This performs a test-ready compile. DFT Compiler synthesizes and optimizes your design relative to area and speed, and it removes redundant logic. This last feature eliminates logically untestable circuitry and is an important part of the Synopsys test methodology. Because of the `-scan` option, all flip-flops in the design are implemented as scan flip-flops.

For details on test-ready compile, see “[Test-Ready Compilation](#)” on page 4-23.

## Postprocessing Your Design

After your design is synthesized, perform the following postprocessing steps:

1. Check that you have satisfied the constraints you specified in step 5 of the previous procedure. Enter the following command:

```
dc_shell> report_constraint -all_violators
```

2. This step is optional. You can save a copy of the design at this point. Enter the following command:

```
dc_shell> write -format ddc \
           -out design_test_ready.ddc
```

Note:

Do not save a copy as ASCII text (Verilog, VHDL, or EDIF formats) if you intend to start a new session, using the saved design. DFT Compiler marks the test-ready logic with attributes that are lost when you save the design as ASCII text.

3. Generate a new test protocol for the synthesized design.
4. Check the test design rules, according to the scan style you chose in step 7 of the previous procedure.

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

For more information on Test DRC, see [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

## Building Scan Chains

To build scan chains, perform the following steps:

1. If you want DFT Compiler to buffer the scan-enable signal, use `set_dft_signal` to identify the dedicated port and declare a drive strength for the port. Enter the following commands:

```
dc_shell> set_dft_signal -view spec -port \
           scan_enable_port -type ScanEnable \
           -active_state 1
```

```
dc_shell> set_drive 2 scan_enable_port
```

If you do not follow this procedure, DFT Compiler creates a scan-enable port and assumes an infinite drive strength on the port. Therefore, no buffering is inserted.

2. Because you ran a test-ready compile, you do not want DFT Compiler to perform a scan replacement on the design. To prevent DFT Compiler from issuing a warning when you build scan chains, enter the command

```
dc_shell> set_scan_configuration -replace false
```

When you build scan chains by using the `insert_dft` command, you implicitly instruct DFT Compiler to perform scan replacement. If you do not enter the `set_scan_configuration -replace false` command, DFT Compiler recognizes that the design is already scan replaced and issues a warning, stating that DFT Compiler is overriding your implicit instruction to perform scan replacement.

The `set_scan_configuration` command determines most of the aspects of how DFT Compiler makes designs scannable. In this example, all of the default settings are used except for the `-replace false` option.

3. Build the scan chains in your design. Enter the following command:

```
dc_shell> insert_dft
```

When you add scan-test circuitry to a design, its area and performance change. DFT Compiler minimizes the effect of adding scan-test circuitry on compile design rules and performance by using synthesis routines. For details of synthesis concepts such as compile design rules, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*. For details on how DFT Compiler fixes compile design rule violations and performance constraint violations, see the [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

4. Check that you have satisfied the constraints you specified in step 5 of the preceding section, [“Synthesizing Your Design” on page 1-4](#). Enter the command

```
dc_shell> report_constraint -all_violators
```

5. Recheck the test design rules, according to the scan style you chose in step 7 of the preceding section, [“Synthesizing Your Design” on page 1-4](#).

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. The checks that are run are more comprehensive than those in step 10 of the [“Synthesizing Your Design” on page 1-4](#) and also include checks for the correct operation of the scan chain. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

See [“Post-Scan Test Design Rule Checking” on page 6-60](#) for more information on this process.

The `dft_drc` command is an essential preprocess step to guarantee that the reports generated by the `report_scan_path` command are correct. You must run the `dft_drc` command if you want to generate reports with the `report_scan_path` command.

6. You can now use the `report_scan_path` command to generate test reports. For example, to view details of the scan chain, use the command

```
dc_shell> report_scan_path -view existing \
           -chain all
```

Note:

Rerun the `dft_drc` command before running `report_scan_path` only if you have modified your circuit since you last ran `dft_drc`.

7. Write out the complete design in the Synopsys database format. Enter the following command:

```
dc_shell> write -format ddc -hierarchy \
           -out design.ddc
```

---

## Mapped Design Flow

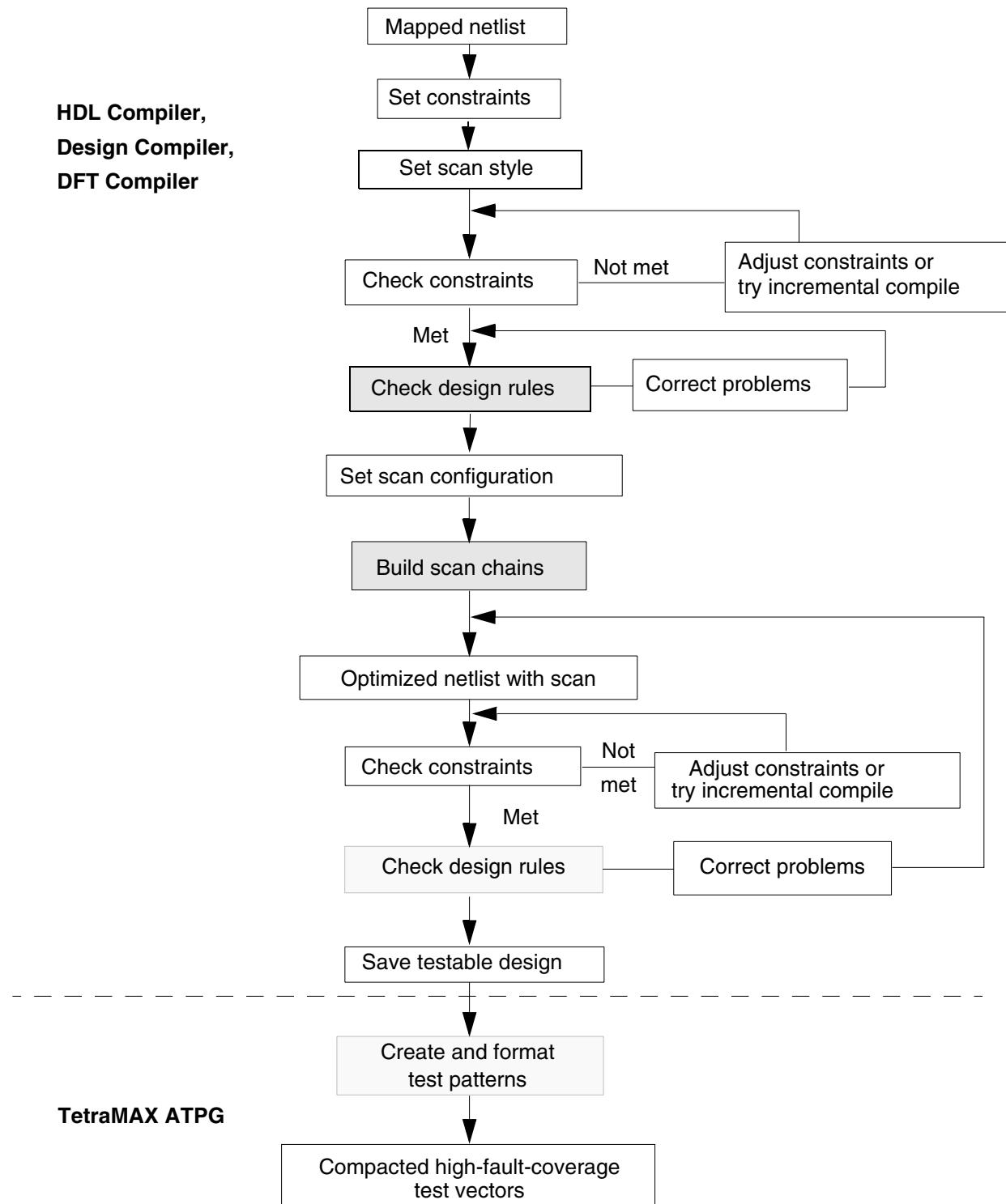
Using the command sequence provided in this section, you can take a mapped design that does not contain existing scans to a fully optimized design with internal scan circuitry. The sample command sequence applies to a full-scan or partial-scan design. The basic process consists of

- [Reading In Your Design](#)
- [Performing Scan Replacement and Building Scan Chains](#)
- [Exporting a Design to TetraMAX ATPG](#)

For information about how to pass the design on to TetraMAX ATPG to generate test patterns, see [Chapter 10, “Exporting to Other Tools.”](#) Also, see the *TetraMAX ATPG User Guide* for information about generating test patterns by using TetraMAX from a DFT Compiler flow.

[Figure 1-2](#) shows a typical mapped design flow for DFT Compiler and HDL Compiler. At the end of the design flow, TetraMAX ATPG produces a set of high fault-coverage test vectors that you can readily adapt to your target tester.

*Figure 1-2 Typical Flat Design Flow From a Mapped Design*



## Reading In Your Design

Perform the following steps to read in your design:

1. Select the target technology library and link technology library.

```
dc_shell> set target_library asic_vendor.ddc
dc_shell> set link_library [list * asic_vendor.ddc]
```

2. Use one of the following commands to read in an HDL circuit description:

```
dc_shell> read_file -format ddc design_name.ddc
dc_shell> read_file -format verilog design_name.v
dc_shell> read_file -format vhdl design_name.vhd
```

If you want DFT Compiler to buffer the scan-enable signal, include a dedicated scan-enable port in your design. At this stage, the scan-enable port should not be connected.

3. Explicitly link the design:

```
dc_shell> link
```

If Design Compiler is unable to resolve any references, you must provide the missing designs before proceeding. See your Design Compiler documentation for details.

4. Select the target technology library and the design constraints. In this example, the design is constrained to be no larger than 1,000 vendor units in area, and it runs with a 20-ns clock period. Enter the following commands:

```
dc_shell> set_max_area 1000
dc_shell> create_clock clock_port -period 20 \
           -waveform [list 10 15]
```

Note:

If the file you read is in the Synopsys logical database format (a .ddc file), the target technology library and the design constraints might already be set.

5. Set the test timing variables to the values required by your ASIC vendor. If you are using TetraMAX to generate test patterns and your vendor does not have specific requirements, the following settings produce the best results:

```
dc_shell> set test_default_delay 0
dc_shell> set test_default_bidir_delay 0
dc_shell> set test_default_strobe 40
dc_shell> set test_default_period 100
```

6. Select the scan style if you did not previously set the `test_default_scan_style` variable in the `.synopsys_dc.setup` file. In this example, the scan style is multiplexed flip-flop.

Enter the command

```
dc_shell> set test_default_scan_style \
           multiplexed_flip_flop
```

You can use the `set_scan_configuration -style` command instead.

For more information on setting the scan style, see “[Setting the Scan Style](#)” on page 2-7.

7. Check that you have satisfied the constraints you specified in step 4. Enter the command

```
dc_shell> report_constraint -all_violators
```

8. Check the test design rules, according to the scan style you chose in step 6:

```
dc_shell> create_test_protocol -infer_clock \
           -infer_async
```

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage. See [Chapter 5, “Pre-Scan Test Design Rule Checking,”](#) for more information on design rule checking.

Because this design did not go through a test-ready flow, the `dft_drc` command checks the library for equivalent scan cells for all the flip-flops in your design.

## Performing Scan Replacement and Building Scan Chains

To perform scan replacement and build scan chains, follow these steps:

1. If you want DFT Compiler to buffer the scan-enable signal, use `set_dft_signal` to identify the port and declare a drive strength for the port.

Enter the commands

```
dc_shell> set_dft_signal -view spec \
           -port scan_enable_port \
           -type ScanEnable -active_state 1
```

```
dc_shell> set_drive 2 scan_enable_port
```

If you do not follow this procedure, DFT Compiler creates a scan-enable port and assumes an infinite drive strength on the port so that no buffering is inserted.

2. Specify the scan architecture. In this example, direct DFT Compiler to build two scan chains. Use the command

```
dc_shell> set_scan_configuration -chain_count 2
```

The `set_scan_configuration` command determines most of the aspects of how DFT Compiler makes designs scannable. For more information about the `set_scan_configuration` command, see [Chapter 6, “Architecting Your Test Design.”](#)

3. Build the scan chains in your design. Enter the command

```
dc_shell> insert_dft
```

Because your design has no scan cells, DFT Compiler replaces the flip-flops in your design with scannable equivalents before it builds scan chains.

When you add scan-test circuitry to a design, the design’s area and performance change. DFT Compiler minimizes the effect of adding scan-test circuitry on the compile design rules and design performance by using synthesis routines. For details of synthesis concepts such as compile design rules, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*. For details about how DFT Compiler fixes compile design rule violations and performance constraint violations, see [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

4. Check that you have satisfied the constraints you specified in step 4 of the preceding section, [“Reading In Your Design” on page 1-10](#). Enter the command

```
dc_shell> report_constraint -all_violators
```

5. Recheck the test design rules, according to the scan style you chose in step 6 of the preceding section, See [“Reading In Your Design” on page 1-10](#).

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. The checks run are more comprehensive than those in step 8 of the [“Reading In Your Design” on page 1-10](#) and include checks for the correct operation of the scan chain. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

The `dft_drc` command is an essential preprocess step to guarantee that the reports generated by the `report_scan_path` command are correct. You must run the `dft_drc` command if you want to generate reports with the `report_scan_path` command.

See [“Post-Scan Test Design Rule Checking” on page 6-60](#) for more information on this process.

6. You can now use the `report_scan_path` command to generate test reports. For example, to view details of the scan chain, use the command

```
dc_shell> report_scan_path -view existing -chain all
```

Note:

Rerun the `dft_drc` command before running `report_scan_path` only if you have modified your circuit since you last ran `dft_drc`.

7. Write out the complete design in Synopsys database format. Enter the command

```
dc_shell> write -format ddc -hierarchy \
           -output    design.ddc
```

---

## Mapped Designs With Existing Scan Flow

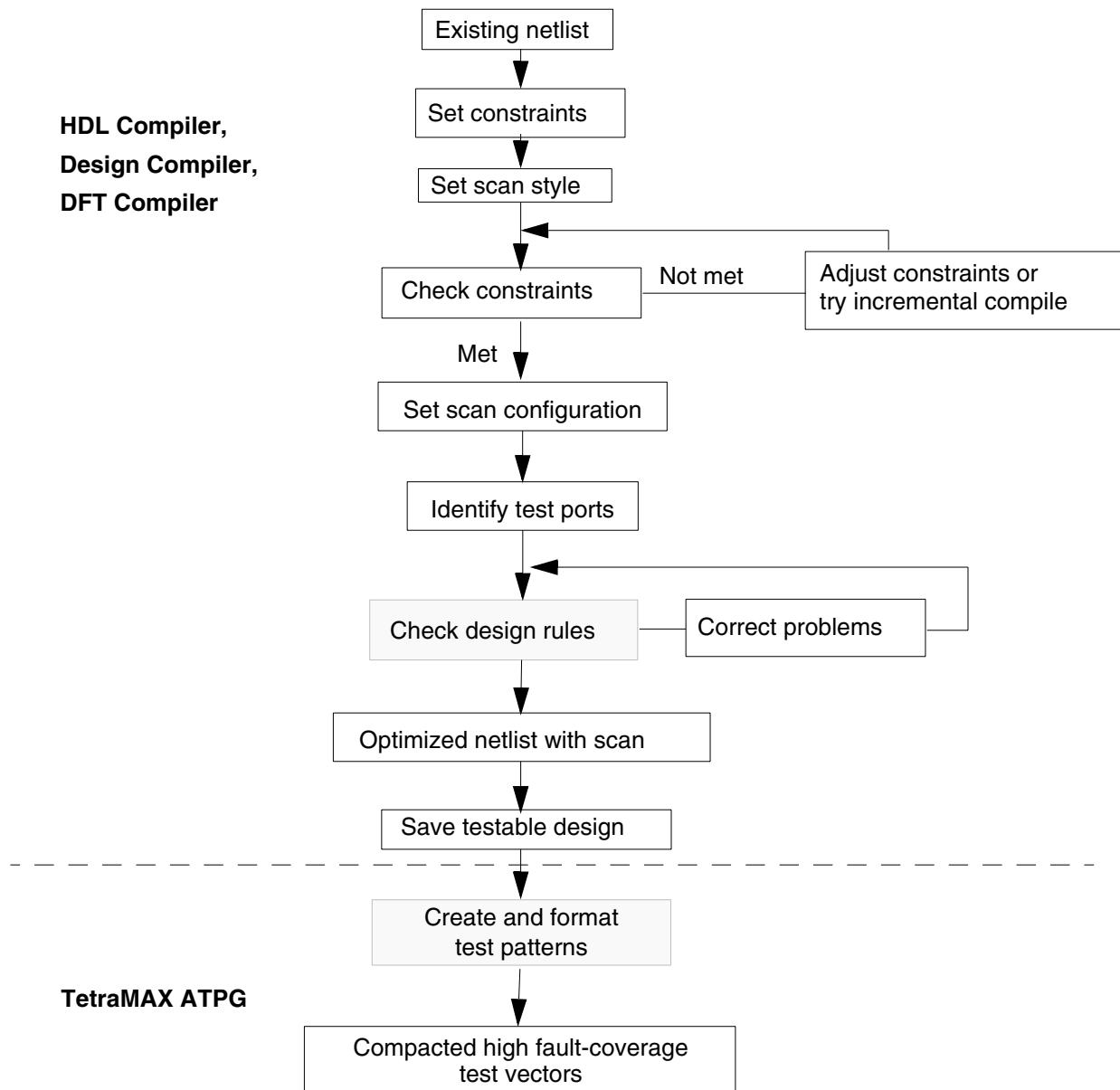
Using the command sequence provided in this section, you can take a design from an HDL-level circuit description that contains existing scans to a fully optimized design with internal scan circuitry. The sample command sequence applies to a full-scan or partial-scan design. The basic process consists of

- [Reading In Your Design](#)
- [Checking Test Design Rules](#)
- [Exporting Your Design to TetraMAX ATPG](#)

For information about how to pass the design to TetraMAX ATPG to generate test patterns, see *Chapter 9, “Exporting to Other Tools.”* Also, see the *TetraMAX ATPG User Guide* on how to generate test patterns by using TetraMAX from a DFT Compiler flow.

[Figure 1-3](#) shows a typical flow for using DFT Compiler on a design with existing scan chains. At the end of the design flow, TetraMAX ATPG produces a set of high fault-coverage test vectors that you can readily adapt to your target tester.

*Figure 1-3 Typical Flat Design Flow With Existing Scan Chains*



## Reading In Your Design

To read in your design, follow these steps:

1. Select the target technology library and link technology library.

```
dc_shell> set target_library asic_vendor.ddc
dc_shell> set link_library [list * asic_vendor.ddc]
```

2. Use one of the following commands to read in an HDL circuit description:

```
dc_shell> read_file -format ddc design_name.ddc
dc_shell> read_file -format verilog design_name.v
dc_shell> read_file -format vhdl design_name.vhdl
```

3. Explicitly link the design:

```
dc_shell> link
```

If Design Compiler is unable to resolve any references, you must provide the missing designs before proceeding. See your Design Compiler documentation for details.

4. Select the target technology library and the design constraints. In this example, the design is constrained to be no larger than 1,000 vendor units in area, and it runs with a 20-ns clock period. Enter the following variables:

```
dc_shell> set max_area 1000
dc_shell> create_clock clock_port -period 20 \
           -waveform [list 10 15]
```

Note:

If the file you read in is a .ddc file, the target technology library and the design constraints might already be set.

5. Set the test timing variables to the values required by your ASIC vendor. If you are using TetraMAX to generate test patterns and your vendor does not have specific requirements, the following settings produce the best results:

```
dc_shell> set test_default_delay 0
dc_shell> set test_default_bidir_delay 0
dc_shell> set test_default_strobe 40
dc_shell> set test_default_period 100
```

6. Select the scan style if you did not previously set the `test_default_scan_style` variable in the `.synopsys_dc.setup` file. In this example, the scan style is multiplexed flip-flop.

Enter the command

```
dc_shell> set test_default_scan_style \
           multiplexed_flip_flop
```

You can also use the `set_scan_configuration`-style command. See “[Architecting Your Test Design](#)” on page [6-1](#) for details.

7. Check that you have satisfied the constraints you specified in step 4. Enter the command

```
dc_shell> report_constraint -all_violators
```

## Checking Test Design Rules

To check test design rules, follow these steps:

1. Identify the test ports to DFT Compiler. This example has a single scan chain. Enter the commands

```
dc_shell> set_scan_state scan_existing  
  
dc_shell> set_scan_path c0 -view existing \  
          -scan_data_in [get_ports test_si] \  
          -scan_data_out [get_ports test_so] \  
          -scan_enable [get_ports test_se] \  
          -infer_dft_signals  
  
dc_shell> create_test_protocol -infer_clock \  
          -infer_async
```

Note:

If you read in the design from a .ddc file, this attribute might already be set. You can check this with the `report_dft_signal -view existing` command.

2. Check the test design rules, according to the scan style you chose in step 6 of the preceding section, “[Reading In Your Design](#)” on page [1-15](#), and using the test attributes set in step 1 of this section.

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. The checks include checking for the correct operation of the scan chain during shift and capture. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

Running the `dft_drc` command is an essential preprocess to guarantee that the reports generated by the `report_scan_path` command are correct. You must run the `dft_drc` command if you want to generate reports with the `report_scan_path` command.

See [Chapter 5, “Pre-Scan Test Design Rule Checking,”](#) for more information on design rule checking.

3. You can now use the `report_scan_path` command to generate test reports. For example, to view details of the scan chain, use the command

```
dc_shell> report_scan_path -view exist -chain all
```

Note:

Rerun the `dft_drc` command before running `report_scan_path` only if you have modified your circuit since you last ran `dft_drc`.

4. Write out the complete design in Synopsys database format. Enter the command

```
dc_shell> write -format ddc -hierarchy -out  
design.ddc
```

---

## Designing Block by Block

Many designers prefer to develop their designs on a block-by-block basis. Most of the processes described in the preceding sections can be applied to a particular block of your design, as well as to the complete design. For example, you can check the design rules and generate an initial fault coverage report on each block of your design as it is developed. The report helps you identify the blocks that have unacceptable fault coverage, which enables you to fix testability problems at an early stage. See [Chapter 5, “Pre-Scan Test Design Rule Checking”](#), for more information on pre-scan DRC.

Although fixing testability problems on a block-by-block basis is an important “divide-and-conquer” technique, testability problems are global in nature. A completely testable subblock might show testability problems when it is embedded in its environment.

---

## Controlling Scan Replacement During Scan Insertion

By default, the `insert_dft` command performs scan replacement in your design prior to scan stitching. Here, design rule checking determines if each sequential element has the appropriate test pins (scan-in, scan-out, scan enable, and so on) defined for the selected scan style. If no violations are found and if the appropriate test pins are found, the cells are included in the scan chain. Otherwise, DRC checks to see if each of the sequential elements in your design has a scan equivalent for the selected scan style. If no violations are found on the cells and if a scan equivalent is present, the cells are scan-replaced and included in the scan chain.

If you have a design that is already scan replaced and you do not want the `insert_dft` command to perform scan replacement, specify the following command prior to stitching scan chains:

```
set_scan_configuration -replace false
```

When you specify this command, DRC only checks to see if each sequential element has the appropriate test pins for the chosen scan style. If no violations are found on the cells and the appropriate test pins are found, the cells are included in the scan chain.

However, if you have a .ddc file that is written out after test-ready compile, by using the `compile -scan` or `compile_ultra -scan` commands, then you do not need to specify again the `set_scan_configuration -replace false` command. Because the design is already scan-replaced, the scan insertion process does not perform scan replacement.

[Table 1-1](#) summarizes the scan replacement flow.

*Table 1-1 Scan Replacement Using set\_scan\_configuration -replace*

<code>set_scan_configuration -replace</code>	<code>false</code>	<code>true (default)</code>
Scan equivalent check	Check based on presence of test pins for the scan style chosen	Check based on presence of test pins for the scan style chosen, and based on scan equivalence for the scan style chosen
Cells violated and message	TEST-126: cells do not have valid test pins for the scan style	TEST-120: cells have no scan equivalents based on synthesis mapping
Scan replacement	No	Yes, if needed
<code>preview_dft</code> , <code>insert_dft</code>	Valid cells included in scan chain	Valid cells included in scan chain

---

## Hierarchical Scan Synthesis Flow

Hierarchical Scan Synthesis (HSS) enables you to specify scan chain designs and implement a top-down, bottom-up, or middle-out hierarchical scan insertion flow. Because HSS reduces memory usage and decreases runtime, it is useful when you are working with large designs.

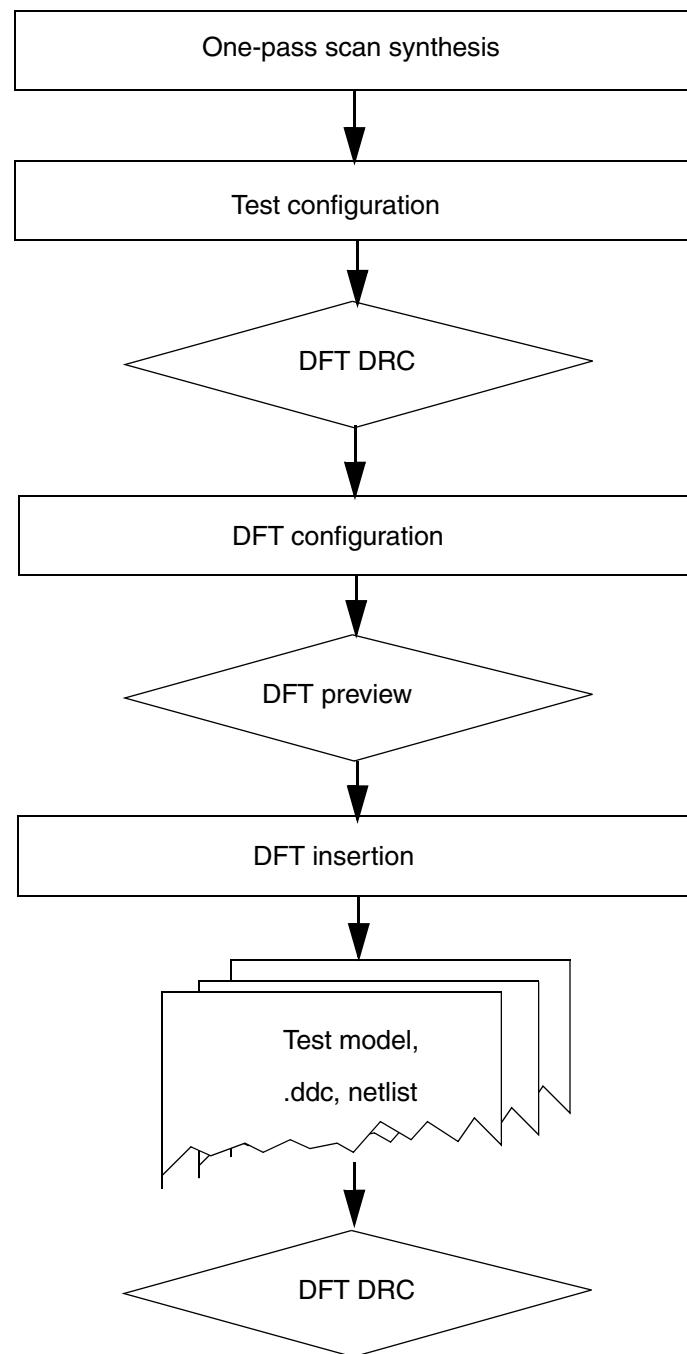
HSS enables you to do the following:

- Implement automatically balanced scan chains
- Specify complete scan chains
- Generate scan chains that enter and exit a design module multiple times
- Automatically fix certain scan rule violations

- Reuse existing modules that already contain scan chains
- Control the routing order of scan chains in the hierarchy
- Perform scan insertion from the top or the bottom of the design
- Implement automatically enabling or disabling logic for bidirectional ports and internal three-state logic
- Implement automatically multiplexed and enabling or disabling logic to shared function ports as test data ports

[Figure 1-4](#) describes the module-level test synthesis flow used for HSS.

*Figure 1-4 Module-Level Test Synthesis Flow*



---

## Introduction to Test Models

HSS automatically creates test models that describe only the portions of subdesigns that are important for inserting the DFT flow. Test models store the following test information:

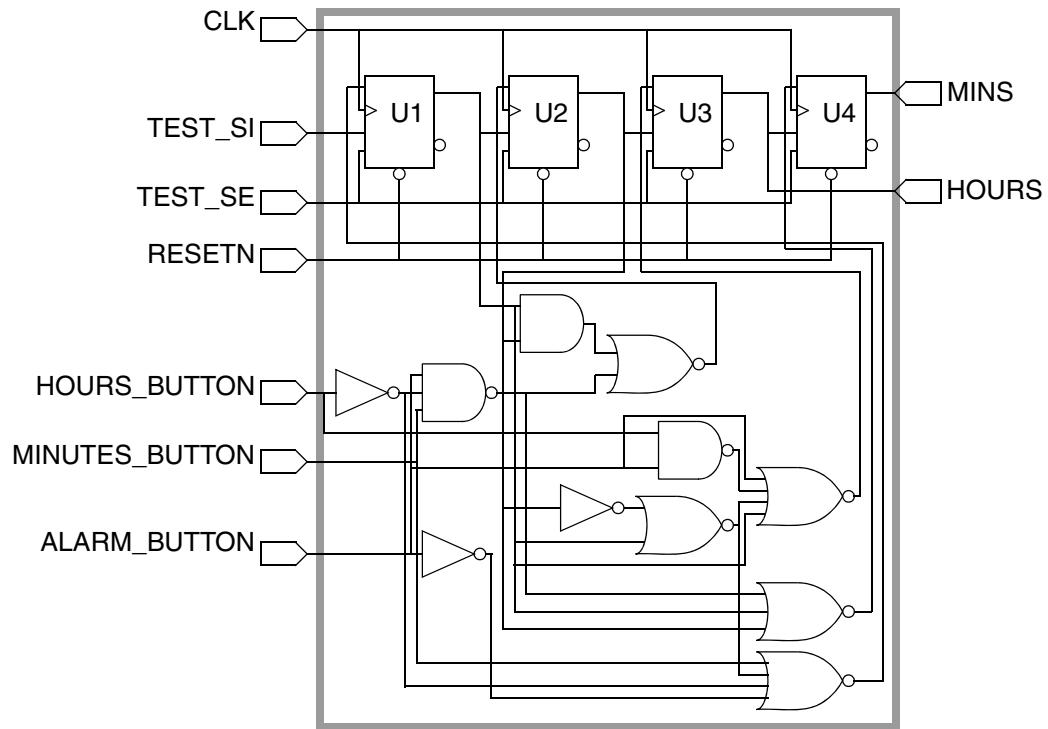
- Port names
- Port directions (input, output, bidirectional)
- Scan structures
- Scan clocks
- Asynchronous sets and resets
- Three-state disables
- Test attributes on ports
- Test protocol information, such as initialization, sequencing, and clock waveforms

Note the following limitations related to the use of test models:

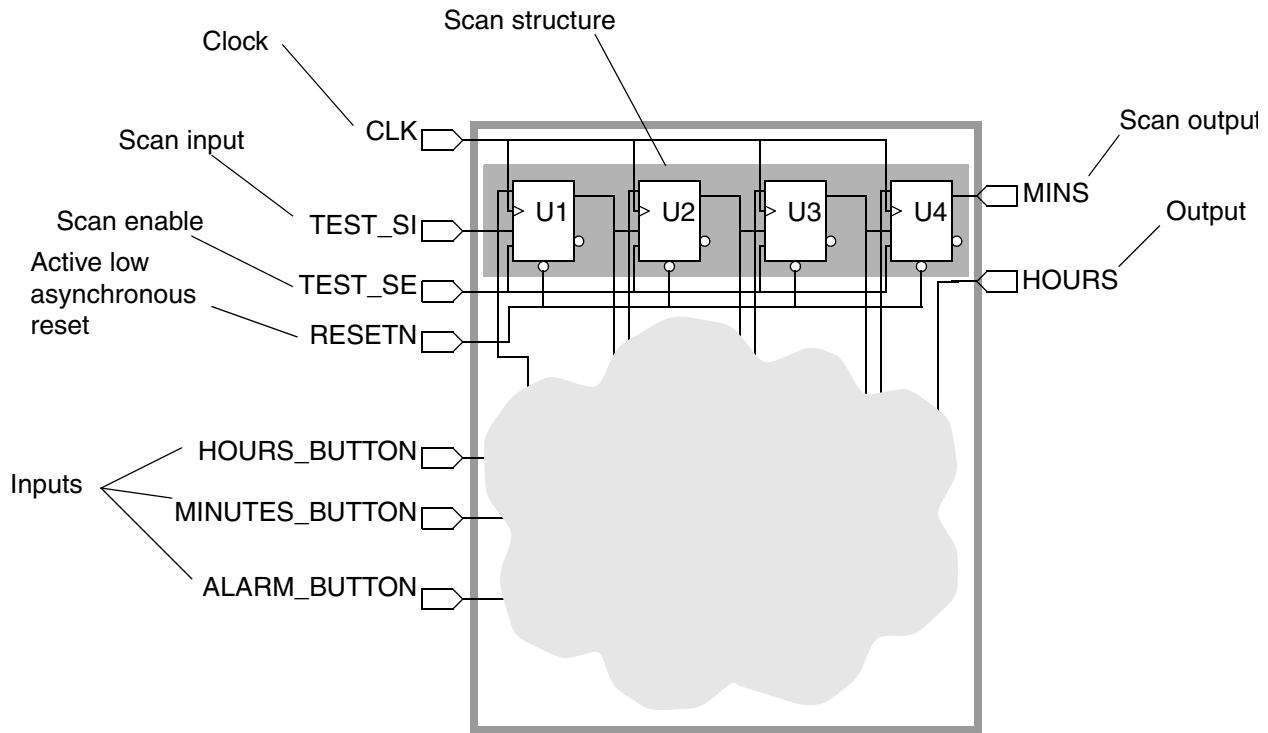
- Because DFT Compiler saves the test model as an attribute to the design in .ddc binary format, you cannot use a text editor to view the test model.
- Test models are not adequate for the generation of test patterns, so you must have an actual netlist representation for use with automatic test pattern generation (ATPG).

[Figure 1-5](#) shows a sample design schematic, and [Figure 1-6](#) shows how a test model might represent the test attributes of that design.

Figure 1-5 Sample Schematic



*Figure 1-6 Test Model Representation*




---

## Linking Test Models to Library Cells

When library cells have built-in scan chains, Core Test Language (CTL) test models must be linked, either inside the library or inside the design, for DFT Compiler to connect the scan chains at the top level. To link a test model inside the library, enter the command

```
read_lib lib_file.lib -test_model \
cell_name:model_file.ctl
```

To link multiple cells in one library, use the following syntax:

```
read_lib lib_file.lib -test_model [list \
model_file1.ctl model_file2.ctl]
```

To link a test model inside a design, which must be done if it has not been linked in the library, enter the command

```
read_test_model -format ctl -design cell_name \
model_file.ctl
```

Note:

If your library already contains library test models and you do not want to overwrite those models with CTL test models, do not use the `read_test_model` command. Remove this command from your scripts as well.

---

## Checking If a Library Contains CTL Models

To determine if a library has CTL models attached to it, read in the library with the `link` command or the `compile` command, and then run `report_lib`. If the library has CTL models attached, the report will indicate `ctl` in the `Attributes` list, as shown in [Example 1-1](#).

### *Example 1-1 Simple report\_lib Output*

Cell	Footprint	Attributes
my_memory	b, d, s, u,	ctl, t

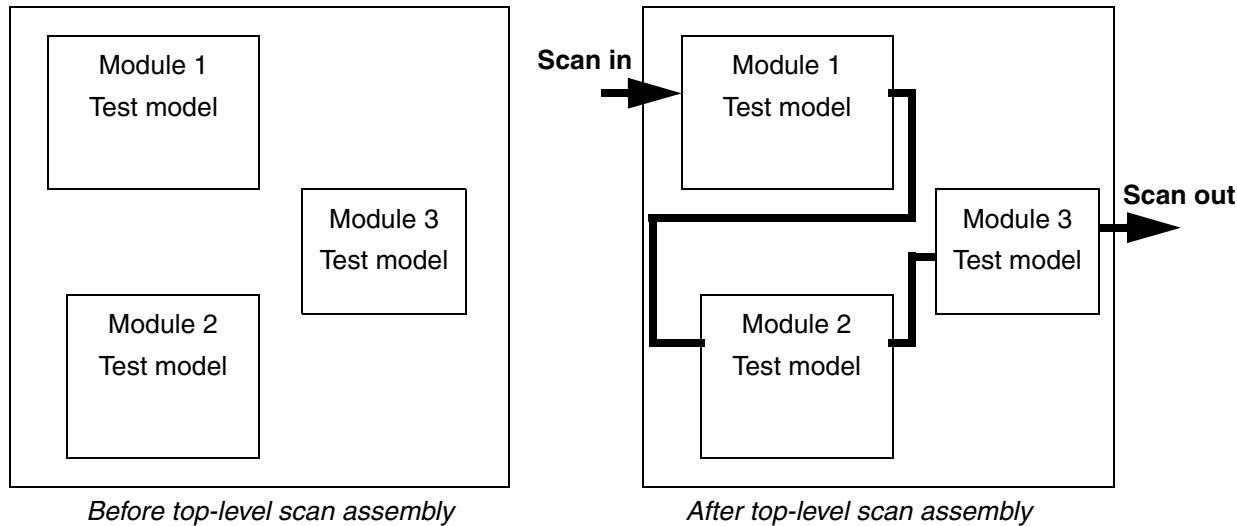
---

## Scan Assembly Using Test Models

In practice, you should use test models to represent subdesigns, and then stitch the test models together at the top level of the design. Because the test model stores less information than a full gate-level representation in a .ddc database, a design with test models uses less memory than a design with subdesigns represented by gate-level .ddc files.

[Figure 1-7](#) shows scan assembly at the top level of a design with several subdesigns represented by test models. The top-level scan assembly process also works if some of the subdesigns are implemented at the gate level or if user-defined logic exists at the top level.

*Figure 1-7 Scan Assembly Using Test Models*



The following sections contain more detail about creating and using test models.

---

## Saving Test Models for Subdesigns

At the end of the flow, use the `write_test_model` command to save the test model to disk. Alternatively, you can use the `write` command to generate a .ddc file containing both the test model and gate information.

You should also write out a Verilog or VHDL netlist of the design to use with TetraMAX.

[Example 1-2](#) shows a command sequence for creating and saving a test model on a design that has no existing scan structures.

*Example 1-2 Test Model Creation on Nonscan Design*

```
dc_shell> remove_design -all
dc_shell> read -f verilog ALARM_BLOCK.v
dc_shell> current_design ALARM_BLOCK
dc_shell> link
dc_shell> create_clock CLK -period 100 -waveform [list 0 50]
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
           -timing [list 45 55] -port CLK
dc_shell> set_dft_signal -view existing_dft -port \
           RESETN -type Reset -active_state 0
dc_shell> compile -scan
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> preview_dft -show all
dc_shell> insert_dft
dc_shell> dft_drc -verbose
dc_shell> write -format ddc -hierarchy -output ALARM_BLOCK.ddc
dc_shell> write -format verilog -hierarchy -output ALARM_BLOCK.v
dc_shell> write_test_model -output ALARM_BLOCK.ctlddc
dc_shell> set test_stil_netlist_format verilog
dc_shell> write_test_protocol -output ALARM_BLOCK.spf
dc_shell> exit
```

---

## Using Test Models

At the top level of your design, you have to read in the test models that you created for the subdesigns and perform scan insertion.

To read in the test models at the top level, use the `read_test_model` command:

```
read_test_model test_model_filenames
```

Use the `list_test_models` command to create a list of the test models loaded and the associated designs, or use the `read` command for the subdesign, as with any other .ddc file. When reading this .ddc file at the top level, DFT Compiler automatically uses the test model for the subdesign.

You can use the top-level DFT flow, described in “[Top-Down Scan Insertion](#)” on page 1-43, to accomplish scan insertion at the top level of the design. DFT Compiler uses the test models loaded instead of the .ddc files for subdesigns.

**Important:**

DFT Compiler does not touch scan chains within the test models. It combines scan chains only at the top level to make the total lengths of the combined chains as close as possible.

DFT Compiler recognizes when lock-up latches have been inserted in a test model and does not insert another lock-up latch when connecting scan chains between two test models unless it is necessary.

After top-level scan insertion, write out the test protocol file in Standard Test Interface Language (STIL) format (the only supported format) and write out a Verilog or VHDL top-level netlist for the top-level design. For example, write out top.spf and top.v, as shown in the following commands:

```
dc_shell> write_test_protocol -out top.spf  
dc_shell> write -format verilog -output top.v
```

If your top-level design contains test models for a subdesign, DFT Compiler does not write out a gate-level netlist for that subdesign, even if you use the `-hierarchy` option. The netlist written contains an empty module for the test model subdesign. You must write a gate-level netlist for each subdesign separately.

[Example 1-3](#) shows how test models can be used at the top level of a design.

### *Example 1-3 Test Model Usage*

```
dc_shell> remove_design -all  
dc_shell> read_file -f verilog ALARM_BLOCK.v  
dc_shell> read_file -f verilog TIME_BLOCK.v  
dc_shell> read_file -f verilog COMPUTE_BLOCK.v  
dc_shell> read_test_model alarm_sm_2.ctlddc  
dc_shell> read_file -f verilog COMPARATOR.v  
dc_shell> link  
dc_shell> list_test_models  
dc_shell> current_design COMPUTE_BLOCK  
dc_shell> link  
dc_shell> create_clock CLK -period 100 \  
           -waveform [list 0 50]  
dc_shell> set_dft_signal -view existing \  
           -type ScanClock -timing [list 45 55] -port CLK  
dc_shell> set_dft_signal -view existing_dft \  
           -port RESETN -type Reset -active_state 0  
dc_shell> compile -scan  
dc_shell> set_scan_configuration -chain_count 1  
dc_shell> set_scan_path chain0 -view spec \  
           -ordered_elements [list U1 U5/chain0 U2]  
dc_shell> create_test_protocol  
dc_shell> dft_drc  
dc_shell> preview_dft -show all  
dc_shell> insert_dft  
dc_shell> dft_drc -verbose  
dc_shell> report_scan_path -chain all  
dc_shell> write -format ddc -hierarchy -output COMPUTE_BLOCK.ddc  
dc_shell> write -format verilog -hier -output COMPUTE_BLOCK.v  
dc_shell> write_test_model -output COMPUTE_BLOCK.ctlddc  
dc_shell> set test_stil_netlist_format verilog  
dc_shell> write_test_protocol -output COMPUTE_BLOCK.spf  
dc_shell> exit
```

[Example 1-3](#) shows a typical script for the HSS flow. Note that `dft_drc` is run right after `insert_dft` to verify that scan insertion is correct, without errors. If `dft_drc` reports any serious violations, such as scan chain blockages, you must fix these and rerun the script. Once `dft_drc` is successful, you can run `report_scan_path -chain all` to get more information about scan chains.

Note:

When you use the HSS flow, you must run `dft_drc` after scan insertion and check the results. The results from the `report_scan_path -chain all` command can be considered only after `dft_drc` is error-free.

---

## Reading Designs Into TetraMAX

By default, in TetraMAX, if you read in two modules with the same name, the last one takes precedence. If you have a top-level netlist with empty submodules, read it into TetraMAX first, and then read in the netlists for the submodules. For example,

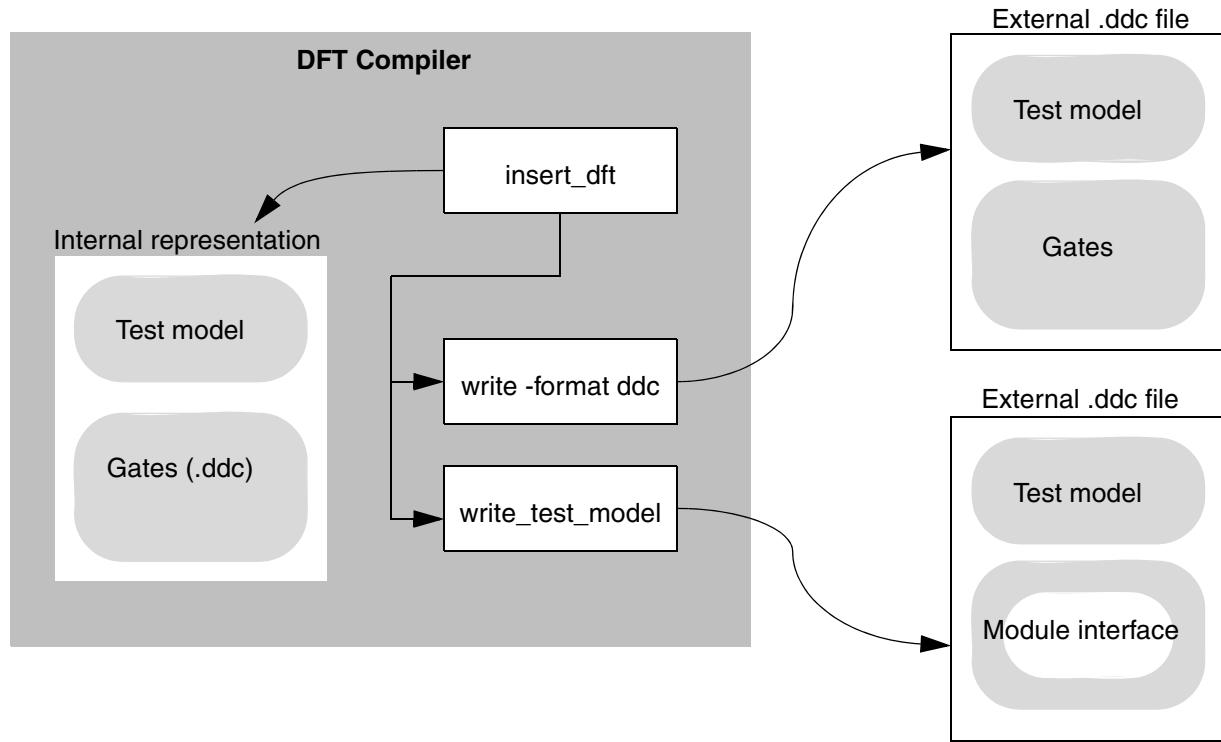
```
BUILD> read netlist top.v
BUILD> read netlist module_1.v module_2.v ... module_n.v
BUILD> run build top
```

---

## Managing Test Models

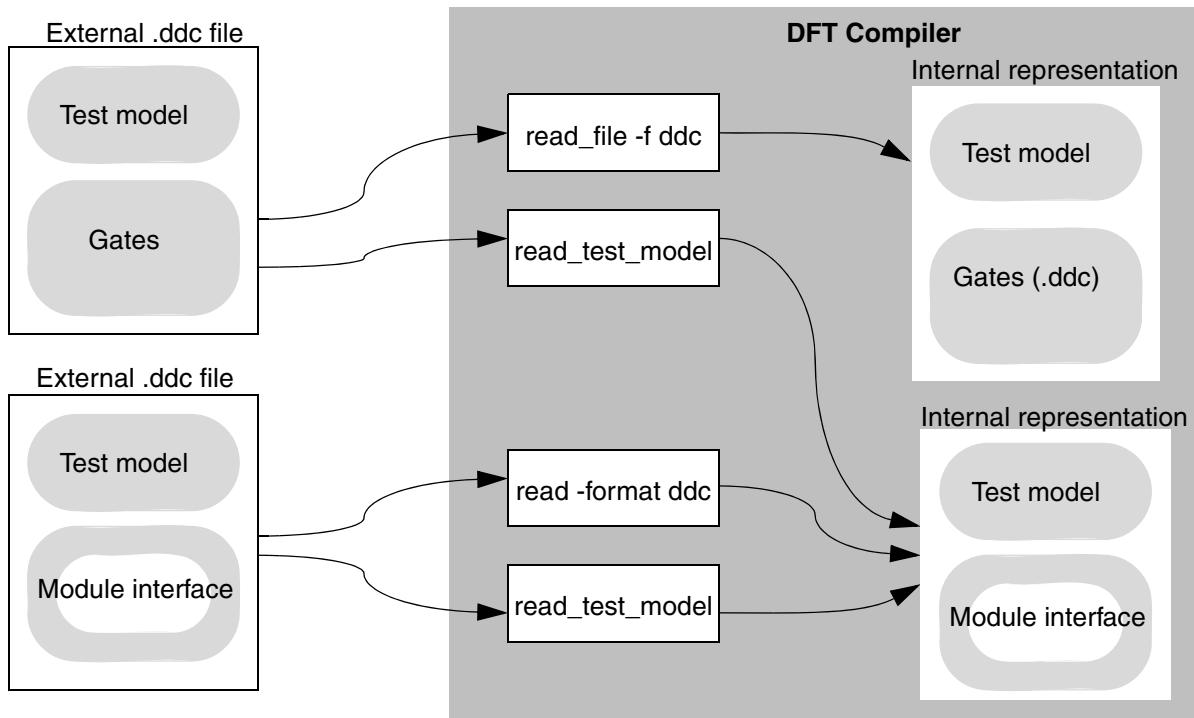
As shown in [Figure 1-8](#), DFT Compiler creates test models when you insert scans. You can save the test models by using the `write` or `write_test_model` command.

*Figure 1-8 Creating and Saving Test Models*



As shown in [Figure 1-9](#), you can read test models into DFT Compiler by using the `read_file` or `read_test_model` command.

Figure 1-9 Using Test Models



The different commands that you can use to write and to read test models enable you to tailor the flow you use to meet your needs. There are several possible flows:

- Use `write_test_model` to save a test model, and use `read_file` or `read_test_model` to read it into DFT Compiler.

In this case, the file created by `write_test_model` contains only the test model and module interface information, so this flow uses less memory and reduces execution time.

This flow is recommended for the greatest reduction in runtime and memory usage, but you have the flexibility of using either of the following flows.

- Use `write` to save a design with a test model, and use `read_file` to read it into DFT Compiler.

The file created by the `write` command contains both a test model and gate information. But the `read_test_model` command reads in only the test model portion of the file and notifies you that DFT Compiler has removed all implementation information. This flow uses less memory and reduces execution time.

- Use `write` to save a design with a test model, and use `read_file` to read it into DFT Compiler.

The `write` command creates a file containing both the test model and the gate information. The `read_file` command reads in both the test model and the gate information, but DFT Compiler uses the test model by default. Because of the test model usage, the execution time is reduced, but you do not save as much memory as with the other flows. This flow does not use commands specific to test models, so it might fit into your existing flow with minimal changes.

---

## Top-Level Integration

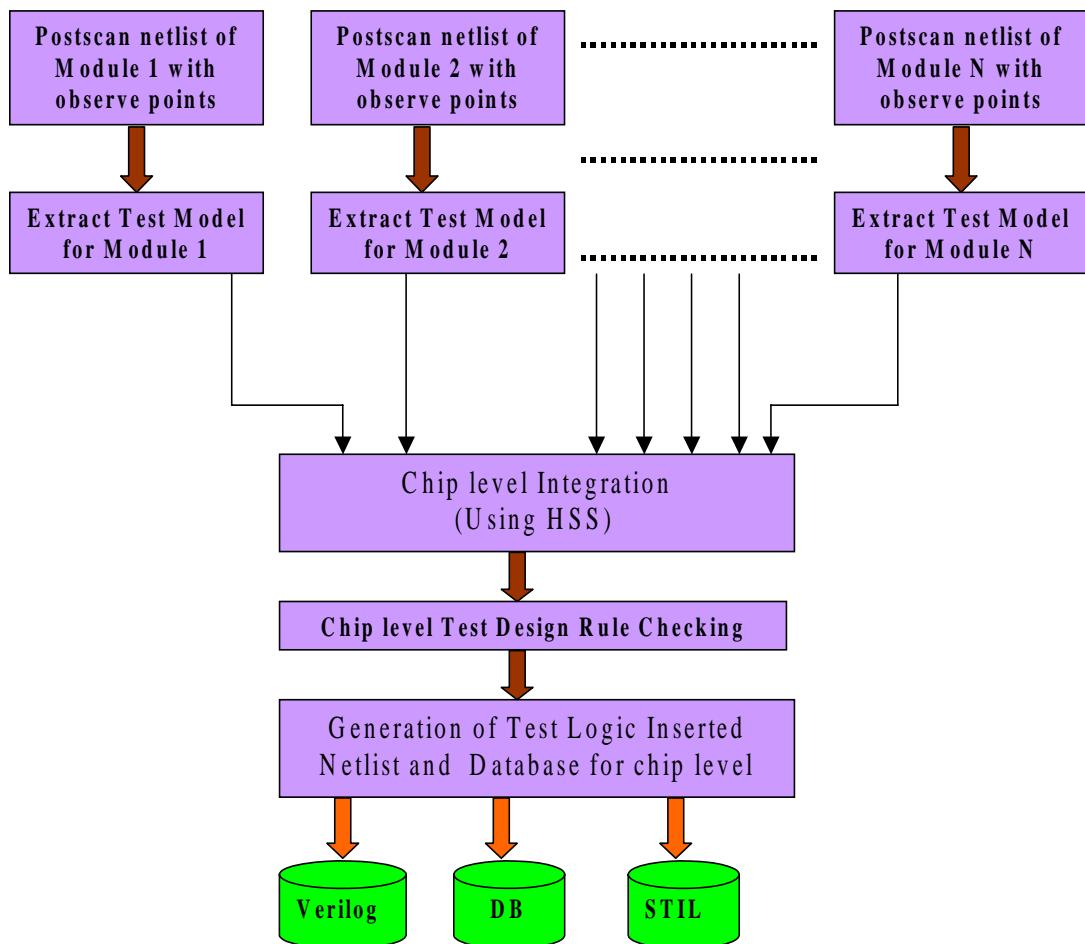
Once the individual modules are complete, with scan chain and observe test logic inserted, stitch the modules together at the top level.

When you use the HSS flow, keep the following points in mind:

- If you are inserting observe points, no observe analysis is performed for any user-defined logic or glue logic at the top level.
- If dedicated clocks exist in some of the modules, they need to be stitched with a chip-level pin to avoid creating another chip-level port.
- You cannot integrate blocks that were created using the internal pin flow.

The HSS flow is shown in [Figure 1-10](#).

Figure 1-10 Chip Level Integration Using the HSS Flow



## DFT Flows in Design Compiler Topographical Mode

DFT Compiler also works within the Design Compiler topographical domain shell (`dc_shell-topo`). Whereas `dc_shell` uses wire load models for timing and area power optimizations, `dc_shell-topo` uses placement timing values instead. For more information, see the *Design Compiler User Guide*.

This section describes the DFT features that are supported in Design Compiler topographical mode. The following topics are covered:

- [Supported DFT Features](#)

- [DFT Insertion Flow in Design Compiler Topographical Mode](#)
  - [Running DFT Insertion in Design Compiler Topographical Mode](#)
  - [Hierarchical Support in Design Compiler Topographical Mode](#)
- 

## Supported DFT Features

The following DFT features are supported:

- Basic scan and adaptive scan flows
- Multiplexed flip-flop scan style
- Multivoltage
- Multiple timing modes, that is, Design Compiler “multimode” (MM) timing for different functional modes (see Design Compiler documentation for further information)
- Clock and asynchronous AutoFix flow for uncontrollable clocks and asynchronous set and reset signal
- Observe and user-defined test points insertion
- Internal pins flow
- Multimode scan architect
- HSS, hierarchical adaptive scan synthesis, and hierarchical adaptive scan synthesis hybrid flows
- BSD Compiler
- On-chip clocking support
- Memories with test models

Note:

DFT Compiler in Design Compiler topographical mode does not support other scan styles.

---

## DFT Insertion Flow in Design Compiler Topographical Mode

The DFT flow in Design Compiler topographical mode is similar to the DFT flow in Design Compiler wire load model mode, except that the `insert_dft` command does not perform any optimization step, that is, only “stitch-only” is run.

See the following simplified DFT-Design Compiler topographical mode flow:

```
compile_ultra -scan  
create_test_protocol  
dft_drc  
preview_dft  
  
insert_dft  
dft_drc  
  
#User must rerun optimization after this  
  
compile_ultra -incremental -scan
```

---

## Running DFT Insertion in Design Compiler Topographical Mode

DFT performs topographical ordering when running in Design Compiler topographical mode. The virtual layout information is utilized, where available, to ensure that there is no severe impact on the functional timing.

When you run DFT Compiler in this mode, the `preview_dft` and `insert_dft` commands print the following message:

```
Running DFT insertion in topographical mode.
```

You should still use the scan chain reordering flow after DFT insertion in Design Compiler topographical mode to obtain optimal results. See “[SCANDEF-Based Reordering Flow](#)” on [page 10-16](#).

## Error Messages

If you run DFT with unsupported commands and options, error messages appear.

Because only “stitch-only” is supported in Design Compiler topographical mode, a warning message is issued if you specify otherwise.

Note the following example:

```
dc_shell-topo> set_dft_insertion_configuration \  
           -synthesis_optimization all
```

```
Warning: Synthesis optimizations for DFT are not allowed in  
DC-Topographical flow. Turning off all the optimizations.  
Accepted insert_dft configuration specification.
```

---

## Hierarchical Support in Design Compiler Topographical Mode

Topographical mode supports two flows:

- [Top-Level Design Stitching Flow](#)
- [Bottom-Up/Hierarchical Flow With Test Models](#)

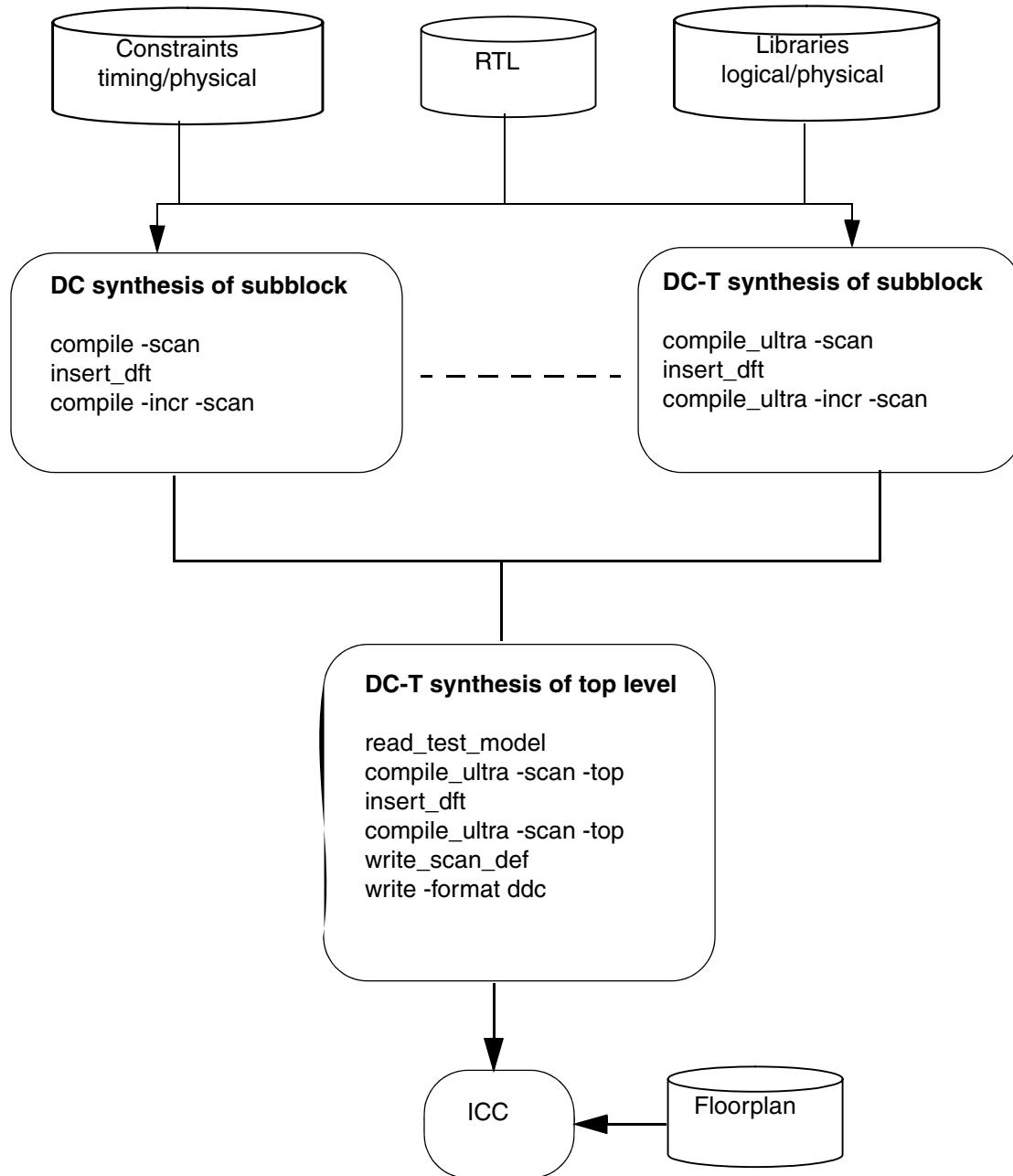
The top-level design stitching flow is used to stitch compiled physical blocks into the top-level design.

The bottom-up hierarchical flow is used when you need to address design and runtime challenges or when you want to use a divide-and-conquer synthesis approach. For more information on the bottom-up HSS flow or the hierarchical adaptive scan synthesis (HASS) flow, see See [“Hierarchical Scan Synthesis Flow” on page 1-18](#). and Chapter 5, Adaptive Scan Synthesis, of the *DFT MAX User Guide: Adaptive Scan* respectively.

### Top-Level Design Stitching Flow

[Figure 1-11](#) shows the top-level design stitching flow in Design Compiler topographical mode.

Figure 1-11 Top-Level Design Stitching Flow



To perform the top-level design stitching flow

1. Set up the design and libraries.

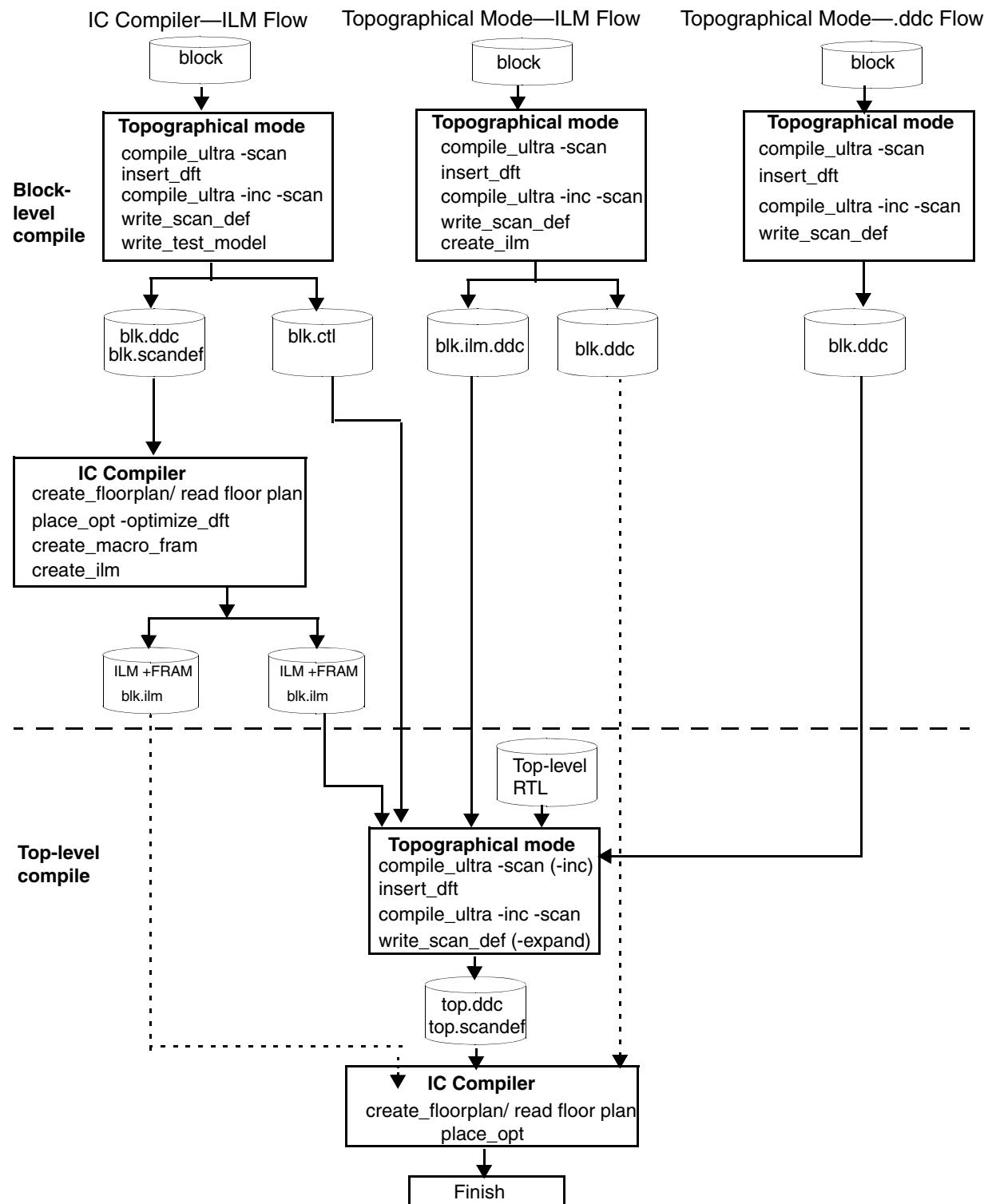
2. Read in the top-level design.
3. Compile each subblock by performing the following steps:
  - a. Set the current design to the subblock.
  - b. Apply timing constraints and power constraints.
  - c. Perform test-ready compile by using the `compile_ultra -scan` command.
  - d. Apply the DFT constraints.
  - e. Use the `insert_dft` command in the subblock so that the inserted scan chains can be included in the top-level scan chain.
  - f. Run the `compile_ultra -scan -incremental` command.
4. Set the current design to the top-level design, link the design, and apply the top-level timing constraints.
5. Run the `compile_ultra -scan -top` command.  
 The `-top` option maps the top-level logic, and the `-scan` option enables the tool to map sequential cells to the appropriate scan flip-flops.
6. Apply DFT constraints.
7. Run the `insert_dft` command to insert scan chains at the top level, followed by `compile_ultra -scan -top` to map any additional unmapped logic that might have been introduced.

## **Bottom-Up/Hierarchical Flow With Test Models**

Bottom-up/hierarchical flows with test models include hierarchical scan synthesis (HSS), hierarchical adaptive scan synthesis (HASS), and hierarchical adaptive scan synthesis-hybrid. In the bottom-up/hierarchical flow, the tool propagates the block-level timing and placement data to the top level and uses this information to drive the top-level optimizations.

[Figure 1-12](#) shows the bottom-up/hierarchical flow with test models in Design Compiler topographical mode.

*Figure 1-12 Bottom-Up/Hierarchical Flow With Test Models*



As shown in [Figure 1-12](#), the tool can read two types of hierarchical blocks in topographical mode:

- Interface logic model (ILM), generated in topographical mode (Topographical Mode: ILM Flow) or in IC Compiler (IC Compiler: ILM Flow)
- Netlist, generated in topographical mode (Topographical Mode: .ddc Flow)

**Block-level hierarchical flow.** To perform the block-level hierarchical flow

1. Set up the design and libraries.
2. Read in the block-level design.
3. Compile each subblock by performing the following steps:
  - a. Set the current design to the subblock.
  - b. Apply timing constraints and power constraints.
  - c. Perform test-ready compile by using the `compile_ultra -scan` command.
  - d. Apply the DFT constraints.
  - e. Use the `insert_dft` command in the subblock so that the inserted scan chains can be included in the top-level scan chain.
  - f. Run the `compile_ultra -scan -incremental` command.
4. Use the following commands to write out the netlist, test model, and SCANDEF files.

```
write_test_model -format ddc -o core1_ins.ctlddc  
write_scan_def -o core1_ins.scandef  
write -format ddc -hier -o core1_ins.ddc  
write -format verilog -hier -o core1_ins.v
```

5. If you are using the topographical mode .ddc flow, no additional steps are required, But the other two flows require these additional steps:
  - In the topographical mode ILM flow, create the ILMs by using the `create_ilm` command and then save the ILMs to the database by using the `write -format ddc` command. For more information, see the Design Compiler user documentation.
  - In the IC Compiler ILM flow, after step 4, exit the run, invoke IC Compiler and create the physical block placements for the subblocks. You must use the `create_macro_fram` command to create the routing (FRAM) view of the design and the `create_ilm` command to generate the ILMs. For more information, see the IC Compiler User Guide.

**Top-level hierarchical flow.** To perform the top-level hierarchical flow

1. Set up the design and libraries.

2. Read in the top-level design, link the design, and apply the top-level constraints.
3. Read in the block-level netlists or ILMs, depending on the flow you are using.

For the IC Compiler ILM flow, use the ILMs you created in IC Compiler by adding the Milkyway design library that contains the ILM view to the Milkyway reference library list at the top level.

For the topographical ILM flow, read in the ILMs in .ddc format or add them to the `link_library` variable.

For the topographical .ddc flow, read in the .ddc files created in topographical mode or add them to the `link_library` variable. Also, use the `set_physical_hierarchy` command to specify that the subblocks should be treated as physical blocks. This prevents top-level synthesis from modifying the physical or logic structure of the blocks.

4. Run the `compile_ultra -scan` command.
5. Apply the DFT constraints.
6. Run the `insert_dft` command to insert scan chains at the top level, followed by `compile_ultra -scan -top` to map any additional unmapped logic that might have been introduced.
7. Write out the netlist and SCANDEF files.

Use the following commands to write out the netlist:

```
write -format ddc -hier -o top_ins.ddc
write -format verilog -hier -o-top_ins.v
```

For the ILM-based flows, remove all the block ILMs before generating the final top-level netlist.

Write out the SCANDEF by using the `write_scan_def` command.

- In the topographical mode, ILM and .ddc flows, the top-level SCANDEF file must directly reference the physical block scan leaf objects instead of the BITS construct to run the IC Compiler top-level flat flow. Use the `-expand_elements` option of the `write_scan_def` command to write out the SCANDEF information as follows:

```
write_scan_def -expand_elements block_instance -o top_ins.scandef
```

- In the IC Compiler ILM flow, the top-level SCANDEF contains the BITS construct, so use the `write_scan_def` command *without* the `-expand_elements` option. You must maintain the test hierarchy partition during top-level physical implementation in IC Compiler. Use the `write_scan_def` command as follows:

```
write_scan_def -o top_ins.scandef
```

---

## Scan Insertion Methodologies

DFT Compiler supports bottom-up and top-down scan insertion. To improve DFT Compiler performance on multimillion-gate designs, use the hierarchical scan synthesis (HSS) flow. The HSS flow reduces runtime and memory usage, and significantly increases the capacity of DFT Compiler.

For more information on HSS, see “[Hierarchical Scan Synthesis Flow](#)” on page 1-18.

This section discusses the following scan insertion methodologies:

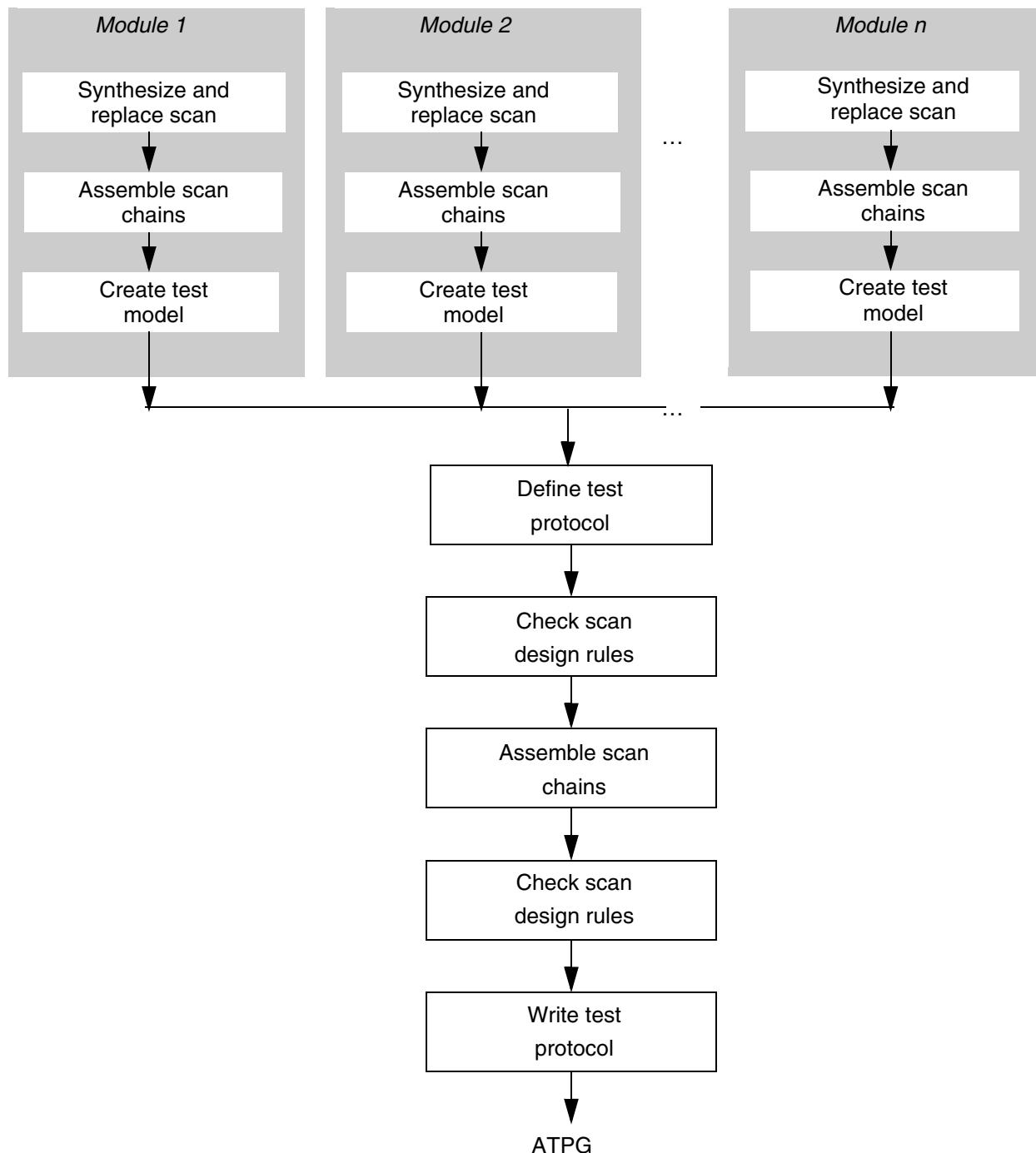
- [Bottom-Up Scan Insertion](#)
- [Top-Down Scan Insertion](#)

---

### Bottom-Up Scan Insertion

For bottom-up scan insertion, follow the flows outlined earlier in this chapter for each module of the design. For each subdesign, save a test model for the scan-inserted design. At higher levels, stitch together the test models instead of using the .ddc files containing the gate information of the subdesigns (see [Figure 1-13](#)).

Figure 1-13 Bottom-Up Scan Insertion Flow



To use the bottom-up scan insertion flow, follow these steps:

1. Synthesize your design, performing scan replacement.
2. Build scan chains in each module of the design.
3. Define the test protocol at the top level.

The test protocol provides information about your design to the test design rule checker.

4. Check design rules at the top level.

Evaluate the scan conformance of your design, and determine if any sequential cells violate the test design rules. Test design rule checking also highlights design issues that can lower your fault coverage results.

If the results of your analysis show that the design does not meet your requirements, you can often improve the results by modifying the test protocol. However, in some cases, you might have to modify the HDL description.

5. Assemble the scan chains.

After inserting scan cells in all modules, assemble the scan chains at the top level of the design. Use the same procedure for assembling scan chains at the top level that you used at the module level, and then run `dft_drc` to make sure the resulting scan chains operate properly.

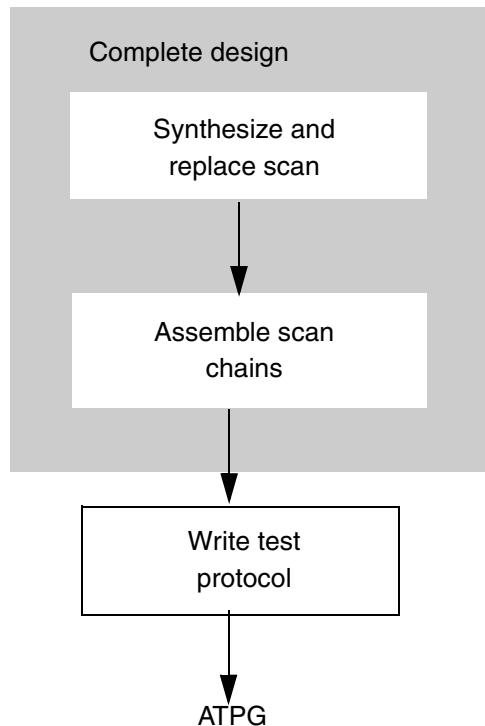
6. Write out the Verilog netlist and the test protocol, and then proceed to TetraMAX, DRC, and ATPG.

---

## Top-Down Scan Insertion

With top-down scan insertion, perform scan insertion only at the top level of the design (see [Figure 1-14](#)).

*Figure 1-14 Top-Down Scan Insertion*



To use the top-down scan insertion flow, follow these steps:

1. Synthesize your design, performing scan replacement.
2. Build scan chains at the top level of the design.
3. Write out the Verilog netlist and the test protocol, and then proceed to TetraMAX, DRC, and ATPG.

---

## DFT Compiler Default Scan Synthesis Approach

If you do not specify scan detail, the `insert_dft` command implements a default scan design by using the full-scan methodology. This section describes the ground rules that the `preview_dft` and `insert_dft` commands apply to generate a default scan design.

The section includes the following subsections:

- [Scan Replacement](#)
- [Scan Element Allocation](#)

- [Test Signals](#)
  - [Pad Cells](#)
  - [Area and Timing Optimization](#)
- 

## Scan Replacement

By default, DFT Compiler performs the following scan replacement tasks during `insert_dft`:

- Converts the scan elements that resulted from `compile -scan` or from a previous scan insertion back to nonscan elements if the test design rule violations prevent their inclusion in a scan chain.
  - Scan-replaces sequential elements if a scan replacement on the sequential elements was not performed previously.
- 

## Scan Element Allocation

DFT Compiler allocates scan elements to scan chains in the following manner:

- Allocates scan elements to produce the minimum number of scan chains consistent with clock domain requirements. By default, the `insert_dft` command generates a scan design with the number of scan chains being equal to the number of clock domains. The resulting design contains one scan chain for each set of sequential elements clocked by the same edge of the same test clock.
  - Automatically infers existing scan chains both in the current design and in subdesigns. This is true only if the design has the proper attributes.
  - Does not reroute existing scan chains built by the `insert_dft` command or subdesign scan chains built by the `insert_dft` command, even if the existing routing does not conform to default behavior.
  - Orders scan elements in scan chains alphanumerically. By default, the `insert_dft` command alphanumerically orders scan elements within scan chains across the full hierarchical path specification of the scan element name.
- 

## Test Signals

DFT Compiler inserts and routes test signals in the following manner:

- Automatically inserts and routes global test signals to support the specified scan style. These test signals include clocks and enable signals.

- Allocates ports to carry test signals. Where possible, the `insert_dft` command uses “mission” ports (that is, normal function ports) to carry scan-out ports and inserts multiplexing logic, if required. The `insert_dft` command performs limited checking for existing multiplexing logic to prevent redundant insertion.
  - Inserts three-state and bidirectional disabling logic during default scan synthesis. The `insert_dft` command checks for existing disabling logic to prevent redundant insertion.
- 

## Pad Cells

If the current design includes pad cells, the `insert_dft` command identifies the pad cells and correctly inserts test structures next to them by

- Ensuring correct core-side hookup to all pad cells and three-state drivers
- Inserting required logic to force bidirectional pads carrying scan-out signals into output mode during scan shift
- Inserting required logic to force bidirectional pads carrying scan-in, control, and clock signals into input mode during scan shift
- Determining requirements and, if necessary, inserting required logic to force all other nondegenerated bidirectional ports into input mode during scan shift
- Inserting required logic to enable three-state output pads associated with scan-out ports during scan shift
- Inserting required logic to disable three-state outputs that are not associated with scan-out ports during scan shift

---

## Area and Timing Optimization

By default, the `insert_dft` command uses constraint-optimized scan insertion to reduce the scan-related impact on performance and area. This process minimizes constraint violations and eliminates compile design-rule errors. For consistency with Design Compiler, `insert_dft` uses the clock waveforms described by the `create_clock` command to determine whether a logic path meets performance constraints. The `insert_dft` command does not use the timing values described by using `set_dft_signal` for constraint optimization. The `insert_dft` command also selects scan-out signal connections (Q or QN) to minimize constraint violations.

Scan chain synthesis is concerned primarily with the scan shift operation. The `dft_drc` command identifies problems associated with scan capture that might require you to resolve problems caused by functional clock waveforms.

Scan chains synthesized by the `insert_dft` command are functional under zero-delay assumptions. Before you perform scan synthesis, you can specify test clocks, using the `create_clock` command. All test clocks have the same period.

---

## Getting the Best Results With Scan Design

To get the best scan design results, your ATPG tool must be able to control the inputs and observe the outputs of individual cells in a circuit. By observing all the states of a circuit (complete fault coverage), the ATPG tool can check whether the circuitry is good or faulty for each output. The quality of the fault coverage depends on how well a device's circuitry can be observed and controlled.

If the ATPG tool cannot observe the states of individual sequential elements in the circuit, fault coverage is lowered because the distinction between a good circuit and a faulty circuit is not visible at a given output.

To maximize your fault coverage, follow these recommendations:

- Use full scan.
- Fix all design rule violations.
- Follow these design guidelines:
  - Be careful when you use gated clocks. If the clock signal at a flip-flop or latch is gated, a primary clock input might not be able to control its state. If your design has extensive clock gating, use AutoFix or provide another way to disable the gating logic in test mode.

Note:

DFT Compiler supports gated-clock structures inserted by the Power Compiler tool.

- Generate clock signals off-chip or use clock controllers compatible with DFT Compiler. If uncontrollable clock signals are generated on-chip, as in frequency dividers, you cannot control the state of the sequential cells driven by these signals. If your design includes internally generated, uncontrollable clock signals, use AutoFix or provide another way to bypass these signals during testing.
- Minimize combinational feedback loops. Combinational feedback loops are difficult to test because they are hard to place in a known state.
- Use scan-compatible sequential elements. Be sure that the library you select has scannable equivalents for the sequential cells in your design.

- Avoid uncontrollable asynchronous behavior. If you have asynchronous functions in your design, such as flip-flop preset and clear, use AutoFix so that you can control the asynchronous pins or make sure you can hold the asynchronous inputs inactive during testing.
- Control bidirectional signals from primary inputs.

The scan design technique does not work well with certain circuit structures, such as

- Large, nonscan macro functions, such as microprocessor cores
- Compiled cells, such as RAM and arithmetic logic units
- Analog circuitry

For these structures, you must provide a test method that you can integrate with the overall scan-test scheme.

---

## DFT Compiler and Power Compiler Interoperability

This section discusses issues associated with the interoperability of DFT Compiler and Power Compiler. It includes the following subsections:

- [Improving Testability in Clock Gating](#)
  - [Power Compiler/DFT Compiler Interoperability Flows](#)
  - [Connecting Test Pins to Clock Gating Cells Using insert\\_dft](#)
- 

### Improving Testability in Clock Gating

Clock gating raises some concerns for ATPG. These concerns involve enabling clock controllability and scan testing as well as optimizing ATPG results.

DFT Compiler might not include a clock-gated register in a scan chain during scan synthesis if gating the register clock makes it uncontrollable for test. When the register is excluded from the scan chain, test controllability is reduced at the register input and test observability is reduced at the register output. If many registers are gated, the fault coverage can be reduced significantly.

If you add control points during clock gating, you can enable scan testing, thus improving the testability of the design. However, by controlling the clock signal, you might introduce untestable faults into the design. To optimize the ATPG results, you might need to insert observation points during clock gating.

This section includes the following subsections:

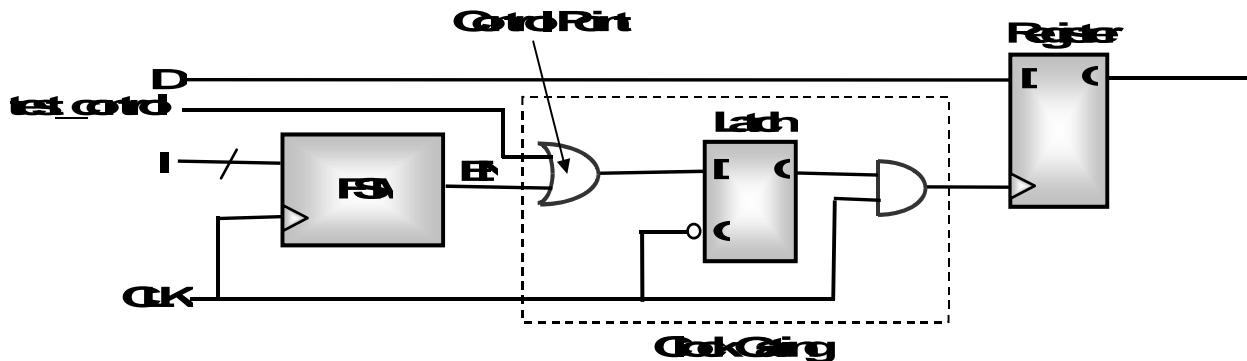
- [Inserting Control Points in Control Clock Gating](#)
- [Scan Enable Versus Test Mode](#)
- [Inserting Observation Points to Control Clock Gating](#)
- [Choosing a Depth for Observability Logic](#)
- [Hooking Up Test Ports Through Hierarchies in Power Compiler](#)

## Inserting Control Points in Control Clock Gating

A control point is an OR gate driven by a control signal. This signal eliminates the gating of the clock signal during test, thus restoring the controllability of the clock signal.

[Figure 1-15](#) shows a control point (OR gate) placed before a latch connected to a `test_control` port. When the `test_control` signal is high, it overrides clock gating, thus making the gated clock and CLK signals identical.

*Figure 1-15 Control Point in Gated Clock Circuitry*



You can place the control point OR gate in front of or behind the latch. Latch-based clock gating requires that the enable signal always arrives after the trailing edge of the clock signal or the rising edge for a falling-edge clock signal. If the control point is inserted before the latch, it is impossible for the control point to violate this requirement. If it is inserted behind the latch, it benefits the circuit by adding an extra hold-time margin for the clock glitch hold check. It does this by delaying the data signal, which helps the clock pulse arrive ahead of the data.

Inserting the control point after the latch causes performance degradation because a gate is added between the latch and the register. In addition, the `test_control` signal must then transition after the trailing edge (rising edge for falling-edge signal) of the clock signal during test because it does not go through the latch; otherwise glitches in the resulting signal corrupt the clock signal.

You can choose to make the `test_control` port either a `scan_enable` signal or a `test_mode` signal. The `scan_enable` signal is active only during scan shifting. The `test_mode` signal is active during the entire test.

The `set_clock_gating_style` command has two options for determining the location and type of the control point for test:

- The `-control_point` option can be `none`, `before`, or `after`. The default is `none`. When you use a latch-free clock-gating style, the `before` and `after` options are equivalent.
- The `-control_signal` option can be `test_mode` or `scan_enable`. The default is `scan_enable`. This option uses an existing test port or creates a new test port and connects the test port to the control point OR gate.

You can use the `-control_signal` option only if the `-control_point` option is used. For existing test ports, this option relies on the `set_dft_signal` command for the `test_mode` signal and for `scan_enable`.

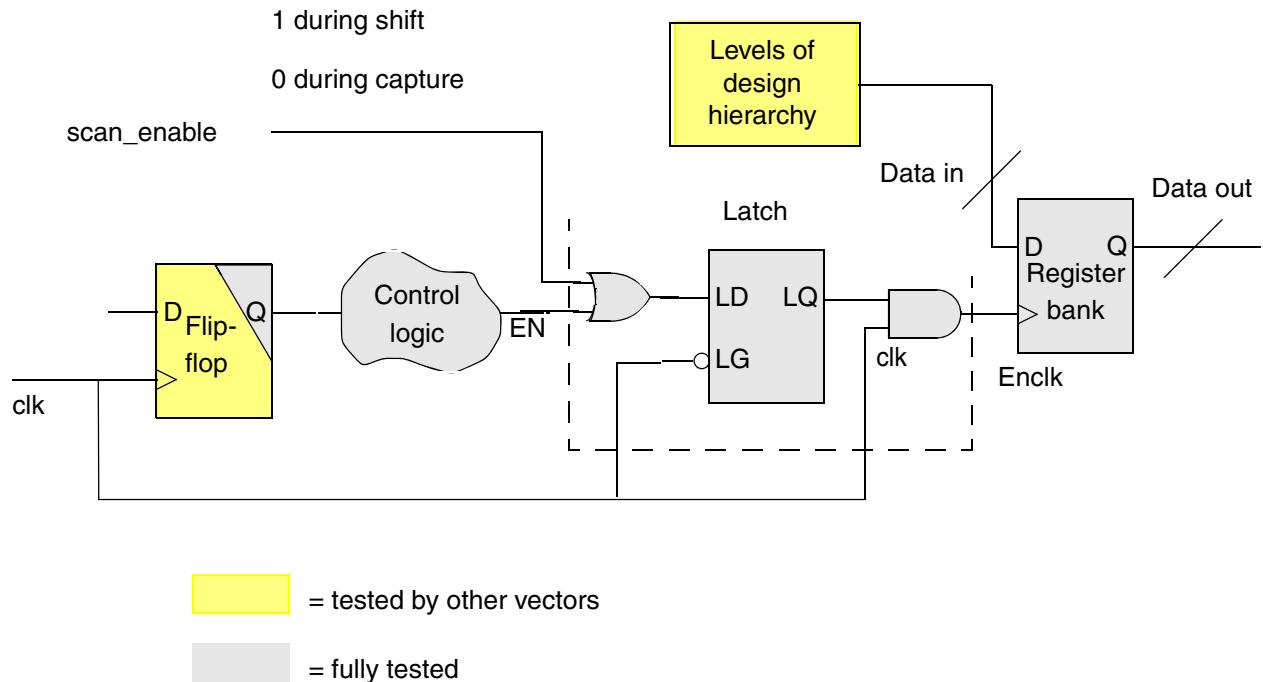
## Scan Enable Versus Test Mode

Scan enable and test mode differ in the following ways:

- Scan enable is active only during scan mode.
- Test mode is active during the entire test (scan mode and parallel mode).

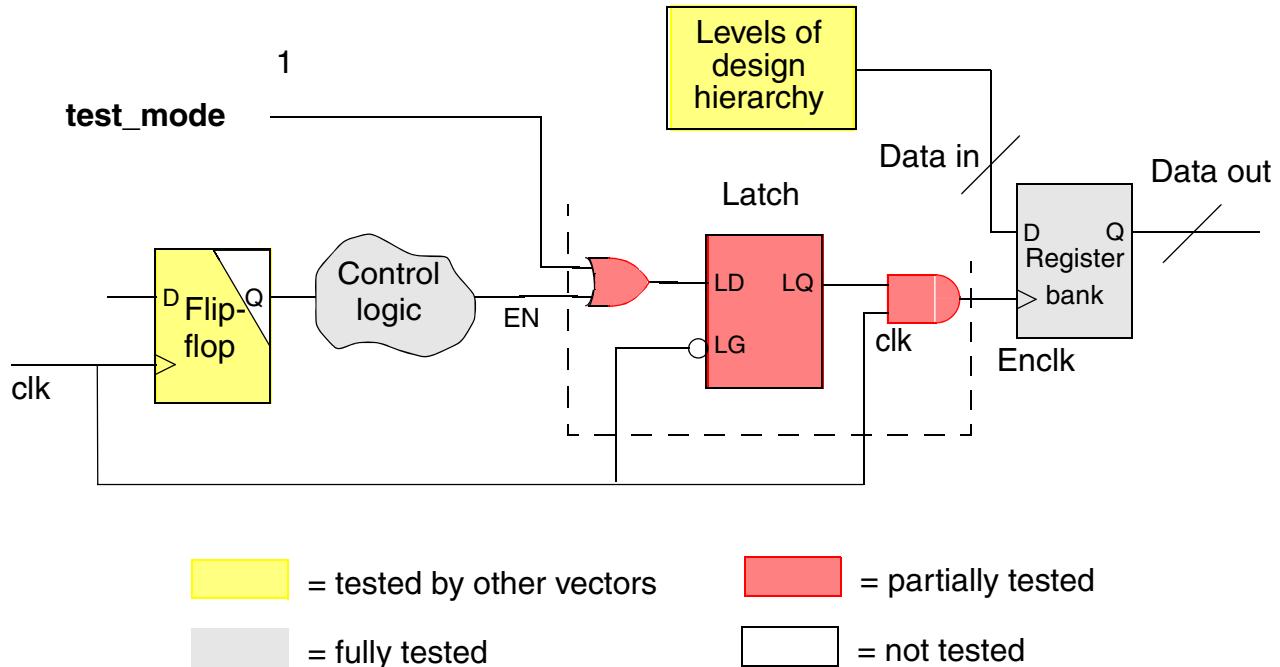
The `scan_enable` signal typically provides higher fault coverage than the `test_mode` signal. Fault coverage with `scan_enable` is comparable to a circuit without clock gating, as shown in [Figure 1-16](#). During the ATPG capture mode, the register bank is clocked whenever the output of the flip-flop is a 1; this behavior matches functional operation.

Figure 1-16 Test Coverage With Test Scan Enable



In some situations, you must use a `test_mode`. The control logic preceding the clock-gating circuitry is not tested, as shown in [Figure 1-17](#). In addition, the clock-gating logic can be tested only for “stuck-at-1” faults. During test, the `test_mode` signal is always a logic 1, which forces the clock at the register bank to switch at all times. One way to test the clock-gating logic circuitry is to add observability logic to the design.

Figure 1-17 Test Coverage With `test_mode`

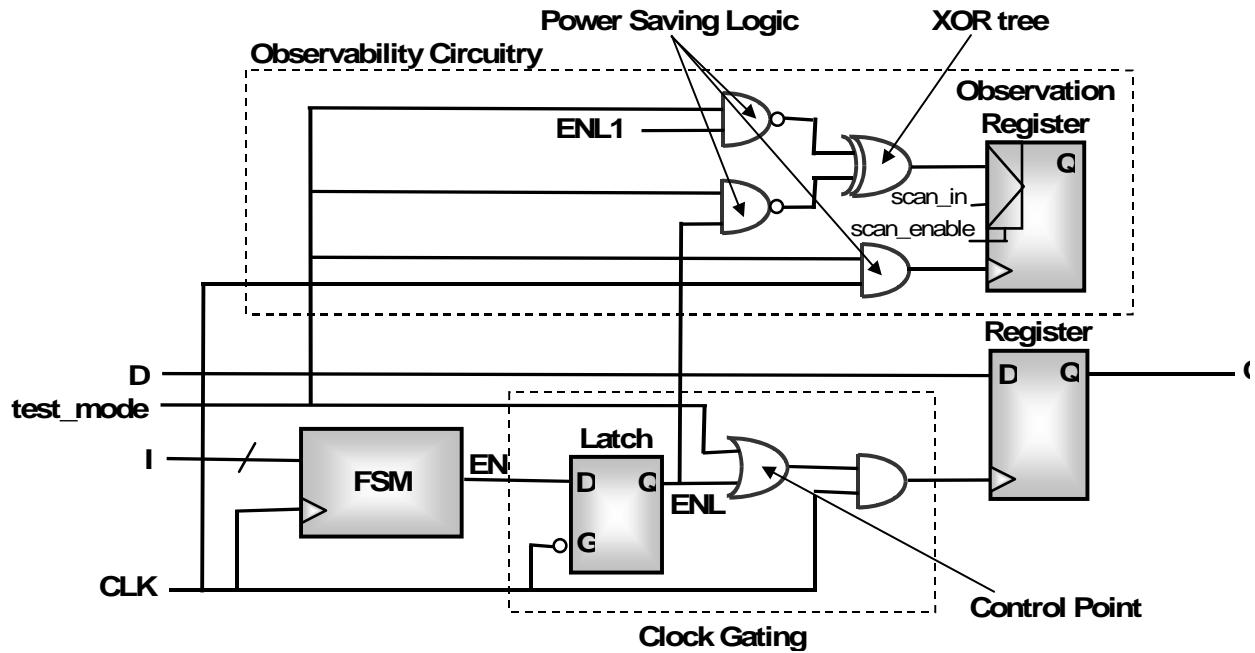


## Inserting Observation Points to Control Clock Gating

When you use the `test_mode` signal, the `EN` signal and other signals in the control logic are untestable. You can add observation points during clock gating to increase the fault coverage.

[Figure 1-18](#) shows clock-gating circuitry, including an XOR tree and a scan register, to observe the `ENL` and `ENL1` signals, which are outputs of the clock-gating latches.

Figure 1-18 Observability Circuitry in Clock-Gating Circuits



During testing, observability circuitry allows observation of the ENL and ENL1 signals. In normal operation of the circuit, the XOR tree does not consume power, because the NAND gates block all signal transitions of the ENL and ENL1 signals.

The insertion of observation test points has high testability and is power-efficient because the XOR tree consumes power only during test and the clock of the observation register is also gated.

The `set_clock_gating_style` command has two options for increasing observability when you use the `-control_signal test_mode` option:

- The `-observation_point` option can be `true` or `false`. The default is `false`. When you set this option to `true`, the `set_clock_gating_style` command adds a cell that contains at least one observation register and an appropriate number of XOR trees. The scan chain includes the observation register. The output of the observation register is not functionally connected to the circuit.
- The `-observation_logic_depth` option allows a `depth_value` specification. The default `depth_value` is 5. The value determines the depth of the XOR tree.

## Choosing a Depth for Observability Logic

Use the `-observation_logic_depth` option of the `set_clock_gating_style` command to set the logic depth of the XOR tree in the observability cell. The default for this option is 5.

Clock-gating software builds one observability cell for each clock-gated design. Each gated register in the design provides a gated enable signal as input to the XOR tree in the observability cell.

If you set the logic depth of your XOR tree too small, clock gating creates more XOR trees and associated registers to provide enough XOR inputs to accommodate signals from all the gated registers. Each additional XOR tree adds some overhead for area and power. Using one XOR tree adds the least amount of overhead to the design.

If you set the logic depth of your XOR tree too high, clock gating can create one XOR tree with plenty of inputs. However, too large a tree can cause the delay in the observability circuitry to become critical.

Use a value that meets the following two criteria in choosing or changing the XOR logic tree depth:

1. High enough to create the fewest possible XOR trees
2. Low enough to prevent critical delay in the observability circuitry

## Hooking Up Test Ports Through Hierarchies in Power Compiler

In Power Compiler, use the `hookup_testports` command to connect test ports through all the levels of hierarchy to the `test_mode` or `scan_enable` pins of the OR gate in the clock-gating logic and to the XOR gates in the clock-gating observability logic. If your design does not have a test port at any level of hierarchy, Power Compiler will create one. If a test port exists, it will be used.

Use the `hookup_testports` command on the top level of the design after all lower levels have been elaborated. It connects to two test port types:

- `test_mode`
- `scan_enable`

You can use the `clock_gate_test_pin` attribute to designate a test port. In this case, label it as a `scan_enable` or a `test_mode` signal by using the `set_dft_signal` or `set_attribute` commands, respectively.

A `scan_enable` port can be connected only to other `scan_enable` ports in the design hierarchy. If a `scan_enable` port does not exist at a higher level of hierarchy, Power Compiler will create one.

The design in [Example 1-4](#) uses the `hookup_testports` command on a three-level hierarchy. Note that the lowermost level is clock gated with a `scan_enable` control point. In this case, `scan_enable` ports are created at all levels, `low_design`, `med_design`, and `top_design`, if they do not already exist. They are then connected to the `scan_enable` pin on the clock-gating logic cell. If the ports are available on one or more designs, Power Compiler connects them to each other.

*Example 1-4 Hooking Up Test Ports in Power Compiler*

```
dc_shell> set_clock_gating_style -control_point after \
           -control_signal scan_enable

dc_shell> analyze -f vhdl low.vhdl med.vhdl top.vhdl

dc_shell> elaborate low_design

dc_shell> elaborate med_design

dc_shell> elaborate top_design

dc_shell> insert_clock_gating

dc_shell> hookup_testports -verbose
```

[Example 1-5](#) uses a `test_mode` port instead of a `scan_enable` port. Because an observation circuit is included, the `test_mode` port of the observation circuit is connected to the rest of the `test_mode` ports.

*Example 1-5 Hooking Up Test Mode Ports in Power Compiler*

```
dc_shell> set_clock_gating_style -control_point before \
           -control_signal test_mode

dc_shell> read_vhdl "low.vhdl med.vhdl top.vhdl"

dc_shell> current_design low_design

dc_shell> current_design top_design

dc_shell> insert_clock_gating

dc_shell> hookup_testports -verbose
```

[Example 1-6](#) directs `hookup_testports` to use an existing port “te” of module `low_design` as a `scan_enable`.

*Example 1-6 Hooking Up Scan Enable Ports by Using an Existing Port*

```
dc_shell> set_clock_gating_style -control_point before \
           -control_signal scan_enable

dc_shell> analyze -f vhdl low.vhdl
```

```

dc_shell> elaborate low_design

dc_shell> set_dft_signal -view spec -port te \
    -type ScanEnable -active_state 1

dc_shell> insert_clock_gating

dc_shell> hookup_testports -verbose

```

**Example 1-7** directs `hookup_testports` to use an existing port “te” of module `low_design` as a `scan_enable` by using alternative commands.

*Example 1-7 Hooking Up Scan Enable Ports by Using an Existing Port*

```

dc_shell> set_clock_gating_style -control_point before \
    -control_signal scan_enable \
    -observation_point true

dc_shell> analyze -f vhdl low.vhdl

dc_shell> elaborate low_design

dc_shell> set_dft_signal -view existing_dft -port te \
    -type ScanEnable -active_state 1

dc_shell> set_attribute te test_port_clock_gating \
    -type boolean true

dc_shell> insert_clock_gating

dc_shell> hookup_testports -verbose

```

**Example 1-8** directs `hookup_testports` to use an existing port “te” of module `low_design` for `test_mode`.

*Example 1-8 Hooking Up Test Mode Ports by Using an Existing Port*

```

dc_shell> set_clock_gating_style -control_point before \
    -control_signal test_mode \
    -observation_point true

dc_shell> analyze -f vhdl low.vhdl

dc_shell> elaborate low_design

dc_shell> set_dft_signal -view spec -type TestMode \
    -port te

dc_shell> insert_clock_gating

dc_shell> hookup_testports -verbose

```

**Example 1-9** directs `hookup_testports` to use an existing port “te” of module `low_design` for `test_mode` by using alternative commands.

### *Example 1-9 Hooking Up Test Ports by Using an Existing Port*

```
dc_shell> set_clock_gating_style -control_point before \
           -control_signal test_mode \
           -observation_point true

dc_shell> analyze -f vhdl low.vhdl

dc_shell> elaborate low_design

dc_shell> set_dft_signal -view existing_dft -type \
           Constant -active_state 1 -port te

dc_shell> set_attribute te test_port_clock_gating \
           -type boolean true

dc_shell> insert_clock_gating

dc_shell> hookup_testports -verbose
```

---

## **Power Compiler/DFT Compiler Interoperability Flows**

This section suggests methodologies that can help you avoid interoperability problems between Power Compiler and DFT Compiler. You can apply these methodologies directly within the current Power Compiler/DFT Compiler flow.

### **Using `test_mode` With Power Compiler**

The `-control_signal test_mode` option of the `set_clock_gating_style` command relies on the `signal_type` or `test_hold` attribute to stitch an existing `test_mode` signal to the clock gating control point.

Power Compiler identifies the `test_mode` signal in the following manner:

1. Power Compiler looks for an existing `test_mode` signal specified with the `set_dft_signal` command. You can set the `-type` to be either `active_state 1` or `active_state 0`.
2. If a `test_mode` signal has not been specified with the `set_dft_signal` command, Power Compiler looks for an existing port with a `test_76hold` attribute, which can then be set with the `set_dft_signal` command.
3. If no port has been identified in steps 1 and 2, Power Compiler creates a new `test_mode` port during clock gating.

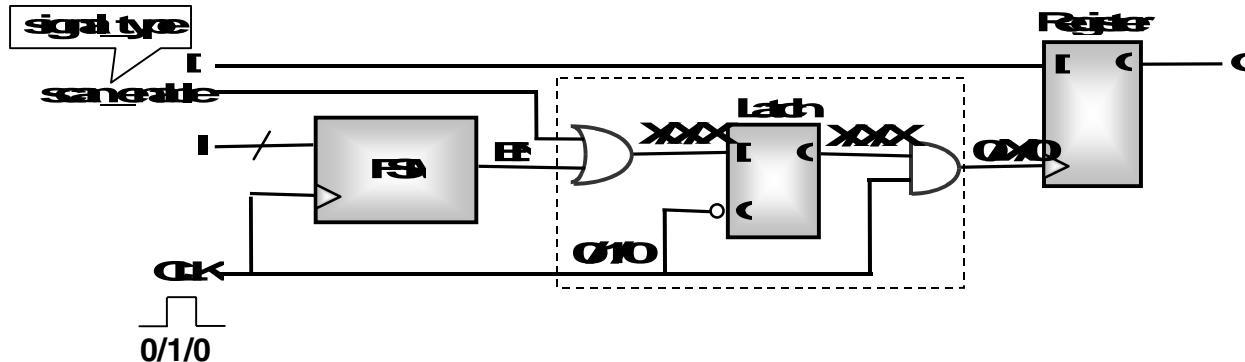
## Using Scan Enables With Power Compiler

To achieve high-fault-coverage in clock-gated designs, the scan chain must be shifted when `scan_enable` is asserted. The easiest way to ensure this is to disable all gating elements when `scan_enable` is asserted. You can achieve this by placing an OR gate or a NOR gate on the EN input, which forces EN on when `scan_enable` is asserted.

In parallel capture mode, the scan chain must also be clocked to ensure that data is captured in the scan registers. However, using `scan_enable` to control latch-based and latch-free clock gating might not guarantee this. Furthermore, using the signal to control latch-free clock gating might make the clock pin of the clock-gated registers uncontrollable.

In [Figure 1-19](#), the latch-based clock gating does not have violations that prevent scan insertion. The clock pin is correctly forced to a known state at time = 0, and it correctly changes state as a result of the test clock toggling (toggling to an unknown state X is a change of state). Test DRC verifies that the test clock pulse arrives at the test clock pin to check the scan shift process. Because the `scan_enable` port is constrained to logic state 1 during this process, the clock pin is controllable.

*Figure 1-19 Violation-Free Clock Gating*



[Table 1-2](#) shows the possible combinations of latch-based clock gating, clock waveforms, control signals and control point location. For each combination, the last column indicates whether scan insertion can be performed on the clock-gated register.

*Table 1-2 Latched-Based Clock-Gating Configurations*

Clock gating	CLK	Control signal	Control point location	Register can-inserted?
High Latch-based		test_mode	Before latch	Yes
			After latch	Yes
		scan_enable	Before latch	Yes
			After latch	Yes
		test_mode	Before latch	Yes**
			After latch	Yes
Low Latch-based		test_mode	Before latch	Yes
			After latch	Yes
		scan_enable	Before latch	Yes
			After latch	Yes
		test_mode	Before latch	Yes**
			After latch	Yes
High Latch-based		scan_enable	Before latch	Yes
Low Latch-based		scan_enable	Before latch	Yes
After latch			No	
After latch			Yes	
After latch			No	

**Table 1-2** describes the configurations that can be used in the Power Compiler/DFT Compiler flow. Two special cases (marked by \*\* in **Table 1-2**) require you to modify the test protocol.

Consider the case of high latch-based clock gating controlled by a `test_mode` and driven by a return-to-1 clock. If the control point is inserted before the latch, then when the clock port is off (time = 0), the latch is blocked and the clock pin is not controllable. This violation also occurs in the case of a low latch-based clock gating controlled by a `test_mode` and driven by return-to-0.

To achieve a known state in the latch, add a clock pulse to the `test_setup` section of the test protocol. Use the following `set_dft_drc_configuration` command so that DFT Compiler automatically updates the protocol:

```
set_dft_drc_configuration -clock_gating_init 1
```

If you have multiple cascaded latches and the first latch is loaded with `test_mode`, use the following `set_dft_drc_configuration` command so that the test protocol is automatically updated with the number of clock pulses in the `test_setup` section:

```
set_dft_drc_configuration -clock_gating_init n
```

Here, `n` = the number of clock pulses required to initialize clock gating latches.

Note the following:

- The set of clock cycles should equal the depth of the chain of latches. Make sure this is the case.
- Violations still occur when there are multiple cascaded latches and the scan-enable and control point location are used as before, with mixed active-high and active-low latches.

Consider a latch state that is known, as in the case of a high latch-based clock gating controlled by a `scan_enable` with a return-to-1 clock. When the clock port is off (time = 0), the register does not hold its state. During pre-DFT DRC capture check, the EN signal goes through the control point and the system clock pulse might not arrive at the register clock pin. Because the register might receive invalid data during the capture phase of ATPG, it will not be scan-inserted. The same DRC violations occur for the low latch-based clock gating controlled by `scan_enable` with a return-to-0 clock.

### Latch-Free Clock-Gating Configurations

[Table 1-3](#) shows combinations of latch-free clock gating, clock waveforms, and control signals. For each combination, the last column indicates when the clock-gated register can be scan-inserted and when it cannot.

*Table 1-3 Latch-Free Clock-Gating Configurations*

Clock Gating	CLK	Control signal	Control point location	Register scan-inserted?
High Latch-based		test_mode	Before	Yes
		scan_enable	Before	Yes
		test_mode	Before	Yes
		scan_enable	Before	Yes

Table 1-3 Latch-Free Clock-Gating Configurations (Continued)

Clock Gating	CLK	Control signal	Control point location	Register scan-inserted?
Low Latch-based		test_mode	Before	Yes
		scan_enable	Before	Yes
		test_mode	Before	Yes
		scan_enable	Before	Yes

## Connecting Test Pins to Clock Gating Cells Using `insert_dft`

You can use the `insert_dft` command to connect clock gating cells to the test ports.

Prior to this capability, you first had to use the Power Compiler `hookup_testports` command to connect the test ports on all levels of the design hierarchy to the `test_mode` or `scan_enable` pins of the OR gate in the clock gating logic, the XOR gates in the clock gating observability logic, and the pins on integrated clock gating cells. You would then use the `insert_dft` command to construct the remaining DFT logic, such as scan chains and test points.

With the `hookup-test-ports` capability, the `insert_dft` command will make clock gating cell connections as well as carry out the construction of DFT logic in one step. The `insert_dft` command makes the connections of the top level test ports to the test pins of the clock gating cells through the hierarchy. If the design does not have a test port at any level of hierarchy, a new test port is created. If a test port exists, it is used.

Note:

In the B-2008.09 release, the `hookup_testports` command is also available, but it is recommended that you do not use this command if you intend to use the new DFT feature. However, if you do use the `hookup_testports` command and the new feature is enabled, `insert_dft` will honor and preserve any clock gating connections made by `hookup_testports` and create only the missing connections.

This feature is enabled by default. You can control whether this functionality is enabled by using the following command:

```
set_dft_configuration -connect_clock_gating <enable|disable>
```

After you have inserted the clock gating cells, if this functionality is enabled, you only need to use the `insert_dft` command to connect test ports in a flow similar to the following:

```
read_ddc design.ddc
set_clock_gating_style -control_signal test_mode
compile_ultra -scan -gate_clock
set_dft_signal -type TestMode -port test_mode
. . .
create_test_protocol
preview_dft
insert_dft
```

You can use the `report_dft_configuration` command to report what was previously set and the `reset_dft_configuration` command to restore the setting to the default value.

All existing DFT flows except those mentioned in “[Limitations](#)” on page 66 are supported.

## Design Requirements

For the feature to work, the design must have the clock gating cells inserted before you run `insert_dft`. Insertion of clock gating cells can be done either by using the `compile_ultra -gate_clock` command or the `insert_clock_gating` command. The design must have the clock gating attributes present in order for `dft_drc` and `insert_dft` to recognize them as valid clock gating cells. The attributes help to distinguish between a design that has an erroneous connectivity for which DRC should flag violations and a design that is correct but for which the connectivity is yet to be established by `insert_dft`.

If you start with the `.ddc` format, the clock gating cell attributes will generally be present. However, if clock gating cell attributes are not present (for example, if you start with a Verilog netlist), you need to ensure that the required attributes are present for the clock gating cells so that the `dft_drc` and `insert_dft` commands can recognize them. You can do this by using the `identify_clock_gating` command or by following a Power Compiler recommended flow and manually identifying the clock gating cells.

## Hookup Testport Connections

The following table describes the connections that `insert_dft` makes:

*Table 1-4 Connections Made to the Clock Gating Cells by insert\_dft*

	Clock gating control signal	DFT command	Top-level port used
1	<code>scan_enable</code>	<code>No set_dft_signal defined with -type ScanEnable</code>	<code>test_se</code> created and connected to test pin of clock gating cell.
2	<code>scan_enable</code>	<code>set_dft_signal -view spec exist -type ScanEnable -port test_se -active_state 0 1</code>	No new port created. <code>test_se</code> used to connect to test pin of clock gating cell.
4	<code>test_mode</code>	<code>No set_dft_signal defined with -type TestMode</code>	New port. <code>test_cgtn</code> created to connect to test pin of clock gating cell.
5	<code>test_mode</code>	<code>set_dft_signal -view spec exist -type TestMode -port test_mode -active_state 0 1</code>	No new port created. <code>test_mode</code> used to connect to test pin of clock gating cell.

Note:

The DFT signal specifications intended for clock gating cell connections are not mode specific. Therefore you cannot specify a `-test_mode` for the `set_dft_signal` command.

## Design Rule Checking Changes

The test pin connections of valid clock gating cells, that is, those identified with Power Compiler clock gating attributes, are checked during `dft_drc`.

Once the clock gating cells are identified, `dft_drc` performs the following checks:

1. If the test pin is already connected, no violation will be obtained during `dft_drc`, and the flip-flops controlled by the clock gating cell are put onto the scan chain during `insert_dft` provided no other violations exist for the cell.
2. If the test pin is not connected, `dft_drc` will generate the following message for the clock gating cells:

Warning: Clock gating cell %s has unconnected test pin. (TEST-130)

This warning is issued irrespective of whether `set_dft_configuration -connect_clock_gating` is enabled or disabled.

For all cells controlled by clock gating cells that are flagged with TEST-130 violations for which `set_dft_configuration -connect_clock_gating` is set to disable, `dft_drc` will not report that flip-flops driven by these clock gating cells will have clock violations (D1 or D9) and will not be included in scan chains (unless Autofix is enabled).

For all cells controlled by clock gating cells that are flagged with TEST-130 violations for which `set_dft_configuration -connect_clock_gating` is set to enable, `dft_drc` will not report that these flip-flops as having clock violations and will stitch these flip-flops into the scan chains.

---

## Specifying a Particular Signal as Test Pin When Automatically Connecting Test Ports to Clock-Gating Cells

Although the preceding sections describe how to connect the test pins to the clock-gating cells without specifying a particular signal when you run the `insert_dft` command, this is not a limitation. You can choose to specify a particular signal.

### Specifying a Particular Signal as Test Pin for Clock-Gating Cells

Starting with DFT Compiler version 2008.09-SP1, you can use the new `-usage clock_gating` option and argument to the `set_dft_signal` command to define the specified signal as a dedicated test signal that is to be connected to the test pins of the clock-gating cells. When you specify the `-usage clock_gating` option of the `set_dft_signal` command, the `insert_dft` command is limited to using only that signal to connect the test pin of a clock-gating cell.

For the `set_dft_signal` command, the `clock_gating` argument of the `-usage` option is only valid with `-type ScanEnable` or `TestMode` and with `-view spec` specified, as shown in the following:

```
set_dft_signal -usage clock_gating -type [ScanEnable|TestMode] \
               -view spec -port <name_of_port>
```

If there are an insufficient number of `ScanEnable` or `TestMode` signals for other purposes, DFT Compiler creates additional `ScanEnable` or `TestMode` signals as needed.

### Associating a Particular ScanEnable or TestMode Signal as the Test Pin for Specified Clock-Gating Cells

The following command allows you to specify a `ScanEnable` or `TestMode` signal to be connected to the test pin of the specified clock-gating cells:

```
set_dft_connect <LABEL> \
```

```

-type clock_gating_control \
-source <name of driving DFT signal port or pin> \
-target <valid clock-gating cells | design names | clock_list> |
-exclude <list or collection of excluded objects>

```

**LABEL** specifies the name for the connectivity association. The specification is not cumulative; that is, a second **LABEL** specification overrides the first one.

With respect to the **-type** option, the `clock_gating control` argument is required for clock-gating connections.

Also, **-source** is a required option. It specifies the ScanEnable or TestMode pin or port that is to be connected to the test pin of the clock-gating cells. The specified pin or port name must be previously defined as a ScanEnable or TestMode by using the `set_dft_signal` command with the **-usage clock\_gating** option and argument.

Using the **-target** option, you can specify the list of clock-gating cells or design names for which the connection is to be made.

The **-exclude** option allows you to specify a list of clock-gating cell names or design names that are not to be connected to the test pin or port specified by the **-source** option.

For example, to use `test_se` as the signal that connects to the test pin of clock-gating cells CGC1 and CGC2, you can specify the following;

```

set_dft_signal -type ScanEnable -view spec -port test_se \
               -usage clock_gating
set_dft_connect CON1 -source test_se -type connect_clock_gating \
                 -target [list CGC1 CGC2]

```

You can use `report_dft_connect [LABEL]` to report the DFT association that was specified with the `set_dft_connect` command, and you can use `remove_dft_connect [LABEL] [-all]` to remove a particular specification.

### **Associating a Particular ScanEnable or TestMode Signal as the Test Pin for Clock-Gating Cells Based on Clock Domain**

The `set_dft_connect` command and options are also used for specifying the connection of a particular ScanEnable or TestMode of the clock-gating cells based on clock domains. You need to specify a clock list for the **-target** option to the `set_dft_connect` command. The clock list must be based on the name defined in the `create_clock` command.

For example,

```

set_dft_signal -type ScanClock -view exist -port clk1 -timing {45 55}
set_dft_signal -type ScanEnable -view spec -port scan_en -usage clock_gating
set_dft_connect CON1 -source scan_en -target clk1 -type connect_clock_gating

```

---

## **Limitations**

Note the following limitations:

- In this release, this new functionality in `insert_dft` is similar to the functionality of the `hookup_testports` command in the A-2007.12 release of Power Compiler except for connecting clock-gating cells before the design is mapped, that is, running `hookup_testports` when the design is still at the RTL stage.
- Only the clock-gating cells recognized by Power Compiler are supported. The methodologies recommended for Power Compiler must be followed to enable successful recognition of clock-gating cells and their test pins.
- clock-gating cell connection is not mode-specific in this release.
- Reporting connections made to clock-gating cells by `insert_dft` or `preview_dft` is not available in this release.

# 2

## Running RTL Test Design Rule Checking

This chapter describes how to prepare for and run RTL test design rule checking (DRC) and analyze DRC violations.

The RTL test design rule checking (DRC) process provides early warnings of test-related issues. This feedback is crucial because it provides you with an opportunity to correct your RTL code before the compile phase of the design flow. By correcting these problems during this stage, you can reduce time-consuming iterations that would occur later in the design process.

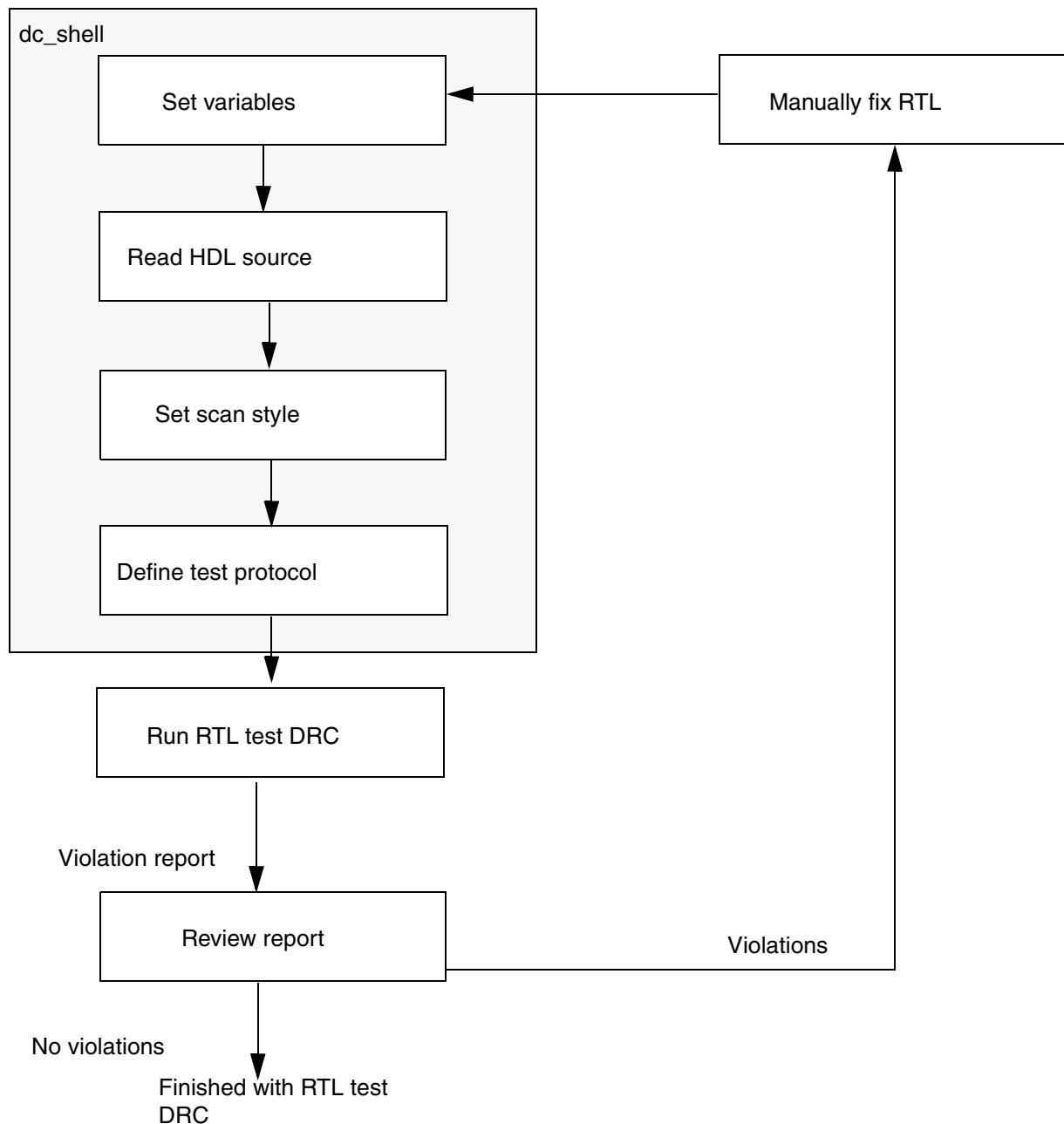
This chapter includes the following sections:

- [Understanding the Flow](#)
- [Specifying Setup Variables](#)
- [Generating a Test Protocol](#)
- [Running RTL Test DRC](#)
- [Understanding the Violations](#)

## Understanding the Flow

Figure 2-1 shows a typical RTL test DRC flow.

Figure 2-1 RTL Test DRC Design Flow



---

## Specifying Setup Variables

To begin preparing for RTL test DRC checking, you need to specify a series of setup variables, as described in the following steps:

1. Set the `hdlin_enable_rtldrc_info` variable to `true`. This variable reports file names and line numbers associated with each violation, which makes it easier for you to later edit the source code and fix violations.

```
dc_shell> set hdlin_enable_rtldrc_info true
```

2. Make sure you define the list of searched logical libraries by using the `link_library` variable.

3. Read in your HDL source code by using the `read` variable. The following variable reads in a Verilog file called `my_design.v`:

```
dc_shell> read_file -format verilog my_design.v
```

---

## Generating a Test Protocol

A test protocol is required for specifying signals and initialization requirements associated with design rule checking. This section has the following subsections related to generating a test protocol:

- [Defining a Test Protocol](#)
- [Setting the Scan Style](#)
- [Design Examples](#)

---

### Defining a Test Protocol

To define the test protocol, you need to

- Identify all test clock signals by using the `set_dft_signal` command, as shown in the following example:

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing {45 55}
```

Make sure you identify a clock signal as a clock and not as any other signal type, even if it has more than one attribute. An error message will appear if you identify a clock signal with any other attribute.

- Identify all nonclock control signals, such as asynchronous presets and clears or scan enable signals, using the `set_dft_signal` command.

You can identify the following nonclock control signals:

- Reset
- ScanEnable
- Constant
- ScanDataIn
- ScanDataOut
- TestData
- TestMode

For example,

```
dc_shell> set_dft_signal -view existing_dft \
           -type Reset -active_state 1
```

- Define constant logic value requirements.

If a signal must be set to a fixed constant value, use the `set_dft_signal` command, as shown in the following example:

```
dc_shell> set_dft_signal -view existing_dft \
           -type constant -active_state 1
```

- Define test mode initialization requirements.

Your design might require initialization to function in test mode. Use the `read_test_protocol` command to read in a custom initialization sequence. You can define a custom initialization sequence by modifying the protocol created by the `create_test_protocol` command.

## Reading in an Initialization Protocol in STIL Format

The following example reads in an initialization protocol in STIL format (only supported format):

```
dc_shell> read_test_protocol -section test_setup my_protocol.spf
```

[Example 2-1](#) shows a complete STIL protocol file, including an initialization sequence. The initialization sequence is found in the `test_setup` section of the `MacroDefs` block.

### *Example 2-1 Complete Protocol File (init.spf)*

```
STIL 1.0 {
    Design P2000.9;
}
Header {
    Title "DFT Compiler 2000.11 STIL output";
    Date "Wed Jan  3 17:36:04 2001";
    History {
    }
}
Signals {
    "ALARM" In; "BSD_TDI" In; "BSD_TEST_CLK" In; "BSD_TMS" In;
    "BSD_TRST" In; "CLK" In; "HRS" In; "MINS" In; "RESETN" In;
    "SET_TIME" In; "TEST_MODE" In; "TEST_SE" In; "TEST_SI" In;
    "TOGGLE_SWITCH" In;
    "AM_PM_OUT" Out; "BSD_TDO" Out; "HR_DISPLAY[0]" Out;
    "HR_DISPLAY[1]" Out; "HR_DISPLAY[2]" Out; "HR_DISPLAY[3]" Out;
    "HR_DISPLAY[4]" Out; "HR_DISPLAY[5]" Out; "HR_DISPLAY[6]" Out;
    "HR_DISPLAY[7]" Out; "HR_DISPLAY[8]" Out; "HR_DISPLAY[9]" Out;
    "HR_DISPLAY[10]" Out; "HR_DISPLAY[11]" Out;
    "HR_DISPLAY[12]" Out;
    "HR_DISPLAY[13]" Out; "MIN_DISPLAY[0]" Out;
    "MIN_DISPLAY[1]" Out;
    "MIN_DISPLAY[2]" Out; "MIN_DISPLAY[3]" Out;
    "MIN_DISPLAY[4]" Out;
    "MIN_DISPLAY[5]" Out; "MIN_DISPLAY[6]" Out;
    "MIN_DISPLAY[7]" Out;
    "MIN_DISPLAY[8]" Out; "MIN_DISPLAY[9]" Out;
    "MIN_DISPLAY[10]" Out;
    "MIN_DISPLAY[11]" Out; "MIN_DISPLAY[12]" Out;
    "MIN_DISPLAY[13]" Out;
    "SPEAKER_OUT" Out;
}
SignalGroups {
    "all_inputs"   ' "ALARM" + "BSD_TDI" + "BSD_TEST_CLK" +
    "BSD_TMS" +
    "BSD_TRST" + "CLK" + "HRS" + "MINS" + "RESETN" + "SET_TIME" +
    "TEST_MODE" + "TEST_SE" + "TEST_SI" + "TOGGLE_SWITCH"; // #signals=14
    "all_outputs"   ' "AM_PM_OUT" + "BSD_TDO" + "HR_DISPLAY[0]" +
    "HR_DISPLAY[1]" + "HR_DISPLAY[2]" + "HR_DISPLAY[3]" +
    "HR_DISPLAY[4]" + "HR_DISPLAY[5]" + "HR_DISPLAY[6]" +
    "HR_DISPLAY[7]" + "HR_DISPLAY[8]" + "HR_DISPLAY[9]" +
    "HR_DISPLAY[10]" + "HR_DISPLAY[11]" + "HR_DISPLAY[12]" +
    "HR_DISPLAY[13]" + "MIN_DISPLAY[0]" + "MIN_DISPLAY[1]" +
    "MIN_DISPLAY[2]" + "MIN_DISPLAY[3]" + "MIN_DISPLAY[4]" +
    "MIN_DISPLAY[5]" + "MIN_DISPLAY[6]" + "MIN_DISPLAY[7]" +
    "MIN_DISPLAY[8]" + "MIN_DISPLAY[9]" + "MIN_DISPLAY[10]" +
    "MIN_DISPLAY[11]" + "MIN_DISPLAY[12]" + "MIN_DISPLAY[13]" +
    "SPEAKER_OUT"; // #signals=31
    "all_ports"     "all_inputs" + "all_outputs"; // #signals=45
}
```

```

        "_pi"    '"all_inputs"'; // #signals=14
        "_po"    '"all_outputs"'; // #signals=31
    }

ScanStructures {
    ScanChain "c0" {
        ScanLength 40;
        ScanIn "TEST_SI";
        ScanOut "SPEAKER_OUT";
    }
}

Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "all_inputs" { 0 { '5ns' D; } }
            "all_inputs" { 1 { '5ns' U; } }
            "all_inputs" { Z { '5ns' Z; } }
            "all_outputs" { X { '0ns' X; } }
            "all_outputs" { H { '0ns' X; '95ns' H; } }
            "all_outputs" { T { '0ns' X; '95ns' T; } }
            "all_outputs" { L { '0ns' X; '95ns' L; } }
            "CLK" { P { '0ns' D; '45ns' U; '55ns' D; } }
            "BSD_TEST_CLK" { P { '0ns' D; '45ns' U; '55ns' D; } }
            "RESETN" { P { '0ns' U; '45ns' D; '55ns' U; } }
        }
    }
}

PatternBurst "__burst__":
    "__pattern__":
}

PatternExec {
    Timing "";
    PatternBurst "__burst__";
}

Procedures {
    "load_unload" {
        W "_default_WFT_";
        V { "BSD_TEST_CLK"=0; "BSD_TRST"=0; "CLK"=0; "RESETN"=1;
            "TEST_MODE"=1; "TEST_SE"=1; "_so"=#; }
        Shift {
            W "_default_WFT_";
            V { "BSD_TEST_CLK"=P; "BSD_TRST"=0; "CLK"=P;
                "RESETN"=1;
                "TEST_MODE"=1; "TEST_SE"=1; "_so"=#; "_si"=#; }
        }
    }

    "capture" {
        W "_default_WFT_";
        F { "BSD_TRST"=0; "TEST_MODE"=1; }
        V { "_pi"=\r14 #; "_po"=\r31 #; }
    }

    "capture_CLK" {
        W "_default_WFT_";
        F { "BSD_TRST"=0; "TEST_MODE"=1; }
        "forcePI": V { "_pi"=\r14 #; }
    }
}

```

```

"measurePO": V { "_po"=\r31 #; }
"pulse": V { "CLK"=P; }
}
"capture_BSD_TEST_CLK" {
    W "_default_WFT_";
    F { "BSD_TRST"=0; "TEST_MODE"=1; }
    "forcePI": V { "_pi"=\r14 #; }
    "measurePO": V { "_po"=\r31 #; }
    "pulse": V { "BSD_TEST_CLK"=P; }
}
"capture_RESETN" {
    W "_default_WFT_";
    F { "BSD_TRST"=0; "TEST_MODE"=1; }
    "forcePI": V { "_pi"=\r14 #; }
    "measurePO": V { "_po"=\r31 #; }
    "pulse": V { "RESETN"=P; }
}
}
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        V { "BSD_TEST_CLK"=0; "CLK"=0; }
        V { "BSD_TEST_CLK"=0; "BSD_TRST"=0; "CLK"=0; "RESETN"=1;
            "TEST_MODE"=1; }
    }
}

```

**Note:**

The `read_test_protocol` -section `test_setup` command imports only the `test_setup` section of the protocol file and ignores the remaining sections.

## Setting the Scan Style

The scan style setting affects messages generated by test design rule checking. This is because some design rules apply only to specific scan styles. To set the scan style, use the following syntax for the `set_scan_configuration` command:

```
set_scan_configuration -style scan_style
```

You can use any of the following arguments for `scan_style`:

- `multiplexed_flip_flop`
- `clocked_scan`
- `lssd`
- `aux_clock_lssd`

Alternatively, you can set the `test_default_scan_style` variable instead of using the `set_scan_configuration` command.

If you do not set the scan style before performing test design rule checking, `multiplexed_flip_flop` is used as the default scan style.

---

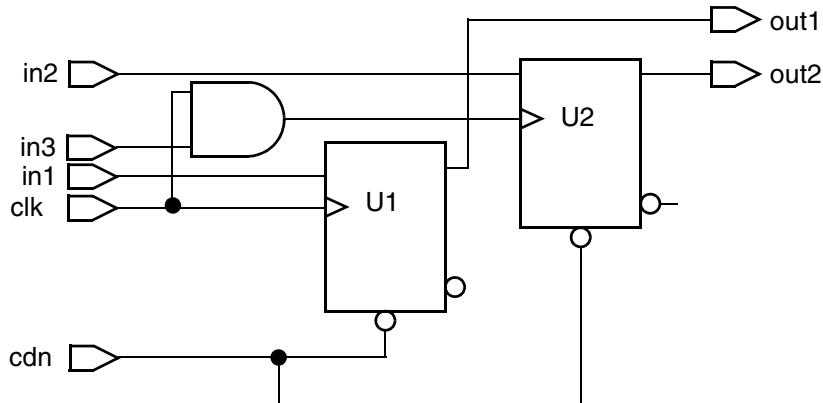
## Design Examples

This section contains two simple design examples that illustrate how to generate test protocols. The first example shows how to use the `set_dft_signal` command to control the clock signal, the scan-enable signal, and the asynchronous reset. The second example describes a two-pass process for defining an initialization sequence in a test protocol.

### Test Protocol Example 1

[Figure 2-2](#) shows a schematic and the Verilog code for a simple RTL design that needs a test protocol.

*Figure 2-2 RTL Design That Needs a Simple Protocol*



```
module tcrm (in1, in2, in3, clk, cdn, out1, out2);
  input in1, in2, in3, clk, cdn;
  output out1, out2;
  reg U1, U2;
  wire gated_clk;

  always @ (posedge clk or negedge cdn) begin
    if (!cdn) U1 <= 1'b0;
    else U1 <= in1;
  end

  assign gated_clk = clk & in3;

  always @ (posedge gated_clk or negedge cdn) begin
    if (!cdn) U2 <= 1'b0;
    else U2 <= in1;
  end

  assign out1 = U2;
  assign out2 = U1;
endmodule
```

```

if (!cdn) U2 <= 1'b0;
else U2 <= in2;
end

assign out1 = U1;
assign out2 = U2;

endmodule

```

In this design, you must define the clock signal, `clk`. You must also specify that `in3` be held at 1 during scan input to enable the clock signal for `U2`. Finally, you must hold the `cdn` signal at 1 during scan input so that the reset signal is not applied to the registers.

The following command sequence specifies a test protocol for the example design:

```

dc_shell> set_dft_signal -view existing_dft \
                     -type ScanClock -timing [list 45 55] \
                     -port clk

dc_shell> set_dft_signal -view existing_dft -port cdn \
                     -type Reset -active_state 0

dc_shell> set_dft_signal -view spec -port in3 \
                     -type ScanEnable -active_state 1

dc_shell> create_test_protocol

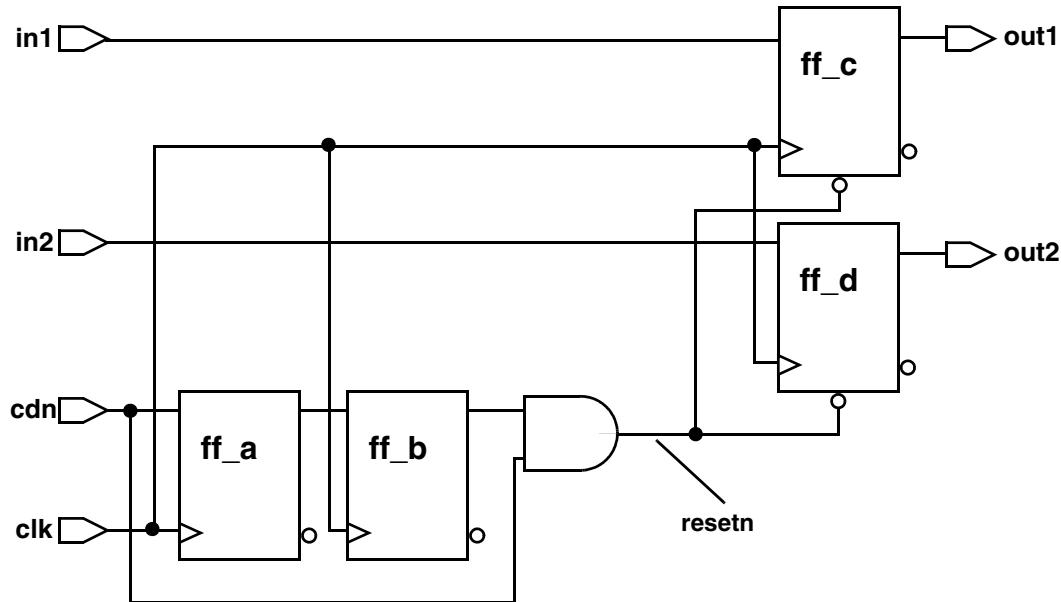
dc_shell> write_test_protocol -output design.spf

```

## Test Protocol Example 2

[Figure 2-3](#) shows a schematic and the corresponding Verilog code for an RTL design that requires initialization.

Figure 2-3 Design That Requires an Initialization Sequence



```

module ssug (in1, in2, clk, cdn, out1, out2);
input in1, in2, clk, cdn;
output out1, out2;
reg ff_a, ff_b, ff_c, ff_d;
wire resetn;

always @ (posedge clk) begin
    ff_b <= ff_a;
    ff_a <= cdn;
end
assign resetn = cdn & ff_b;
always @ (posedge clk or negedge resetn) begin
    if (!resetn) begin
        ff_c <= 1'b0;
        ff_d <= 1'b0;
    end
    else begin
        ff_c <= in1;
        ff_d <= in2;
    end
end
assign out1 = ff_c;
assign out2 = ff_d;
endmodule

```

In this design, you must define the clock signal, `clk`. You must also make sure that `cdn` and the Q output of `ff_b_reg` remain at 1 during the test cycle, so that the `resetn` signal remains at 1.

If you do not initialize the design, test DRC assumes that the `resetn` signal is not controllable and marks the `ff_c` and `ff_d` flip-flops as having design rule violations.

To initialize the design, you must hold `cdn` at 1 and pulse the `clk` signal twice so that the `resetn` signal is at 1.

For this example, the protocol is generated in a two-pass process. In the first pass, the generated protocol contains an initialization sequence based on the test attributes placed on `clk` and `cdn` ports. The command sequence that defines the preliminary protocol is as follows:

```
dc_shell> set_dft_signal -view existing_dft \
                     -type ScanClock -timing [list 45 55] \
                     -port clk

dc_shell> set_dft_signal -view existing_dft \
                     -type Constant -active_state 1 -port cdn

dc_shell> create_test_protocol

dc_shell> write_test_protocol -output first.spf
```

The resulting protocol contains the initialization steps shown in [Example 2-2](#).

#### *Example 2-2 Preliminary Initialization Sequence*

```
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        V { "clk"=0; }
        V { "cdn"=1; "clk"=0; }
    }
}
```

If you run test design rule checking without modifying these initialization steps, it reports the following violation:

```
Warning: Reset input of DFF ff_d_reg was not controlled.
(D3-1)
```

For the second pass of the protocol generation process, modify the initialization sequence as shown:

1. Add the three lines shown in bold to the `test_setup` section of the MacroDefs block:

```
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        V { "clk"=0; }
        V { "cdn"=1; "clk"=0; }
V { "cdn"=1; "clk"=P; }
V { "cdn"=1; "clk"=P; }
V { "cdn"=1; "clk"=0; }
    }
}
```

The added steps pulse the clock signal twice while holding the `cdn` port to 1. The final step holds `clk` to 0 because the test design rule checker expects all clocks to be in an inactive state at the end of the initialization sequence.

2. Save the protocol into a new file. In this case, the file is called `second.spf`.
3. Read in the new macro in one of two ways:
  - a. Reread the whole modified protocol file:  
`read_test_protocol second.spf`
  - b. Read just the initialization portion of the protocol, and use the `create_test_protocol` command to fill in the remaining sections of the protocol:  
`remove_test_protocol  
read_test_protocol -section test_setup second.spf  
create_test_protocol`
4. After you have read in the initialization protocol, perform test DRC again. The following violation is reported:

```
Warning: Cell ff_b_reg has constant 1 value. (TEST-505)
```

This is to be expected because the outputs of `ff_a` and `ff_b` did not reach 0. Constant flip-flops are not included in the scan chain.

---

## Running RTL Test DRC

After generating the test protocol, you are ready to run the test DRC process. To do this, specify the `dft_drc` command, as shown in the following example:

```
dc_shell> dft_drc
```

This command generates a set of report files containing all known design violations. You'll need to review these reports and manually fix any violations before advancing to design compilation and scan insertion.

---

## Understanding the Violations

The Test DRC process checks your design to determine if you have any test design rule violations. Before you can fix your design, you must understand what types of violations are checked and why these checks are necessary.

This section explains the test design rule checks that are performed on your design, describes messages you see when you encounter test design rule violations, and describes the methods you can use to fix the violations.

The section has the following subsections:

- [Violations That Prevent Scan Insertion](#)
  - [Violations That Prevent Data Capture](#)
  - [Violations That Reduce Fault Coverage](#)
- 

### **Violations That Prevent Scan Insertion**

Scan design rules require that in test mode the registers have the functionality to operate as cells within a large shift register. This enables data to get into and out of the chip. The following violations prevent a register from being scannable:

- The flip-flop clock signal is uncontrollable.
- The latch is enabled at the beginning of the clock cycle.
- The asynchronous controls of registers are uncontrollable or are held active.

### **Uncontrollable Clocks**

This violation can be caused by undefined or unconditioned clocks. DFT Compiler considers a clock to be controlled only if both of these conditions are true:

- The clock is forced to a known state at time = 0 in the clock period, which is the same as the “clock off state” in TetraMAX.
- The clock changes state as a result of the test clock toggling.

Going to an unknown state (X) is considered to be a change of state. However, if the clock stays in a single known state no matter what state the test clock is in, the clock will generate a violation for not being reached by any test clock.

You must use the `set_dft_signal` command to define test clocks in your design. For more information, see [“Defining a Test Protocol” on page 2-3](#).

Also use the `set_dft_signal` command to condition gated clocks to reach their destinations, as shown in the following example:

```
dc_shell> set_dft_signal -view existing_dft  
          -type constant -active_state 1
```

The violation message provides the name of the signal that drives the clock inputs and the registers that ATPG cannot control.

If a design has an uncontrollable register clock pin, it generates one of the following messages:

```
D1      Clock input clocked_on of DFF was not controlled.  
D4      Clock input clocked_on of DLAT was not controlled.
```

## Latches Enabled at Beginning of Clock Cycle

This violation applies only when the scan style is set to level-sensitive scan design (LSSD). For a latch to be scannable, the latch must be forced to hold its value at the beginning of the cycle, when the clock is inactive. This violation can be caused by undefined or unconditioned clocks.

This violation indicates that there are registers that cannot be controlled by ATPG. If the violation is not corrected, these registers will be unscannable and fault coverage will be reduced.

## Asynchronous Control Pins in Active State

Asynchronous pins of a register must be capable of being disabled by an input of the design. If they cannot be disabled, this is reported as a violation. This violation can be caused by asynchronous control signals, such as the preset or clear pin of the flip-flop or latch, that are not properly conditioned before you run DFT Compiler. You might be able to fix this by setting a signal as `active_state` that has a hold value of 0 during scan shift or by defining a signal as `active_state` that has a hold value of 1. If you create all signal definitions correctly before running DFT Compiler, this violation indicates registers that ATPG cannot control.

If a register has an asynchronous pin that is not controlled by an asynchronous control signal, you get one of the following messages:

```
D2      Set input of DFF was not controlled.  
D3      Reset input of DFF was not controlled.  
D5      Set input of DLAT was not controlled.
```

## Violations That Prevent Data Capture

After DFT Compiler checks for violations that prevent scan insertion, the next step is to verify that your design can get valid data during the capture phase of ATPG.

Note that ATPG does not consider timing when generating vectors for a scan design. If you do not fix the violations in this section, ATPG might generate vectors that fail functional simulation or fail on the tester, although in TetraMAX you would also have to override the TetraMAX default settings.

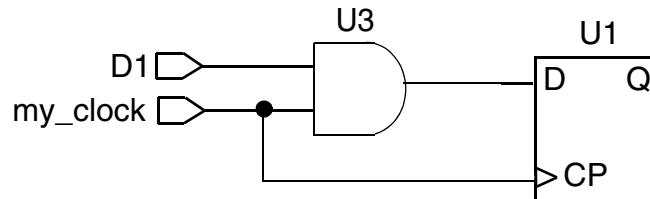
The violations are described in the following sections:

- [Clock Used As Data](#)
- [Black Box Feeds Into Clock or Asynchronous Control](#)
- [Source Register Launch Before Destination Register Capture](#)
- [Registered Clock-Gating Circuitry](#)
- [Three-State Contention](#)
- [Clock Feeding Multiple Register Inputs](#)

### Clock Used As Data

When a clock signal drives the data pin of a cell, as in [Figure 2-4](#), ATPG tools cannot determine the captured value. Modify the logic leading to the datapaths to eliminate dependency on the clock.

*Figure 2-4 Clock Signal Used As Data Input*



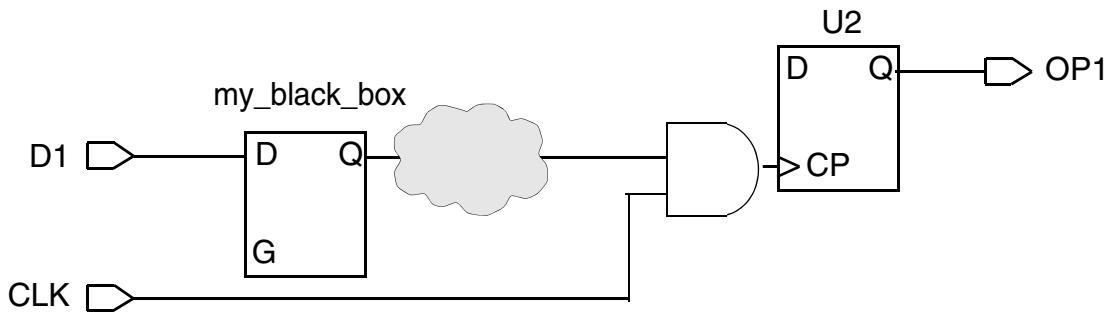
If the clock and data input to a register are interdependent, you might get the following message:

D11              Clock my\_clock connects to clock and data inputs of DFF

## Black Box Feeds Into Clock or Asynchronous Control

If the output of a black box indirectly feeds into the clock of a register, the register might not be able to capture data. An example is shown in [Figure 2-5](#).

*Figure 2-5 Black Box Feeds Clock Input*



For more information about black boxes, see [“Black Boxes” on page 2-22](#).

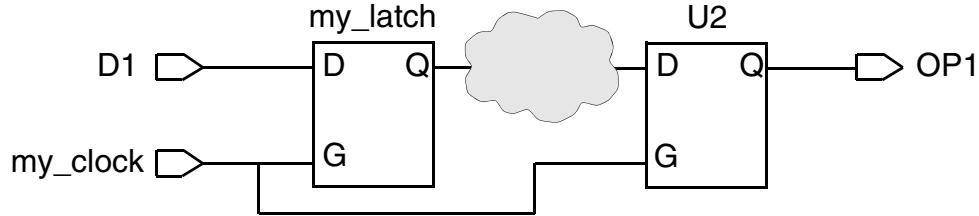
## Source Register Launch Before Destination Register Capture

This section describes the violations caused by source registers that launch new data to the destination registers before they can capture and shift out the original data.

When two latches are enabled by the same clock but have a combinational datapath between them, data can propagate through both latches in a single clock cycle. This reduces the ability of ATPG to observe logic along this path. Modify the logic leading to the affected latches to eliminate any paths affected by latches that are enabled by the same clock.

An example of this violation is shown in [Figure 2-6](#). When the clock turns off, that is, pulses from an inactive state to an active state and then back, the second latch (U2) can capture the value originally on port D1 or on its data pin, depending on the relationship between the clock width and the delay on the datapath. The possibility of data feedthrough causes the destination latch (U2) to capture data unreliable.

*Figure 2-6 Latch-Based Circuit With Source Register Launch Before Destination Register Capture*



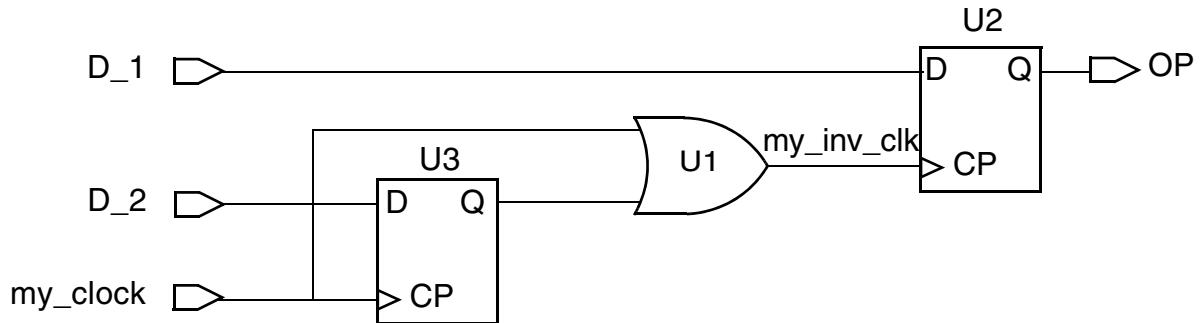
If multiple latches are enabled so that the latches feed through capture data, you get the following message:

D8       clock my\_clock cannot capture data with other clocks off.

## Registered Clock-Gating Circuitry

If you gate the register output with the same clock signal that is used to clock the register, you cannot use the same phase of the resulting signal as a clock. An example is shown in [Figure 2-7](#).

*Figure 2-7 Invalid Clock-Gating Circuit*



The U1 output invalidly clocks register U2. The OR gate, U1, has two inputs, where one is the output of register U3 and the other is the signal used to clock U3.

Note that Power Compiler clock gating does not lead to this violation because Power Compiler uses opposite edge-triggered flip-flops or latches to create the clock-gating signals.

This circuit configuration results in timing hazards, including clock glitches and clock skew. Modify the clock-gating logic to eliminate this type of logic.

If you implement this type of clock-gating circuitry, you get the following message:

D1                  Clock input CP of DFF was not controlled.

## Three-State Contention

DFT Compiler can check to see if your RTL code contains three-state contention conditions. If floating or contention is found, one of the following three error messages is issued:

D20                  Bus gate N failed contention ability check for drivers G1 and G2 .

D21                  Bus gate N failed Z state ability check.

D22                  Wire gate N failed contention ability check for drivers G1 and G2 .

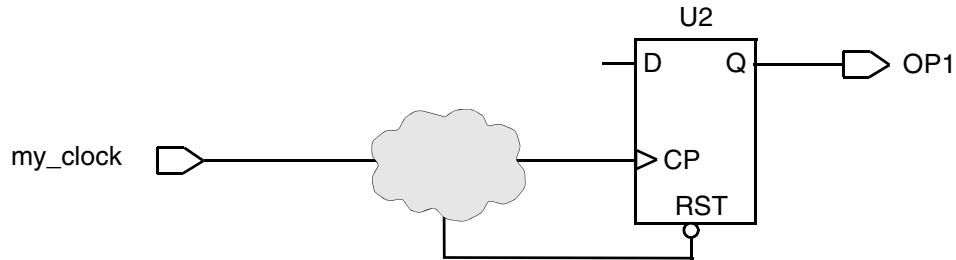
## Clock Feeding Multiple Register Inputs

A clock that feeds multiple register inputs reduces the fault coverage attainable by ATPG. The signal can be one of the following:

- A clock signal that feeds into more than one register clock pin
- A clock signal that feeds into a clock pin and an asynchronous control of a register

The logic that feeds the same clock into multiple clock pins or asynchronous pins should be modified so that the clock reaches only one port on the register. [Figure 2-8](#) shows an example of this violation.

Figure 2-8 Clock Signal Feeds Register Clock Pin and Asynchronous Reset



If you implement this type of design circuitry, you get the following message:

D12                  Clock CLK connects to clock/set/reset inputs (G1/G2) of DFF

---

## Violations That Reduce Fault Coverage

Violations that can reduce your fault coverage are discussed in the following sections:

- [Combinational Feedback Loops](#)
- [Clocks That Interact With Register Input](#)
- [Multiple Clocks That Feed Into Latches and Flip-Flops](#)
- [Black Boxes](#)

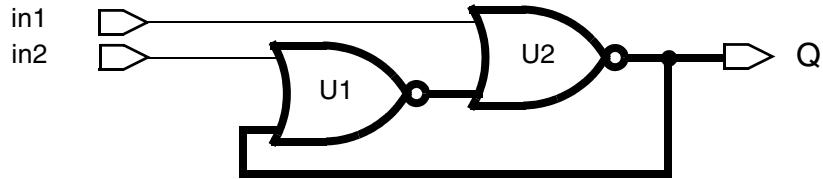
### Combinational Feedback Loops

An active (or *sensitizable*) feedback loop reduces the fault coverage that ATPG can achieve by increasing the difficulty of controlling values on paths containing parts of the loop.

A loop that oscillates causes severe problems for ATPG and for fault simulation. You can break these loops by placing test constraints on the design. This creates a feedback loop that is not active. DFT Compiler does not report violations on loops that you have broken by setting constraints.

If you are using the loop as a latch, convert the combinational elements that make up this feedback loop into a latch from your ASIC vendor library. [Figure 2-9](#) shows this type of loop.

*Figure 2-9 Highlighted Combinational Feedback Loop*



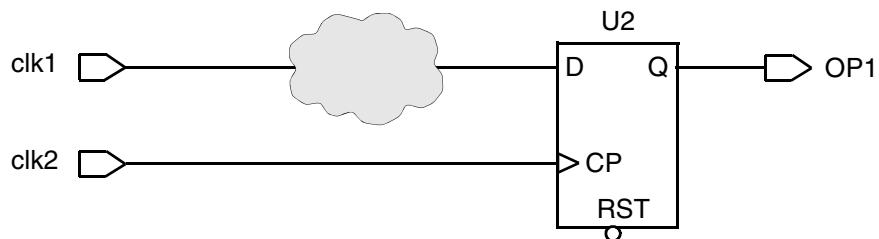
If your design contains a sensitizable feedback loop, you get the following message:

D23              Feedback path network X sensitizable through source gate U1.

## Clocks That Interact With Register Input

A clock that affects the data input of a register reduces the fault coverage attainable by ATPG, because ATPG pulses only one clock at a time, keeping all other clocks in their off states. Attempting to fix this purely in the ATPG setup can result in timing hazards. Do not use the circuit shown in [Figure 2-10](#), because testing this logic requires multiple ATPG iterations and might also require special scan chain design considerations (not discussed here). Redesign the logic feeding the data inputs of the registers to eliminate dependency on other clocks.

*Figure 2-10 Clock Interacting With Register Input*



If a clock affects the data of a register, you'll get the following message:

D10              Clock clk1 connects to data input (D) of DFF U2.

## Multiple Clocks That Feed Into Latches and Flip-Flops

This section describes the types of clock-gating configurations that can reduce fault coverage. Other clock-gating configurations that prevent scan insertion and data capture are described in “Violations That Prevent Scan Insertion” on page 2-13 and “Violations That Prevent Data Capture” on page 2-15.

### Latch Requires Multiple Clocks to Capture Data

For a latch to be usable as part of a scan chain, it must be enabled by one clock or by a clock ANDed with data derived from sources other than that clock. Multiple clocks and gated clocks must be ORed together so that any one of the clocks can capture data. ATPG forces all but one clock off at any time. Latches that can capture data as a result of more than one clock must be able to capture data with one clock active and all others off.

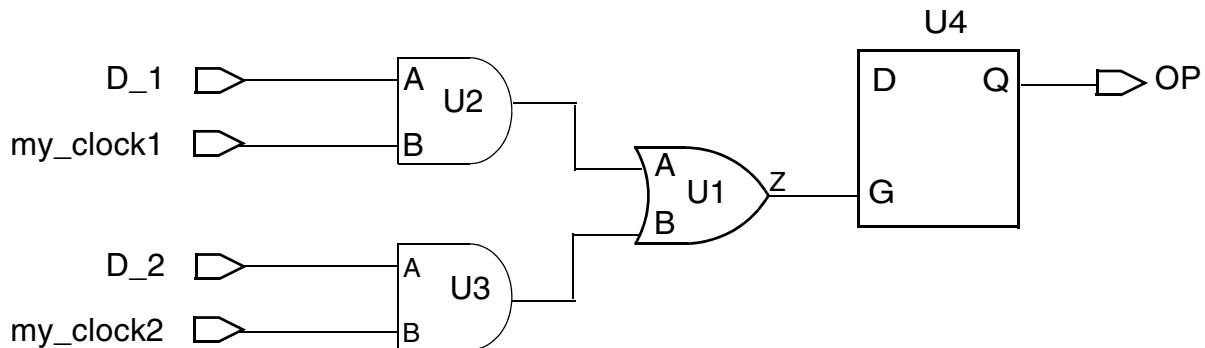
If your design has an OR gate with clock and data inputs, the output clock of the OR gate has extra pulses that depend on the data input. If your design has an AND gate with more than one clock input, the output of the AND gate never generates a clock pulse. Both of these cases are violations, and DFT Compiler generates a warning message.

You can create valid clock-gating logic for latches if your circuitry contains an

- AND gate with only one clock input and one or more data inputs
- OR gate with clock or gated clock inputs

A combination of these valid clocking rules is shown in [Figure 2-11](#).

*Figure 2-11 Valid Latch Clock Gating*



If you generate logic that violates these clock rules, you get the following message:

D8

Clock clk1 cannot capture data with other clocks off. (D8-1)

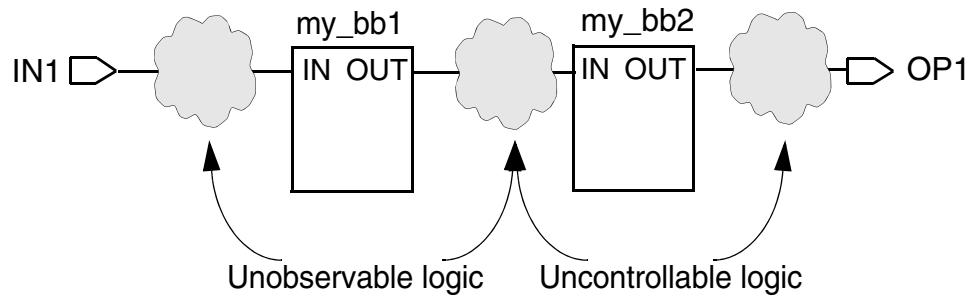
## Latches Are Not Transparent

Latches should be transparent in certain types of scan styles. If a latch is not transparent, ATPG might have more difficulty controlling it. This could cause a loss of fault coverage on the path through the latch.

## Black Boxes

Logic that drives or is driven by black boxes cannot be tested because it is unobservable or uncontrollable. This violation can drastically reduce fault coverage, because the logic that surrounds the black box is unobservable or uncontrollable. [Figure 2-12](#) shows an example.

*Figure 2-12 Effect of Black Boxes on Surrounding Logic*



If there are any black boxes in your design, `dft_drc` issues the following warning:

```
Warning: Cell U0 (black_box) is unknown (black box) because
functionality for output pin Z is bad or incomplete. (TEST-
451)
```

# 3

## Running Test DRC Debugger

---

This chapter describes debugging design rule checking (DRC) violations by using the DFT graphical user interface (GUI).

The DFT GUI is run in the Design Vision environment, and it provides analysis tools for viewing and analyzing your design. It allows you to view the design violations, and it can provide an early warning of test-related issues. The DFT GUI provides the debug environment for pre-DRC violations, post-DRC violations, and Core Test Language (CTL) models.

This chapter includes the following sections:

- [Starting and Exiting the Graphical User Interface](#)
- [Exploring the Graphical User Interface](#)
- [Viewing Design Violations](#)
- [Commands Specific to the DFT GUI](#)

---

## Starting and Exiting the Graphical User Interface

To invoke the DFT GUI from Design Vision, you need to

- Execute `design_vision -xg_mode -tcl` from the command line.
- Choose File > Execute Script to run `dc_shell` script.
- Choose Test > Run DFT DRC to check the design for DRC violations. This brings up the violation browser.

Alternatively,

- Enter `dft_drc` command on the command line. Then choose Test > Browse Violations to invoke the violation browser.

To exit Design Vision,

- Choose File > Exit

You can also enter `exit` or `quit` on the command line or press Control-c three times in the UNIX or Linux shell.

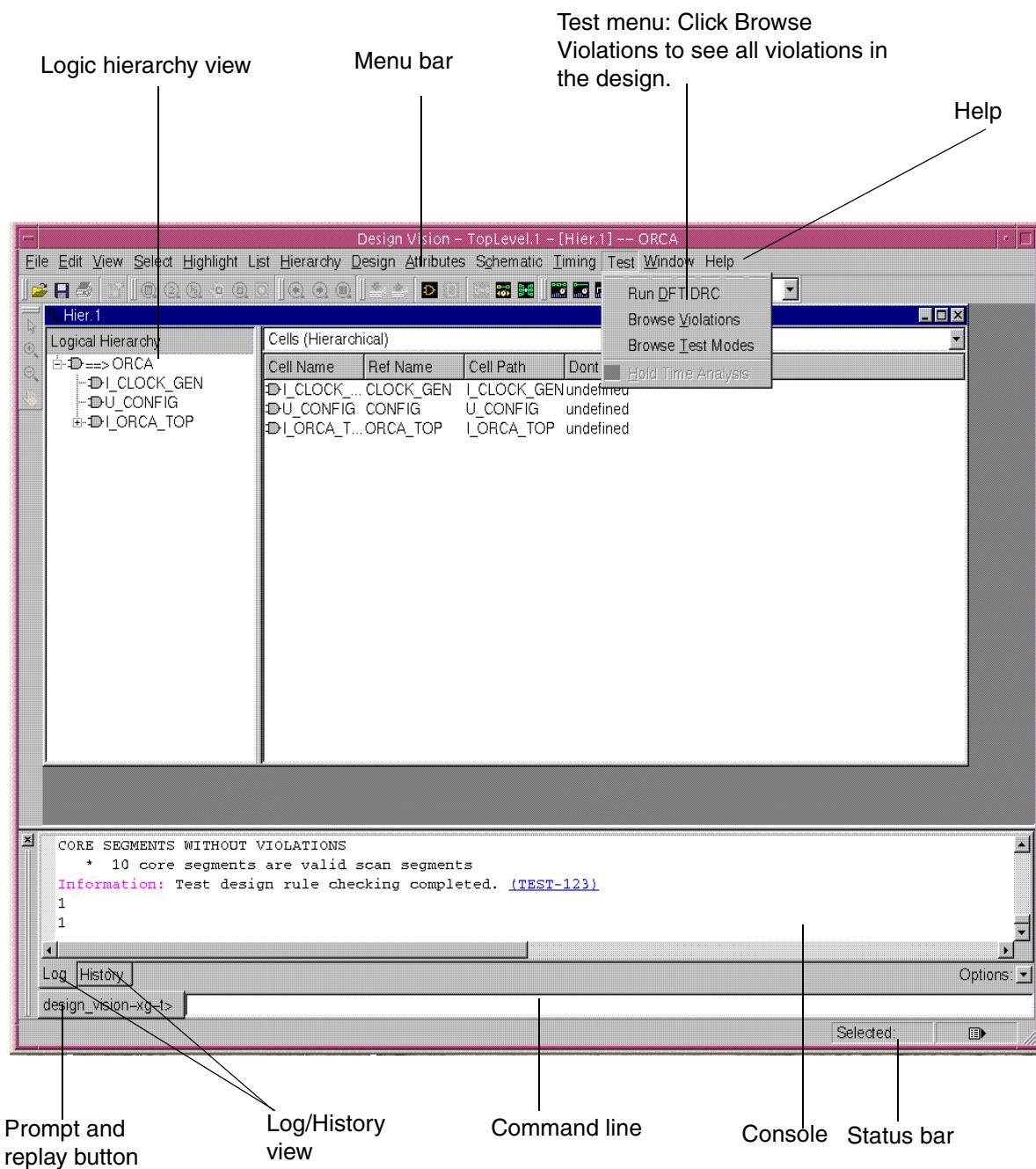
---

## Exploring the Graphical User Interface

The Design Vision window is a top-level window in which you can display design information in various types of analysis views and windows. The DFT GUI is invoked in this environment. It functions as a visual analysis tool to help you to visualize and analyze the violations in your design.

[Figure 3-1](#) shows the DFT GUI window running in the Design Vision foreground.

Figure 3-1 The DFT GUI Window



The window consists of a title bar, a menu bar, and several toolbars at the top of the window and a status bar at the bottom of the window.

You use the workspace between the toolbars and the status bar to display view windows containing graphical and textual design information. You can open multiple windows and use them to compare views, or different design information within a view, side by side.

---

## Logic Hierarchy View

The logic hierarchy view helps you navigate through your design and gather information. The view is divided into the following two panes:

- Instance tree
- Objects list

The instance tree lets you quickly navigate the design hierarchy and see the relationships among its levels. If you select the instance name of a hierarchical cell (one that contains subblocks), information about that instance appears in the object table. You can Shift-click or Control-click instance names to select combinations of instances.

By default, the object table displays information about hierarchical cells belonging to the selected instance in the instance tree. To display information about other types of objects, select the object types in the list above the table. You can display information about hierarchical cells, all cells, pins and ports, pins of child cells, and nets.

---

## Console Window

The console provides a command-line interface and displays information about the commands you use in the session in the following two views:

- Log view
- History view

The log view is displayed by default when you start Design Vision. The log view provides the session transcript. The history view provides a list of the commands that you have used during the session. To select a view, click the tab at the bottom of the console.

---

## **Command Line**

You can enter dc\_shell commands on the command line at the bottom of the console. Enter these commands just as you would enter them at the dc\_shell prompt in a standard UNIX or Linux shell. When you issue a command by pressing Return or clicking the prompt button to the left of the command line, the command output, including processing messages and any warnings or error messages, is displayed in the console log view.

You can display, edit, and reissue commands on the console command line by using the arrow keys to scroll up or down the command stack and to move the insertion point to the left or right on the command line. You can copy text in the log view and paste it on the command line.

You can also select commands in the history view and edit or reissue them on the command line.

---

## **Viewing Man Pages**

The DFT GUI provides an HTML-based browser window that lets you view, search, and print man pages for commands, variables, and error messages.

To view a man page in the man page viewer,

1. Choose Help > Man Pages.
  2. Click the category link for the type of man page you want to view: Commands, Variables, or Messages.
  3. Click the title link for the man page you want to view.
- 

## **Menus**

The menu bar provides menus with the commands you need in order to use the DFT GUI. Use the Test menu to view design violations and to invoke the violation browser.

## **Checking Scan Test Design Rules**

Check the current design for DRC violations in your scan test implementation before you perform other DFT Compiler operations such as inserting scan cells. You can use the violation browser and the violation inspector to examine and debug any DRC violations that you find.

To view DRC violations,

- Choose Test > Run DFT DRC.

DFT Compiler checks the design for DRC violations and displays messages in the console log view. If violations exist, Design Vision automatically opens a new Design Vision window and displays the violation messages in the violation browser window.

## Examining DRC Violations

You can use the DRC violation browser to search for and view information about DFT unified DRC violations in the current design. The violation browser can display both static and dynamic violation messages. Static violations occur as a result of the design topology. To detect dynamic violations, you must simulate the design in the violation inspector.

To invoke the violation browser and view violations,

- Choose Test > Browse Violations

The violation browser window appears in a new Design Vision window docked to the left side of the window.

See “[Viewing Design Violations](#)” on page 3-6.

## Viewing Test Protocols

You can view details about the default test protocol and any user-defined test protocols that you created for the design.

To view test protocols,

- Choose Test > Browse Test Modes. The Test Modes Details dialog box appears. Alternatively, you can open this dialog box by clicking the Test Modes button in the violation inspector window.
- Select a test protocol name in the Test Modes list.

---

## Viewing Design Violations

This section covers the following topics:

- [Examining DRC Violations](#)
- [Inspecting DRC Violations](#)

## Examining DRC Violations

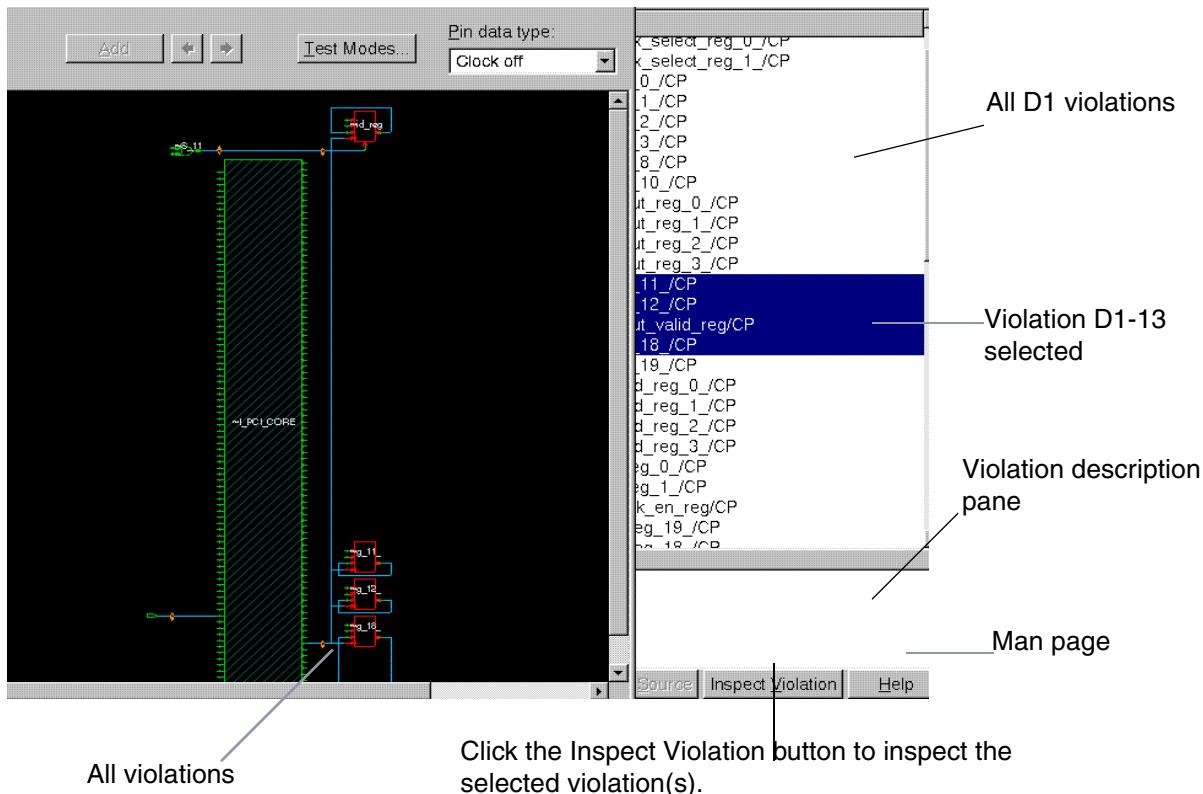
The violation browser lets you examine detailed information about violations and also provides a variety of tools for viewing the different aspects of a violation. The violation browser groups the warning and error messages into categories that help you find the problems you are concerned about.

To view a violation,

- In the main window, choose Test > Browse Violations.

This launches the violation browser window, as shown in [Figure 3-2](#).

*Figure 3-2 Violation Browser Window*



The violation browser window consists of two panes: a violation category tree on the left and a violation pin list on the right. The Violation Categories pane lists different categories of violations, for example, Modeling and Pre-DFT.

- Click the expansion button (plus sign) of the violation category to display the violations of that group.

The expanded view displays the types and number of violations.

When you select a violation in the left pane, a list of pins where the violations occur appears in the right pane.

For example,

- Expand the Pre-DFT category view.
- Select violation D1.

The resulting D1 violations are shown in the right-side pane.

- Click a specific violation pin or violation ID, and the corresponding description is displayed in the description pane.
- Click the Inspect Violation button to view the violation schematic. For more details, see “[Inspecting Static DRC Violations](#).”

(Optional) To view the man page of a violation, click the Help button.

---

## Inspecting DRC Violations

You can analyze and debug DFT unified DRC violations by inspecting them in a violation inspector window. You can inspect one or more violations of the same type. The violation inspector provides both a schematic view for inspecting static violations and a coordinated waveform view for simulating dynamic violations.

This section has the following subsections:

- [Inspecting Static DRC Violations](#)
- [Inspecting Dynamic DRC Violations](#)

---

## Inspecting Static DRC Violations

The violation inspector integrates the schematic viewer with the waveform viewer to debug violations. However, the waveform view within the violation inspector is useful only for debugging dynamic violations and only for pin data that consists of data values over a sequence of events or a period of time. For pin data types that are constant or do not fit the concept of values over time or sequence of events, pin data values are represented as strings on schematics, and the waveform view in the violation inspector is hidden in such cases.

This section has the following subsections:

- [Viewing a Violation](#)
- [Viewing Multiple Violations](#)
- [Viewing CTL Models](#)

## **Viewing a Violation**

When you select a violation in the violation browser, the corresponding path schematic is displayed in the violation inspector so that you can investigate the violations. A path schematic can contain cells, pins, ports, and nets.

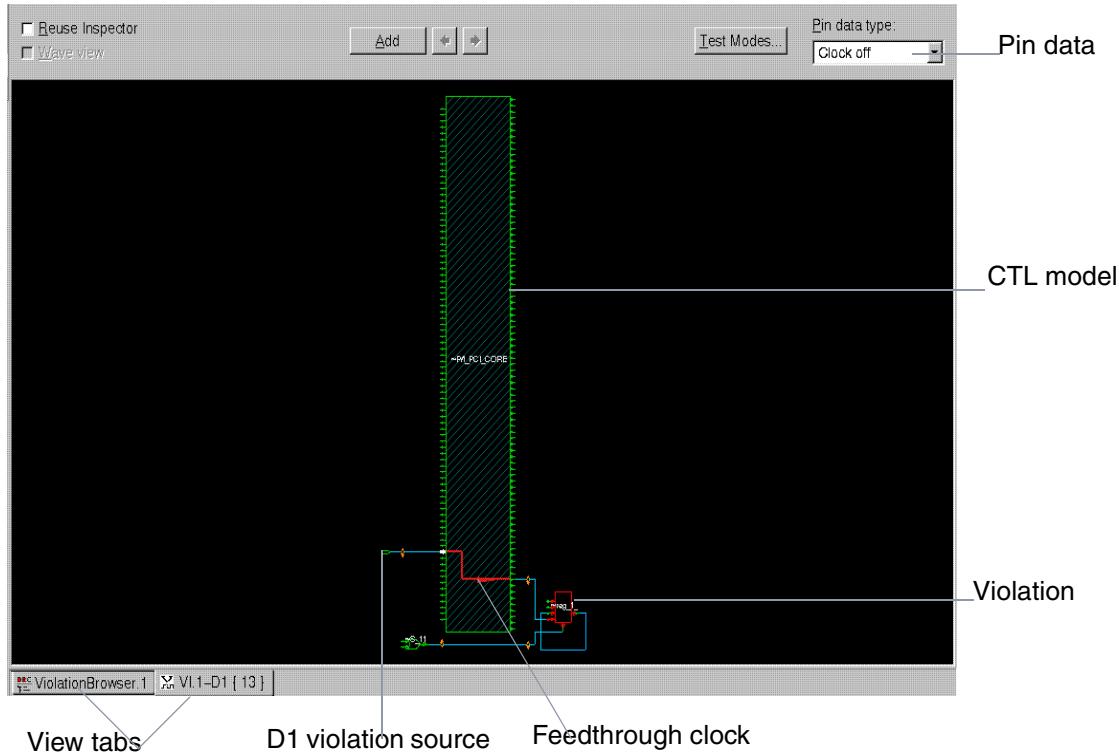
To open the violation schematic,

- Click the Inspect Violation tab at the bottom of the console.

This opens the violation inspector window, as shown in [Figure 3-3](#).

This window displays the path schematic for visually examining any violations and the violation source. A path schematic provides contextual information and details about the path and its components. Red-colored cells indicate pins with violations.

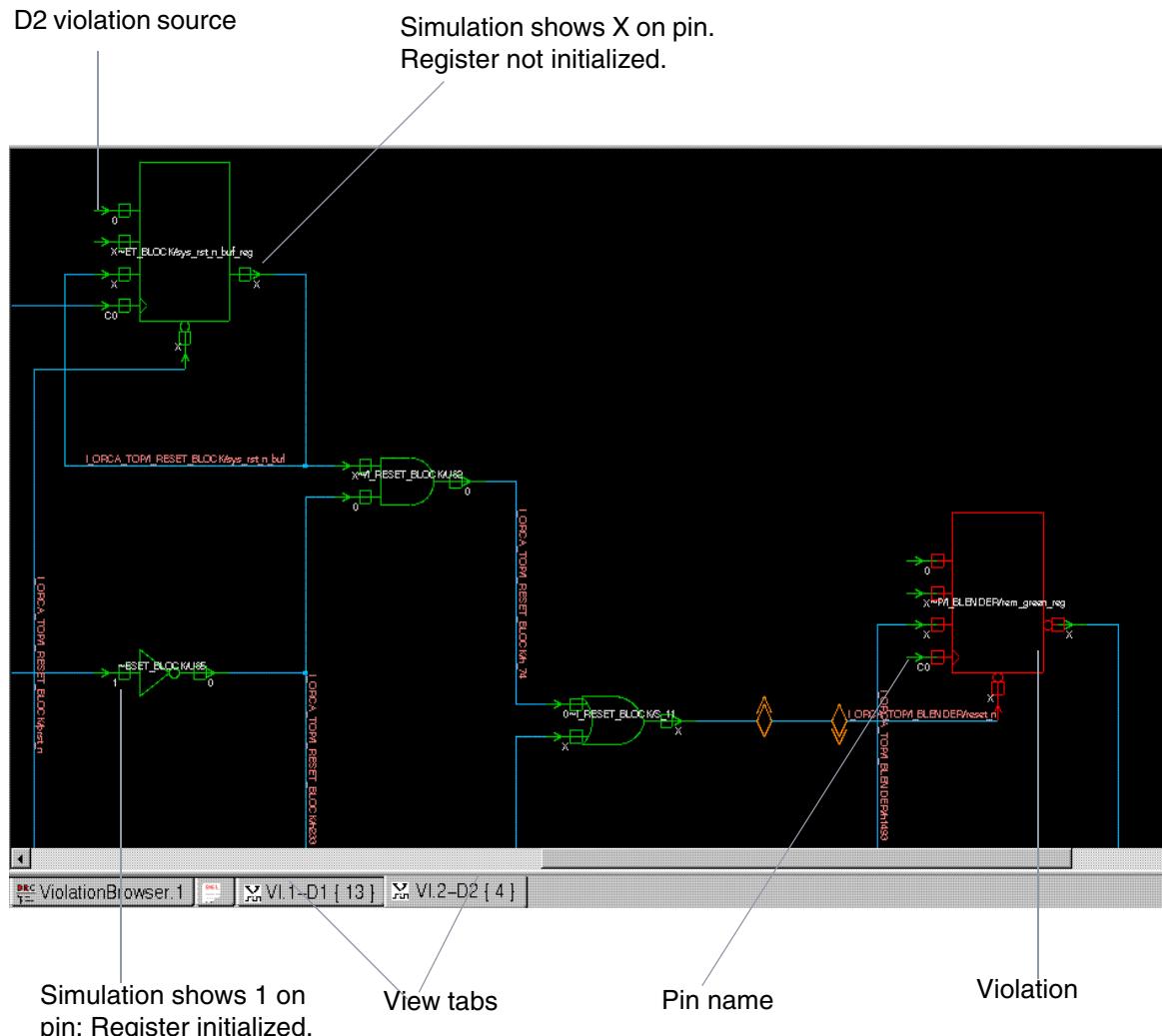
Figure 3-3 Schematic Viewer: Viewing a Violation



- (Optional) Select the data type name in the “Pin data type” menu to display a different pin data type.
- Display object information in an InfoTip by moving the pointer over a pin, cell, net, or other type of object.
- Pin information includes the cell name, pin direction, and simulation values.
- Cell information includes the cell name and the names and directions of the attached pins.
- Net information includes the net name, local fanout value, and fanout value.

Observe the simulation values displayed at the input and output pins in [Figure 3-4](#).

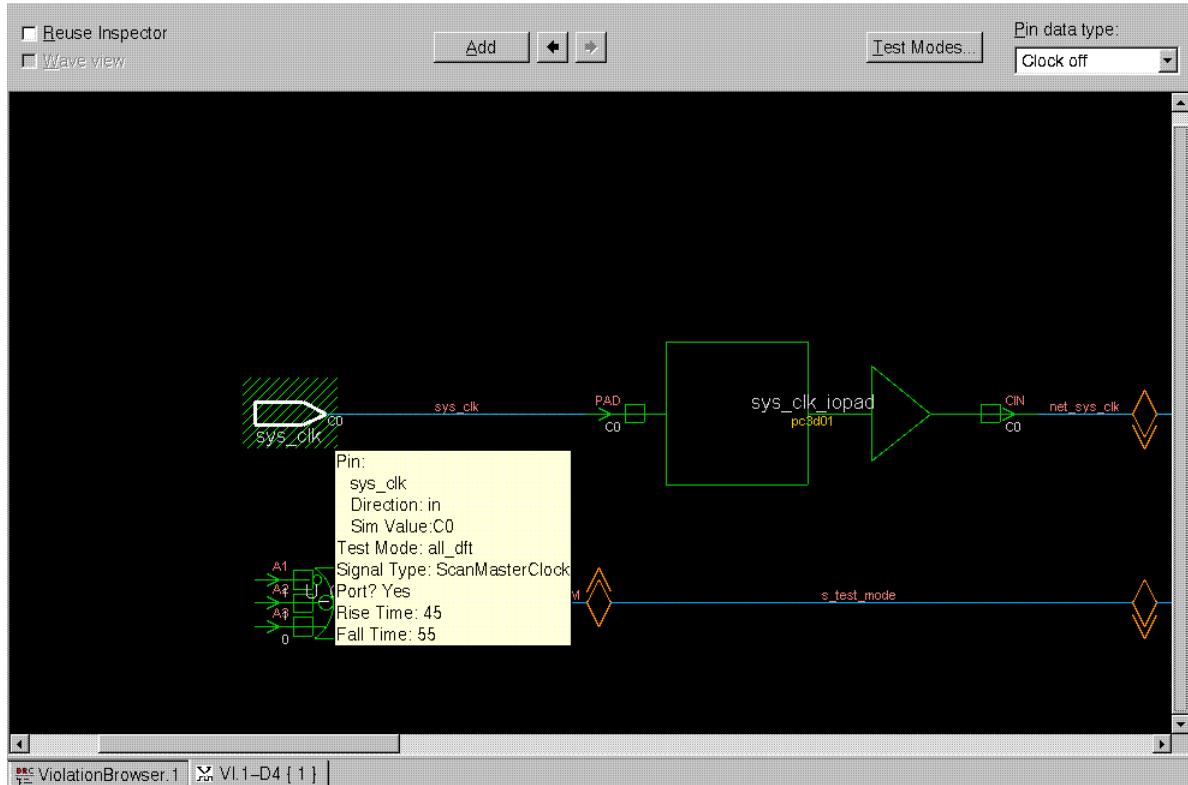
Figure 3-4 Viewing Simulation Values



Note that you can browse back and forth between the path schematics of D1 and D2 violations by clicking the corresponding D1 and D2 tabs at the bottom of the console window.

- If you define a test pin with the `set_dft_signal`, the pin appears with a hatched fill pattern and the InfoTip displays the pin information. See how the test pins are specified in [Figure 3-5](#).

Figure 3-5 Test Pin Defined



## Viewing Multiple Violations

To view the path schematics for multiple violations,

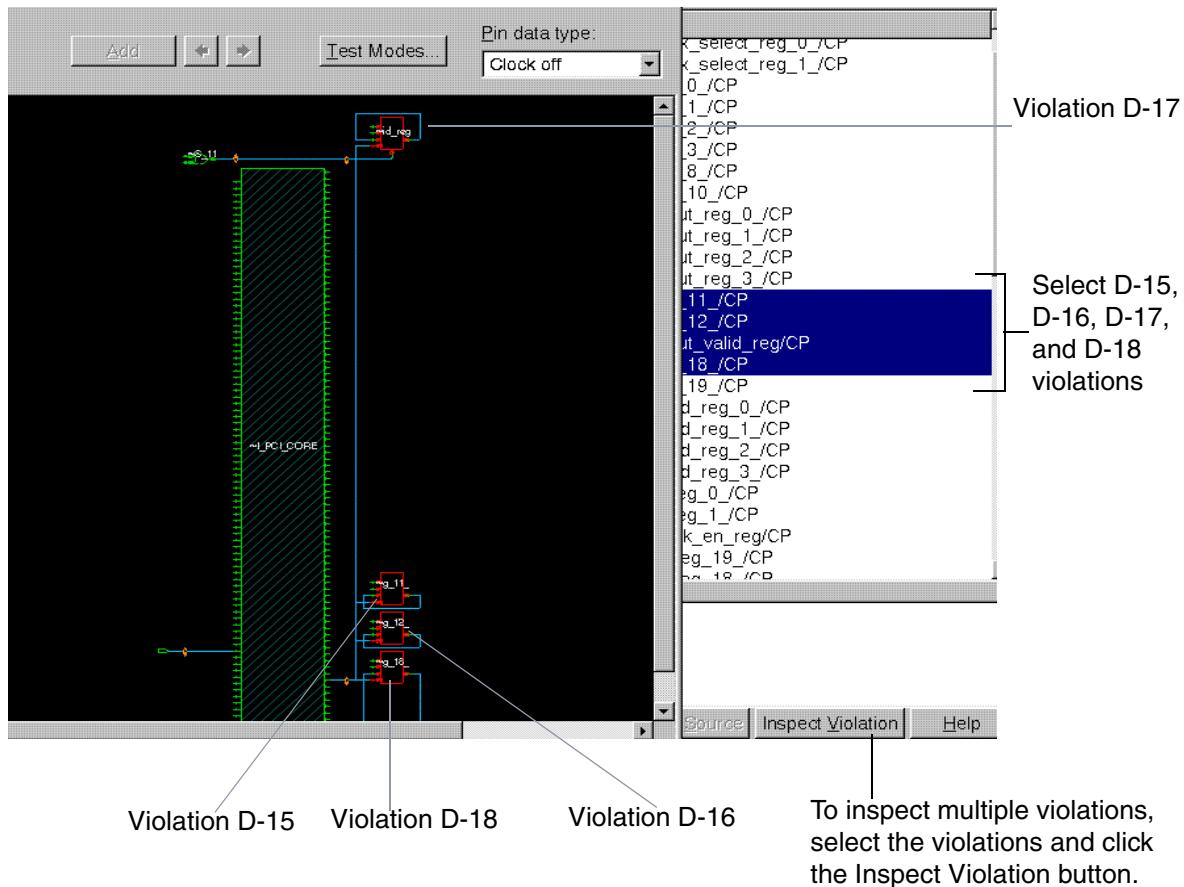
1. Shift-click to select multiple violation IDs in the violation browser.
2. Click the Inspect Violation button at the bottom of the window.

The schematics of the selected violations are displayed in the schematic viewer of the violation inspector.

3. (Optional) Click the Help button to bring up the Man Page Viewer.

Figure 3-6 shows the selection of multiple violations.

**Figure 3-6 Viewing Multiple Violations**



## Viewing CTL Models

A CTL model provides information about scan cells and the test modes in which they are active. It also describes the number of scan cells in a chain and the pins associated with the particular scan chain.

If your design contains CTL models, the violation schematic displays them as black boxes with a hatched fill pattern to distinguish them from other cells.

A CTL model allows you to view the feedthrough signals, like clocks and asynchronous signals. You can view the fanin and fanout for the path schematics. This step can provide useful information about the logic that drives, or is driven by, the problem path.

The following pins are displayed in a CTL model:

- Scan input

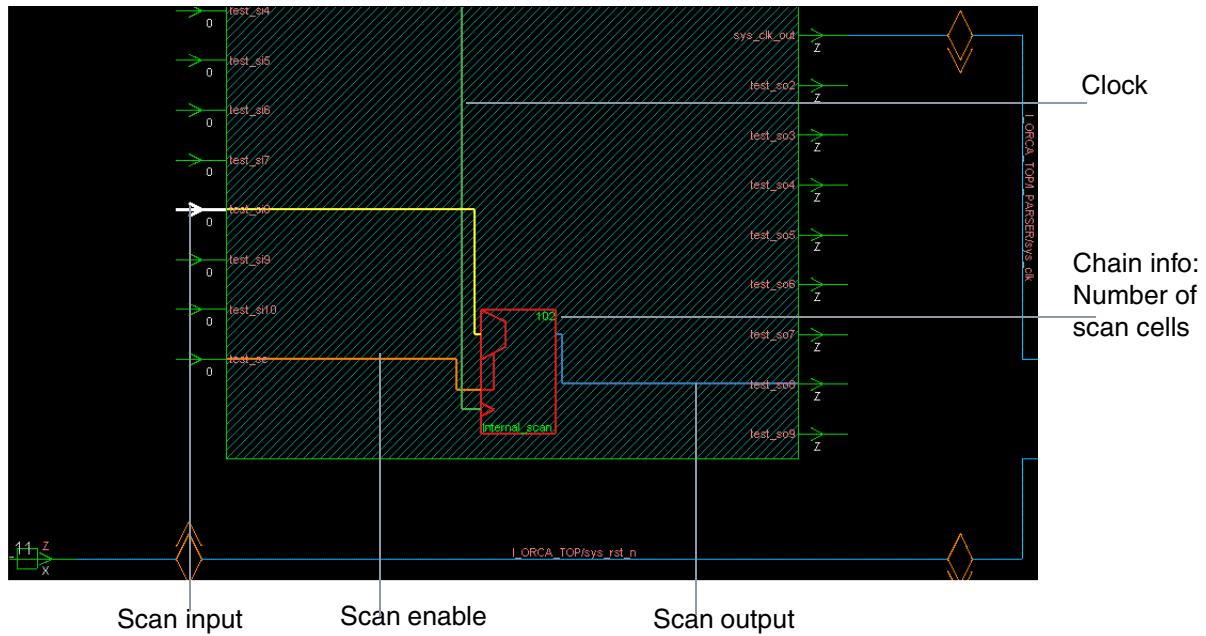
- Scan output
- Scan enable
- Scan clock

To display the feedthrough flylines for a CTL model,

1. Select an input or output pin on the model.
2. Right-click and choose “Show feed throughs”.

See the design schematics of the CTL model in [Figure 3-7](#).

*Figure 3-7 CTL Scan Chain Information*



## Inspecting Dynamic DRC Violations

The waveform viewer displays a coordinated waveform view for simulating dynamic violations. The violation inspector provides both a schematic view (violation schematic) for inspecting static violations and a coordinated waveform view for simulating dynamic violations.

The pin data for dynamic violations represents simulation values for a series of initialization cycles. To debug dynamic violations, you can select pins in the violation schematic and view their simulation values in the waveform view. Simulation values can be constant, or they can vary over time in a series of simulation “events.”

The violation inspector displays the pin data that corresponds to the most suitable pin data type for debugging the violation. To simulate the pin data for a dynamic violation, you must change to a pin data type that supports simulation values.

To display waveforms for pins with dynamic violations,

- Select one or more pin names in the violation browser.
- Click the Inspect Violation button.
- To display the waveform view, select “Test setup” in the “Pin data type” list.

The waveform view appears below the schematic view in the violation inspector window, as shown in [Figure 3-8](#). You can adjust the relative heights of these views by dragging the split bar up or down.

The waveform view consists of two panes: an expandable signal list on the left and the waveform viewer on the right. You can adjust the relative widths of the panes by dragging the split bars left or right.

- Select one or more objects (pins, cells, nets, or buses) for the signals that you want to inspect.
- Click the “Add to Wave View” button.

The signal names and values appear in the signal list, and a waveform for each signal appears in the waveform viewer.

Figure 3-8 Waveform Viewer



To change the visible time range,

- Drag the pointer left or right over the portion of the global time range that you want to view.

You can use the reference and target markers, C1 and C2, to measure the time between events. C1 marks the current event and C2 marks the event you want to measure. The number of events or time units between the markers appears in the marker region above the upper time scale.

- To move C1, click or drag the pointer in the marker region.

- To move C2, middle-click or drag the pointer with the middle mouse button in the marker region

You can move or copy signals into a group or from one group to another. You can also remove selected signals or clear the waveform view.

To move signals into a group or from one group to another,

1. Select the signal names in the signal list pane.
2. Drag the selected signals over the group name.

To copy signals into a group or from one group to another,

1. Select the signal names in the signal list pane.
2. Shift-drag the selected signals over the group name.

To remove signals from the waveform view,

1. Select the signal names in the signal list pane.
2. Click the Selected button.

To clear the waveform view, click the All button.

---

## Commands Specific to the DFT GUI

Detailed descriptions of commands and options specific to the DFT GUI are listed in this section.

---

### **gui\_inspect\_violations**

The `gui_inspect_violations` command brings up the specified DFT DRC violations in a new violation inspector window unless a violation inspector window has been marked for reuse. If no violation inspector window exists, a new violation inspector window is created as a new top-level window. Subsequent windows are created in the active top-level window. The new violation inspector window that is created is not marked reusable.

The syntax for this command is

```
gui_inspect_violations -type violation_type violation_list
```

---

Options	Description
<code>-type violation_type</code>	Specifies the type of violation, such as D1, to be inspected. This optional argument affects how the <code>violation_list</code> option is interpreted.
<code>-type violation_list</code>	Specifies the list of one or more violations to inspect. If the <code>-type</code> option is not used, it specifies that only a single violation instance needs to be inspected by the violation inspector.

---

To inspect multiple violations (5, 9,13) of type D1, for example, use the following syntax:

```
gui_inspect_violations -type D1 {5 9 13}
```

To inspect a single violation 4 of type D2, for example, use the following syntax:

```
gui_inspect_violations -type D2 4
```

or

```
gui_inspect_violations D2-4
```

---

## **gui\_wave\_add\_signal**

This command adds specified objects to the waveform view of a specified violation inspector window. If you specify a cell, a group is created in the waveform view and all the pins of the cell are added to this group as a list of signals. For a bus, all nets are added. The objects that are added will be selected.

The syntax of the command is

```
gui_wave_add_signal
[-window inspector_window]
[-clct list]
```

Options	Descriptions
-window <i>inspector_window</i>	Specifies a signal to be added to the specified violation inspector window. If <i>inspector_window</i> is not a valid violation in the inspector window, an error message appears and the command exits. If no -window option is specified, the signal is added to the waveform viewer of the first launched violation inspector.
-clct <i>list</i>	Specifies that <i>list</i> is to be considered as a collection of object handles. In the absence of the -clct option, <i>list</i> is considered as a collection of object names.

To add a port object *i\_rd*, for example, use the following syntax:

```
gui_wave_add_signal i_rd
# This command adds the port object to the first violation in
inspector window with a waveform view
```

To add selected objects, use the following syntax:

```
gui_wave_add_signal -window ViolationInspector.3 -clct [get_selection]
# Adds selected objects to the waveform view of the violation inspector named
ViolationInspector.3
```

---

### **guiViolationSchematicAddObjects**

This command adds specified objects to the schematic view of a specified violation inspector window and selects them.

The syntax of this command is

```
guiViolationSchematicAddObjects  
[-window inspector_window]  
[-clct list]
```

Options	Descriptions
-window <i>inspector_window</i>	<p>Specifies a signal to be added to the specified violation inspector window.</p> <p>If <i>inspector_window</i> is not a valid violation in the inspector window, an error message displays and the command exits.</p> <p>If no -window option is specified, the signal is added to the waveform viewer of the first launched violation inspector.</p>
-clct <i>list</i>	<p>Specifies that <i>list</i> is to be considered as a collection of object handles.</p> <p>In the absence of the -clct option, <i>list</i> is considered as a collection of object names.</p>

# 4

## Performing Scan Replacement

---

This chapter describes the scan replacement process, including constraint-optimized scan insertion.

The scan replacement process inserts scan cells into your design by replacing nonscan sequential cells with their scan equivalents. If you start with an HDL description of your design, scan replacement occurs during the initial mapping of your design to gates. You can also start with a gate-level netlist; in this case, scan replacement occurs as an independent process.

With either approach, scan synthesis considers the design constraints and the impact of both the scan cells themselves and the additional loading due to scan chain routing to minimize the impact of the scan structures on the design.

This chapter includes the following sections:

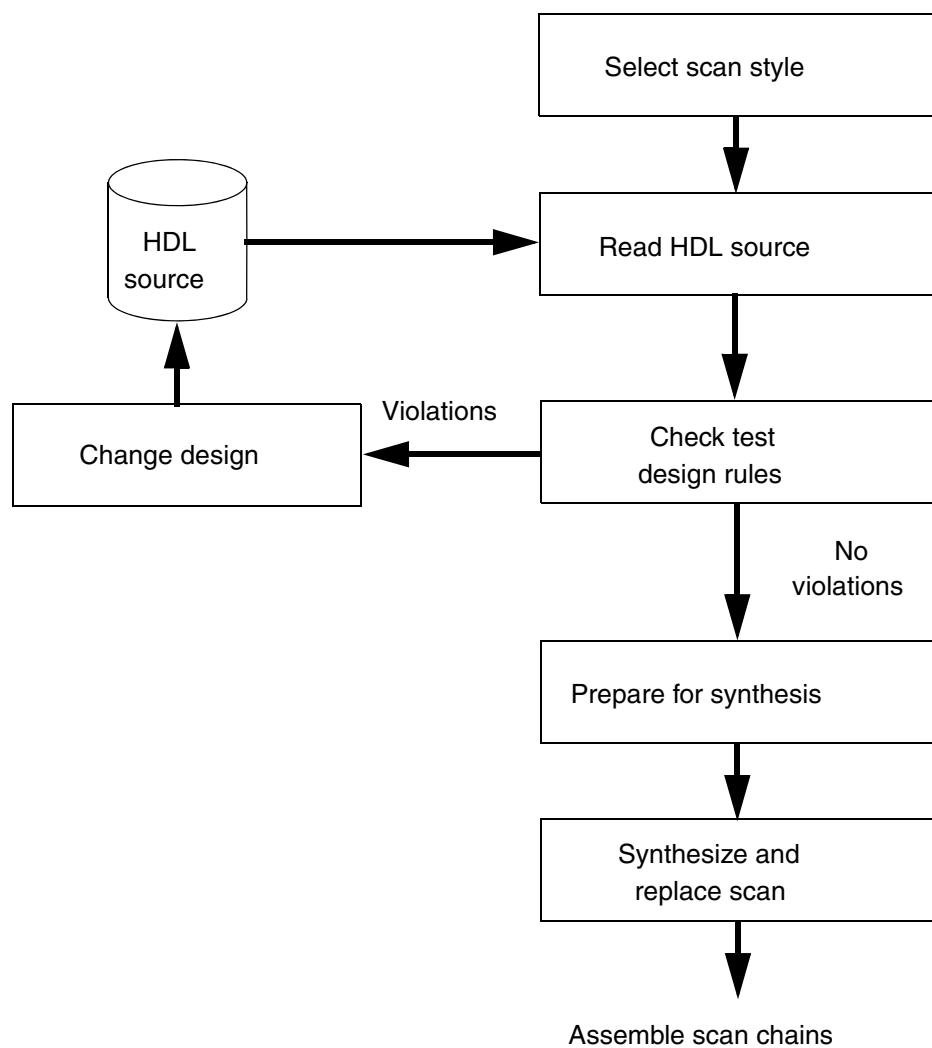
- [Scan Replacement Flow](#)
- [Preparing for Scan Replacement](#)
- [Specifying a Scan Style](#)
- [Verifying Scan Equivalents in the Technology Library](#)
- [Scan Cell Replacement Strategies](#)
- [Test-Ready Compilation](#)

- [Validating Your Netlist](#)
- [Performing Constraint-Optimized Scan Insertion](#)

## Scan Replacement Flow

Figure 4-1 shows the flow for the scan replacement process. This figure assumes that you are starting with an HDL description of the design. If you are starting with a gate-level netlist, you must use constraint-optimized scan insertion. (See “[Preparing for Constraint-Optimized Scan Insertion](#)” on page 4-35).

Figure 4-1 Synthesis and Scan Replacement Flow



The following steps explain the scan replacement process:

1. Select a scan style.

DFT Compiler requires a scan style to perform scan synthesis. The scan style dictates the appropriate scan cells to insert during optimization. You must select a single scan style and use this style on all the modules of your design.

2. Check test design rules of the HDL-level design description.
3. Synthesize your design.

Test-ready compile maps all sequential cells directly to scan cells. During optimization, DFT Compiler considers the design constraints and the impact of both the scan cells themselves and the additional loading due to scan chain routing to minimize the impact of the scan structures on the design.

---

## Preparing for Scan Replacement

This section discusses what to consider before starting the scan replacement process, and has the following subsections:

- [Selecting a Scan Replacement Strategy](#)
- [Identifying Barriers to Scan Replacement](#)
- [Preventing Scan Replacement](#)

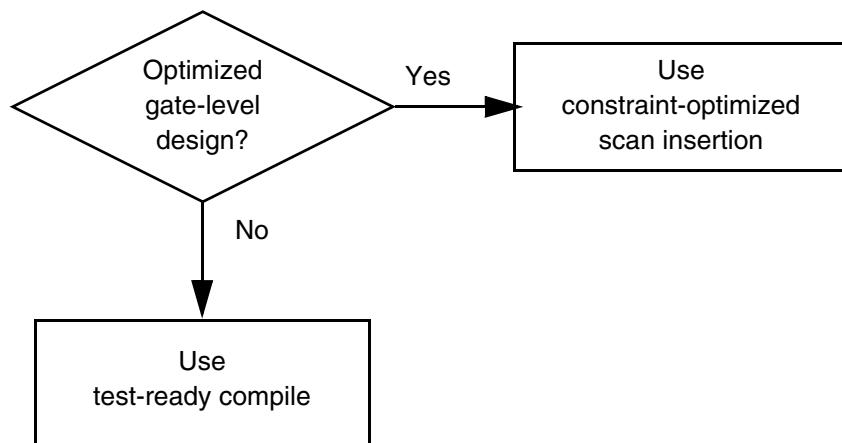
---

## Selecting a Scan Replacement Strategy

You should select the scan replacement strategy based on the status of your design. If you have an optimized gate-level design and will not be using the `compile` command to perform further optimization, you should use constraint-optimized scan insertion. In all other cases, you should use test-ready compile to insert the scan cells.

[Figure 4-2](#) shows how to determine the appropriate scan replacement strategy.

*Figure 4-2 Selecting a Scan Replacement Strategy*



Test-ready compile offers the following advantages:

- Single-pass synthesis

With test-ready compile, the Synopsys tools converge on true one-pass scan synthesis. As a practical matter, design constraints usually result in some cleanup and additional optimization after compile, but test-ready compile is more straightforward compared with other methods.

- Better quality of results

Test-ready compile offers better quality of results (QoR) compared with past methods. Including scan cells at the time of first optimization results in fewer design rule violations and other constraint violations due to scan circuitry.

- Simpler overall flow

Test-ready compile requires fewer optimization iterations compared with previous methods.

Note:

For details on constraint-optimized scan insertion, see “[Performing Constraint-Optimized Scan Insertion](#)” on page 4-32.

---

## Identifying Barriers to Scan Replacement

You should perform test DRC to identify conditions that prevent scan replacement.

Test DRC identifies the following conditions that prevent scan replacement:

- The technology library does not contain an appropriate scan cell for the sequential cell.
- DFT Compiler does not support scan replacement for the sequential cell.
- The sequential cell has an attribute that prevents scan replacement.
- An invalid net drives the clock pin of the sequential cell.
- An invalid net drives the asynchronous pin of the sequential cell.

The following sections provide details about each of these conditions.

Note:

For full details on performing test DRC, see [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

## **Technology Library Does Not Contain Appropriate Scan Cells**

If a scan equivalent does not exist for a sequential cell, scan replacement cannot occur for that cell. DFT Compiler generates the following message when a scan equivalent does not exist for a sequential cell:

```
Warning: No scan equivalent exists for cell %s (%s).  
(TEST-120)
```

This warning message can occur when

- The technology library does not contain scan cells.

If DFT Compiler does not find scan cells in the technology library, it also generates this message:

```
Warning: Target library for design contains no  
scan-cell models. (TEST-224)
```

Check with your semiconductor vendor to see if the vendor provides a technology library that supports scan synthesis.

- The technology library contains scan cells but does not support a scan equivalent for the nonscan cell.
- The technology library contains scan cells but incorrectly models the scan equivalent for the nonscan cell.

If DFT Compiler finds a scan cell in the technology library that is not the obvious replacement cell you expect, the reason could be that

- The chosen scan equivalent results in a lower-cost implementation overall.

- The technology library has a problem. In that case, contact the ASIC vendor for more information.

## Unsupported Sequential Cells

DFT Compiler supports sequential cells that have the following characteristics:

- During functional operation, the cell functions as a D flip-flop, a D latch, or a master-slave latch.
- During scan operation, the cell functions as a D flip-flop or a master-slave latch.
- The cell stores a single bit of data.

Edge-triggered cells that violate this requirement cause DFT Compiler to generate the following message:

Warning: Cell %s (%s) is not supported because it has too many states (%d states). This cell is being black-boxed. (TEST-462)

Master-slave latch pairs with extra states cause DFT Compiler to generate one of these messages, depending on the situation:

Warning: Master-slave cell %s (%s) is not supported because state pin %s is neither master nor slave. This cell is being black-boxed. (TEST-463)

Warning: Master-slave cell %s (%s) is not supported because there are two or more master states. This cell is being black-boxed. (TEST-464)

Warning: Master-slave cell %s (%s) is not supported because there are two or more slave states. This cell is being black-boxed. (TEST-465)

- The cell has a three-state output.

Cells that violate this requirement cause DFT Compiler to generate this message:

Warning: Cell %s (%s) is not supported because it is a sequential cell with three-state outputs. This cell is being black-boxed. (TEST-468)

- The cell uses a single clock per internal state. Note that the cell might use different clocks for functional and test operations.

Cells that violate this requirement cause DFT Compiler to generate one of these messages:

Warning: Cell %s (%s) is not supported because state pin %s has no clocks. This cell is being black-boxed.  
(TEST-466)

Warning: Cell %s (%s) is not supported because state pin %s is multi-port. This cell is being black-boxed.  
(TEST-467)

Your design contains unsupported sequential cells only if you instantiate them. DFT Compiler does not insert unsupported sequential cells in your design.

## Attributes That Prevent Scan Replacement

The following attributes prevent scan replacement:

- `scan_element false`

A `scan_element false` attribute excludes sequential cells from scan replacement. Just remember that nonscan sequential cells generally reduce the fault coverage results for full-scan designs.

- `dont_touch`

A `dont_touch` attribute on a flip-flop prevents scan replacement only on a Verilog netlist but still allows scan routing of the flip-flop if it is already scan replaced. If a .ddc file is used, the command `set_scan_configuration -exclude` must be used to prevent scan replacement, because the `dont_touch` attribute will not achieve this.

A `dont_touch` attribute also prevents the AutoFix feature from fixing a clock or reset violation on a flip-flop. For more information on AutoFix, see “[Using AutoFix](#)” on [page 7-14](#).

Be aware that a `dont_touch` attribute on the top-level design does not prevent scan replacement on the design.

These attributes differ in that the `dont_touch` attribute can be assigned at any level, but the `scan_element` attribute is ignored unless you apply it at the same level where you issue the `insert_dft` command.

DFT Compiler generates this message when a sequential cell has a `scan_element false` attribute:

Information: Cell %s (%s) will not be scanned due to a or `set_scan_element` command. (TEST-202)

DFT Compiler generates this message when a nonscan sequential cell has a `dont_touch` attribute:

Information: Cell %s (%s) could not be made scannable as it is dont\_touched. (TEST-121)

The `set_dont_touch` command is intended to prevent logic optimization and should not be used to prevent scan replacement.

If you do not want to exclude the affected cells from scan replacement, locate and remove the attribute causing the violation. Locate the attribute by using the `get_attribute` or `report_attribute` command. Remove the attribute by using the `remove_attribute` command.

## Invalid Clock Nets

The term “system clock” refers to a clock used in the parallel capture cycle. The term “test clock” refers to a clock used during scan shift. Multiplexed flip-flop designs use the same clock as both the system clock and the test clock.

In a nonscan design, an invalid clock net, whether a system clock or a test clock, prevents scan replacement of all sequential cells driven by that clock net.

The requirements for valid clocks in DFT Compiler include the following:

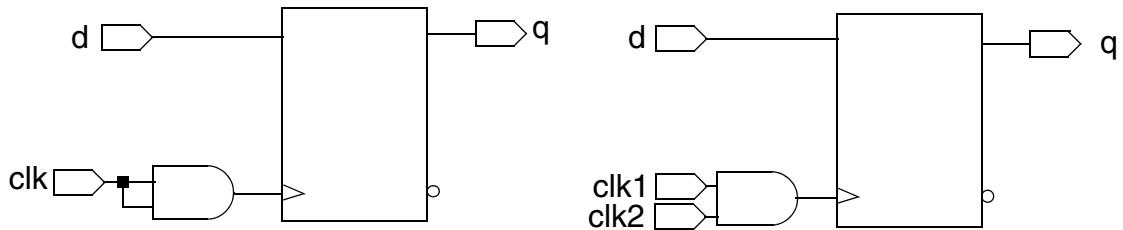
- A system or test clock used during scan testing must be driven from a single top-level port.  
An active clock edge at a sequential cell must be the result of a clock pulse applied at a top-level port, not the result of combinational logic driving the clock net.
- A system or test clock used during scan testing can be driven from a bidirectional port.  
DFT Compiler supports the use of bidirectional ports as clock ports if the bidirectional ports are designed as input ports during chip test. If a bidirectional port drives a clock net but the port is not designed as an input port during chip test mode, DFT Compiler forces the net to X and cells clocked by the net to become black box sequential cells.
- A system or test clock used during scan testing must be generated in a single tester cycle.  
The clock pulse applied at the clock port must reach the sequential cells in the same tester cycle. DFT Compiler does not support sequential gating of clocks, such as clock divider circuitry.
- A system or a test clock used during scan testing cannot be the result of multiple clock inputs.  
DFT Compiler does not support the use of combinationally combined clock signals, even if the same port drives the signal.

Note:

If the same port drives the组合ally combined clock signal, as shown in the design on the left in [Figure 4-3](#) or the design in [Figure 4-4](#), DFT Compiler does not detect the problem in nonscan or unrouted scan designs.

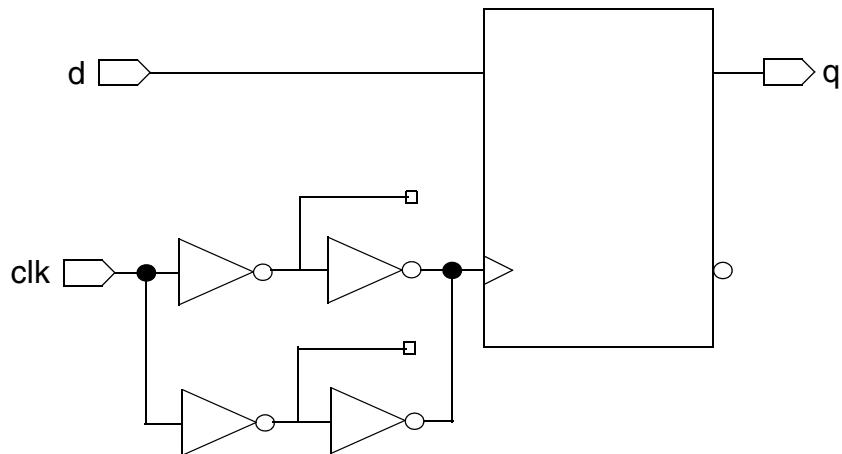
[Figure 4-3](#) shows sample designs that use组合ally combined clocks. When multiple clock signals drive a clock net, DFT Compiler forces the net to X and cells clocked by the net become black box sequential cells.

*Figure 4-3 Examples of Combinationally Combined Clock Nets*



By default, DFT Compiler supports the use of reconvergent clocks, such as clocks driven by parallel clock buffers. [Figure 4-4](#) shows an example of a design that uses a reconvergent clock net. To prevent `dft_drc` from allowing such reconvergent clocks in your design, set the `test_allow_clock_reconvergence` variable to false.

*Figure 4-4 Example of a Reconvergent Clock Net*



- A test clock must remain active throughout the scan shift process.

To load the scan chain reliably, make sure the test clock remains active until scan shift completes. For组合逻辑 gated clocks, you must configure the design to disable the clock gating during scan shift.

DFT Compiler supports combinational clock gating during the parallel capture cycle.

Test design rule checking on a nonscan design might not detect invalid clock nets. DFT Compiler identifies all invalid clock nets only in existing scan designs.

DFT Compiler cannot control the clock net when

- A sequential cell drives the clock net
- A multiplexer with an unspecified select line drives the net of a test clock
- Combinational clock-gating logic can generate an active edge on the clock net

DFT Compiler generates this message when it detects an uncontrollable clock:

Warning: Normal mode clock pin %s of cell %s (%s) is uncontrollable. (TEST-169)

Because uncontrollable clock nets prevent scan replacement, you should correct uncontrollable clocks. Sequentially driven clocks require test-mode logic to bypass the violation. You can bypass violations caused by other sources of uncontrollable clocks by using test configuration or test-mode logic.

DFT Compiler can control a combinational clock that cannot generate an active clock edge. However, DFT Compiler considers this type of clock invalid, because the clock might not remain active throughout scan shift. In this case, DFT Compiler generates this message:

Warning: Shift clock pin %s of cell %s (%s) is illegally gated. (TEST-186)

Because invalid gated-clock nets prevent scan replacement, you should correct invalid gated clocks. You can use AutoFix to bypass invalid gated clocks when using the multiplexed flip-flop scan style. You might also be able to change the test configuration to bypass the violation. For more information on AutoFix, see “[Using AutoFix](#)” on page 7-14.

## Invalid Asynchronous Pins

DFT Compiler considers a net that drives an asynchronous pin as valid if it can disable the net from an input port or from a combination of input ports. DFT Compiler cannot control an asynchronous pin driven by ungated sequential logic.

In a nonscan design, a net with an uncontrollable asynchronous pin prevents scan replacement of all sequential cells connected to that net.

DFT Compiler generates this message when it detects an uncontrollable asynchronous pin:

```
Warning: Asynchronous pins of cell FF_A (FD2) are  
uncontrollable. (TEST-116)
```

Because nets with an uncontrollable asynchronous pin prevent scan replacement, you should correct uncontrollable nets. Use AutoFix if you are using the multiplexed flip-flop scan style, test configuration, or test-mode logic to bypass uncontrollable asynchronous pin violations.

---

## Preventing Scan Replacement

When running test-ready compile on a gate-level netlist, DFT Compiler respects the following attributes that prevent scan replacement:

- dont\_touch
- scan\_element false

When either of these attributes exists on a mapped sequential cell, test-ready compile does not replace the cell with a scan equivalent.

---

## Specifying a Scan Style

This section explains the process for selecting and specifying a scan style for your design. It has the following subsections:

- [Types of Scan Styles](#)
  - [Scan Style Considerations](#)
  - [Setting the Scan Style](#)
- 

## Types of Scan Styles

DFT Compiler supports the scan styles listed in the following subsections:

- [Multiplexed Flip-Flop Scan Style](#)
- [Clocked Scan Style](#)
- [LSSD Scan Style](#)

Note:

The term LSSD refers to LSSD and clocked LSSD.

This section describes each scan style in detail.

## Multiplexed Flip-Flop Scan Style

DFT Compiler supports multiplexed flip-flop scan equivalents for D, JK, and master-slave flip-flops and for D latches. The multiplexed flip-flop scan equivalents for all flip-flop styles must be fully functionally modeled in the technology library. This scan style has the following advantages:

- Multiplexed flip-flop is the most widely known and understood scan style.
- If the technology library does not contain scan cells, you can create flip-flop scan equivalents from discrete multiplexer and flip-flop components. To use this technique, you must add a dummy scan cell to the technology library.

The multiplexed flip-flop scan style has the disadvantage that hold-time or clock skew problems can occur on the scan path because of a short path from a scan cell's scan output pin to the next scan cell's scan input pin. DFT Compiler can reduce the occurrence of these problems by considering hold-time violations during optimization.

## Clocked Scan Style

DFT Compiler supports clocked-scan equivalents for D and JK flip-flops and for D latches. The clocked-scan style is well suited for use in multiple-clock designs because of the dedicated test clock.

The clocked-scan style also has some disadvantages:

- Hold-time or clock skew problems can occur on the scan path because the path from a scan cell's scan output pin to the next scan cell's scan input pin is too short. DFT Compiler can reduce the occurrence of these problems by considering hold-time violations during optimization.
- This scan style requires the routing of two edge-triggered clocks. Routing clock lines is difficult because you must carefully control the clock skew.

## LSSD Scan Style

DFT Compiler supports Level-Sensitive Scan Design (LSSD) equivalents for D, JK, and master-slave flip-flops and for D latches. Timing problems on the scan path are unlikely in LSSD designs because of the use of nonoverlapping two-phase clocks during the scan operation.

The LSSD scan style also has some disadvantages:

- This scan style requires a greater wiring area than the multiplexed flip-flop or clocked-scan styles.

- DFT Compiler does not support the more complex LSSD cells, such as multiple data port master latches.

When you use the LSSD scan style, define the clock waveforms so that the master and slave clocks have nonoverlapping waveforms because master and slave latches should never be active simultaneously.

---

## Scan Style Considerations

You must select a single scan style and use this style for all modules of your design. Consider the following questions when selecting a scan style:

- Which scan styles are supported in your technology library?

To make it possible to implement internal scan structures in the scan style you select, appropriate scan cells must be present in the technology libraries specified in the target\_library variable.

Use of sequential cells that do not have a scan equivalent always results in a loss of fault coverage in full-scan designs. Techniques to verify scan equivalents are discussed in [“Verifying Scan Equivalents in the Technology Library” on page 4-16](#).

- What is your design style?

If your design is predominantly edge-triggered, use the multiplexed flip-flop, clocked scan, or clocked LSSD scan style.

If your design has a mix of latches and flip-flops, use the clocked scan or LSSD scan style.

If your design is predominantly level-sensitive, use the LSSD scan style.

- How complete are the models in your technology library?

The quality and accuracy of the scan and nonscan sequential cell models in the Synopsys technology library affect the behavior of DFT Compiler. Incorrect or incomplete library models can cause incorrect results during test design rule checking.

DFT Compiler requires a complete functional model of a scan cell to perform test design rule checking. The Library Compiler UNIGEN model supports complete functional modeling of all supported scan cells. However, the usual sequential modeling syntax of Library Compiler supports only complete functional modeling for multiplexed flip-flop scan cells.

When the technology library does not provide a functional model for a scan cell, the cell is a black box for DFT Compiler.

For information on the scan cells in the technology library you are using, see your ASIC vendor. For information on creating technology library elements or to learn more about modeling scan cells, see the information about defining test cells in the Library Compiler documentation.

---

## Setting the Scan Style

You can specify the scan style by using either the `test_default_scan_style` variable or the `set_scan_configuration -style` command:

- `test_default_scan_style` – applies to all designs in the current session. The syntax is as follows:

```
set test_default_scan_style style
```

- `set_scan_configuration -style` – applies only to the current design. If your selected scan style differs from the default scan style, you must execute this command for each module. The syntax is as follows:

```
set_scan_configuration -style style
```

[Table 4-1](#) shows the scan style keywords used to specify the scan style. Use these keywords with either the `test_default_scan_style` variable or the `set_scan_configuration -style` command.

*Table 4-1 Scan Style Keywords*

Scan style	Keyword
Multiplexed flip-flop	<code>multiplexed_flip_flop</code>
Clocked scan	<code>clocked_scan</code>
Level-sensitive scan design	<code>lssd</code>

The following examples show how to specify the scan style by using either the `test_default_scan_style` variable or the `set_scan_configuration` command:

```
dc_shell> set test_default_scan_style \
           multiplexed_flip_flop

dc_shell> set_scan_configuration -style clocked_scan
```

---

## Verifying Scan Equivalents in the Technology Library

Before starting scan synthesis, you need to confirm that your technology library contains scan cells and then verify that the scan cells are suitable for the selected scan style.

This section has the following subsections:

- [Checking the Technology Library for Scan Cells](#)
  - [Checking for Scan Equivalents](#)
- 

### Checking the Technology Library for Scan Cells

You can determine whether the technology library contains scan cells by using either of the following methods:

- Search the library .ddc file.

Every scan cell, regardless of the scan style, must have a scan input pin and a scan output pin. You can determine whether the technology library contains scan cells by using the `filter` command to search for scan input or scan output pins.

Depending on its polarity, a scan input pin can have a `signal_type` attribute of either `test_scan_in` or `test_scan_in_inverted` in the technology library. A scan output pin can have a `signal_type` attribute of either `test_scan_out` or `test_scan_out_inverted` in the technology library, depending on its polarity.

The following command sequence shows the use of the `filter` command:

```
dc_shell> read_ddc class.ddc  
dc_shell> get_pin class/*/* -filter "@signal_type \  
= test_scan_in"
```

If the library contains scan cells, the `filter` command returns a list of pins; if the library does not contain scan cells, the `filter` command returns an empty list.

- Check the test design rules.

As one of the first checks it performs, the `dft_drc` command determines the presence of scan cells in the technology library. If the technology libraries identified in the `target_library` variable do not contain scan cells, `dft_drc` generates the following message:

```
Warning: Target library for design contains no  
scan-cell models. (TEST-224)
```

You must define the `current_design` before you run the `dft_drc` command.

If your technology library does not contain scan cells, check with your semiconductor vendor to see if the vendor provides a technology library that supports test synthesis.

---

## Checking for Scan Equivalents

To verify that the technology library contains scan equivalents for the sequential cells in your design, run the `dft_drc` command on your design or on a design containing the sequential cells likely to be used in your design.

If the technology library does not contain a scan equivalent for a sequential cell in a nonscan design, `dft_drc` generates the following message:

```
Warning: No scan equivalent exists for cell instance  
(reference). (TEST-120)
```

In verbose mode (`dft_drc -verbose`), the TEST-120 message lists all scan equivalent pairs available in the target library in the selected scan style. If the target library contains no scan equivalents in the chosen scan style, no scan equivalents are listed.

Suppose you have a design containing D flip-flops but the target technology library contains scan equivalents only for JK flip-flops. [Example 4-1](#) shows the warning message issued by `dft_drc`, along with the scan equivalent mappings to the available scan cells.

### *Example 4-1 Scan Equivalent Listing*

```
Warning: No scan equivalent exists for cell q_reg (FD1P).  
(TEST-120)
```

Scan equivalent mappings for target library are:

FJK3	-> FJK3S
FJK2	-> FJK2S
FJK1	-> FJK1S

---

## Scan Cell Replacement Strategies

This section describes how to select the set of scan cells and multibit components to use in your scan replacement strategy.

It includes the following subsections:

- [Specifying Scan Cells](#)
- [Multibit Components](#)

---

## Specifying Scan Cells

Before you perform scan cell replacement, you need to specify the set of scan cells to be used by DFT Compiler. This section has the following subsections:

- [Restricting the List of Available Scan Cells](#)
- [Sample Scan Cell Replacement Strategies](#)
- [Mapping Sequential Gates in Scan Replacement](#)

### Restricting the List of Available Scan Cells

The `set_scan_register_type` command lets you specify which flip-flop scan cells are to be used by `compile -scan` to replace nonscan cells. The command restricts the choices of scan cells available for scan replacement. You can apply this restriction to the current design, to particular designs, or to particular cell instances in the design.

Note:

The `set_scan_register_type` command applies to the operation of both the `compile -scan` command and the `insert_dft` command.

The `set_scan_register_type` command has the following syntax:

```
set_scan_register_type [-exact]
                      -type scan_flip_flop_list [cell_or_design_list]
```

The `scan_flip_flop_list` is the list of scan cells that `compile -scan` is allowed to use for scan replacement. There must be at least one such cell named in the command. Specify each scan cell by its cell name alone, without the library name.

The `cell_or_design_list` is a list of designs or cell instances where the restriction on scan cell selection is to be applied. In the absence of such a list, the restriction applies to the current design, set by the `current_design` command, and to all lower-level designs in the design hierarchy.

The `-exact` option determines whether the restriction on scan cell selection also applies to back-end delay and area optimization done by `insert_dft` or by subsequent synthesis operations such as `compile -incremental`. If `-exact` is used, the restriction still applies to back-end optimization. In other words, scan cells can be replaced only by other scan cells in the specified list. If `-exact` is not used, the optimization algorithm is free to use any scan cell in the target library.

### Sample Scan Cell Replacement Strategies

Here are some examples of `set_scan_register_type` commands:

```
dc_shell> set_scan_register_type -exact -type FD1S
```

This command causes `compile -scan` to use only FD1S scan cells to replace nonscan cells in the current design. Because of the `-exact` option, this restriction applies to both initial scan replacement and subsequent optimization.

```
dc_shell> set_scan_register_type -exact \
           -type {FD1S FD2S} {add2 U1}
```

This command causes `compile -scan` to use only FD1S or FD2S scan cells to replace each nonscan cell in all designs and cell instances named add2 or U1. In all other designs and cell instances, `compile -scan` can use any scan cells available in the target library. The `-exact` option forces any back-end delay optimization to respect the scan cell list, thus allowing only FD1S and FD2S to be used.

```
dc_shell> set_scan_register_type \
           -type {FD1S FD2S} {add2 U1}
```

This command is the same as the one in the previous example, except that the `-exact` option is not used. This means that the back-end optimization algorithm is free to replace the FD1S and FD2S cells with any compatible scan cells in the target library.

If you use the `set_scan_register_type` command on generic cell instances, be sure to include the `-scan` option in the `compile` command. Otherwise, the scan specification will be lost.

To report scan paths, scan chains, and scan cells in the design, use the `report_scan_path` command, as shown in the following examples:

```
dc_shell> report_scan_path -view existing_dft \
           -chain all
```

```
dc_shell> report_scan_path -view existing_dft -cell all
```

To cancel all `set_scan_register_type` settings currently in effect, execute the following command:

```
dc_shell> remove_scan_register_type
```

## Mapping Sequential Gates in Scan Replacement

To use the `set_scan_register_type` command effectively, understanding the scan replacement process is important.

The `compile -scan` command maps sequential gates into scan flip-flops and latches, using three steps:

1. The `compile -scan` command maps each sequential gate in the generic design description into an initial nonscan latch or flip-flop from the target library. In the absence of any `set_scan_register_type` specification, `compile -scan` chooses the smallest area-cost flip-flop or latch. For a design or cell instance that has a `set_scan_register_type` setting in effect, `compile -scan` chooses the nonscan equivalent of a scan cell in the `scan_flip_flop_list`.
2. The `compile -scan` command replaces the nonscan flip-flops with scan flip-flops, using only the scan cells specified in the `set_scan_register_type` command, where applicable. If `compile -scan` is unable to use a scan cell from the `scan_flip_flop_list`, it uses the best matching scan cell from the target library and issues a warning.
3. If the `-exact` option is not used in the `set_scan_register_type` command, Design Compiler and DFT Compiler attempt to remap each scan flip-flop into another component from the target library to optimize the delay or area characteristics of the circuit. If the `-exact` option is used, optimization is restricted to using the scan cells in the `scan_flip_flop_list`.

The operation of step 1 can be controlled by the `set_register_type` command in Design Compiler. The `set_register_type` command specifies a list of allowed cells for implementing functions specified in the HDL description of the design, but you need to be careful about using this command in conjunction with scan replacement. For example, if you tell the `compile` command to use a sequential cell that has no scan equivalent, DFT Compiler will not be able to replace the cell with a corresponding scan cell.

The `set_scan_register_type` command affects only the replacement of nonscan cells with scan cells. It cannot be used to force existing scan cells to be replaced by new scan cells. To make this type of design change, you need to go back to the original nonscan design and apply a new `set_scan_register_type` specification, followed by a new `compile -scan` or `insert_dft` operation.

## Multibit Components

Multibit components are supported by DFT Compiler during scan replacement. This section has the following subsections:

- [What Are Multibit Components?](#)
- [How DFT Compiler Assimilates Multibit Components](#)
- [Controlling Multibit Test Synthesis](#)
- [Performing Multibit Component Scan Replacement](#)

## What Are Multibit Components?

A multibit component is a sequence of cells with identical functionality. It can consist of single-bit cells or the set of multibit cells supported by Design Compiler. Cells can have identical functionality even if they have different bit widths. Multibit synthesis ensures regularity and predictability of layout.

HDL Compiler infers multibit components through HDL directives. See the *HDL Compiler (Presto Verilog) Reference Manual* for more information about multibit inference. Specify multibit components by using the Design Compiler `create_multibit` command and `remove_multibit` command. Control multibit synthesis by using the `set_multibit_options` command. See the *Design Compiler Reference Manual: Optimization and Timing Analysis* for details.

When you create a new multibit component with the `create_multibit` command, choose a name that is different from the name of any existing object in your design. This will prevent possible conflicts later when you use the `set_scan_path` command.

Structured logic synthesis is a special case of multibit synthesis in which the individual bits of a multibit component are implemented as distinct elements. Use the `set_multibit_options -mode structured` command to enable structured logic synthesis.

## How DFT Compiler Assimilates Multibit Components

Multibit components have the following properties:

- All the synthesis and optimization that DFT Compiler performs is as prescribed by the multibit mode in effect.
- Scan chain allocation and routing result in a layout that is as regular as possible.

To achieve these goals, DFT Compiler assimilates sequential multibit components into synthesizable segments.

A synthesizable segment, an extension of the user segment concept, has the following properties:

- Its implementation is not fixed at the time of specification.
- It consists of a name and a sequence of cells that implicitly determine an internal routing order.
- It lacks access pins and possibly internal routing.
- It does not need to be scan-replaced.
- Test synthesis controls the implementation.

A synthesizable segment that cannot be synthesized into a valid user segment is invalid. Only multibit synthesizable segments are supported.

## Controlling Multibit Test Synthesis

You control multibit test synthesis through the specification of the scan configuration by using the following commands:

- `set_scan_configuration`
- `reset_scan_configuration`
- `set_scan_path`
- `set_scan_element`

Commands that accept segment arguments also accept multibit components. You can refer by instance name to multibit components from the top-level design through the design hierarchy. Commands that accept sets of cells also accept multibit components. When you specify a multibit component as being a part of a larger segment, the multibit component is included in the larger user-defined segment without modification.

## Performing Multibit Component Scan Replacement

Use the `compile -scan` command or the `insert_dft` command to perform multibit component scan replacement. These commands perform a homogeneous scan replacement. Bits of a multibit component are either all scan-replaced or all not scan-replaced. Bits are then assembled into multibit cells as specified by the `set_multibit_options` command.

The number of cells after scan replacement can change. For example, a 4-bit cell can be scan-replaced by two 2-bit cells. If this occurs, the two 2-bit cells get new names. If the cell is scan-replaced with a cell of equal width, a 4-bit cell replaced by a 4-bit cell for example, the name of the cell remains the same.

You control the scan replacement of multibit components by using the `set_scan_element` command.

When specifying individual cells by using either of these commands, do not specify an incomplete multibit component unless you previously disabled multibit optimization.

## Disabling Multibit Component Support

You can disable structured logic and multibit component support by doing one of the following:

- Remove some or all of the multibit components by using the `remove_multibit` command.

- Turn off scan synthesis by using the `set_scan_configuration -multibit_segments false` command.
  - Remove a previously defined scan path by using the `remove_scan_path` command.
- 

## Test-Ready Compilation

Scan cell replacement works most efficiently if it is done when you compile your design. This section describes the following topics related to the test-ready compilation process:

- [What Is Test-Ready Compile?](#)
  - [Preparing for Test-Ready Compile](#)
  - [Controlling Test-Ready Compile](#)
  - [Comparing Default Compile and Test-Ready Compile](#)
  - [Complex Compile Strategies](#)
- 

### What Is Test-Ready Compile?

Test-ready compile integrates logic optimization and scan replacement. During the first synthesis pass of each HDL design or module, test-ready compile maps all sequential cells directly to scan cells. The optimization cost function considers the impact of the scan cells themselves and the additional loading due to the scan chain routing. By accounting for the timing impact of internal scan design from the start of the synthesis process, test-ready compile eliminates the need for an incremental compile after scan insertion.

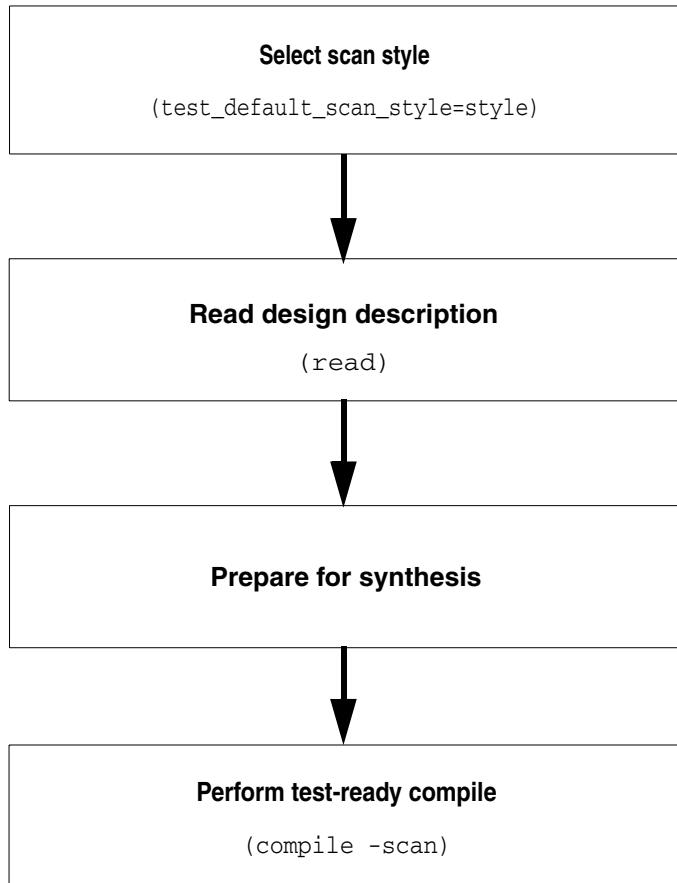
During optimization, DFT Compiler cannot determine whether the sequential cells in your HDL description meet the test design rules, so it maps all sequential cells to scan cells. Later in the scan synthesis process, DFT Compiler can convert some sequential cells back to nonscan cells. For example, test design rule checking might find scan cells with test design rule violations. In other circumstances, you might manually specify some sequential cells as nonscan elements. In such cases, DFT Compiler converts the scan cells to nonscan equivalents during execution of the `insert_dft` command.

Typically, the input to test-ready compile is an HDL design description. You can also perform test-ready compile on a nonscan gate-level netlist that requires optimization. For example, a gate-level netlist resulting from technology translation usually requires logic optimization to meet constraints. In such a case, use test-ready compile to perform scan replacement.

## The Test-Ready Compile Flow

Figure 4-5 shows the test-ready compile flow and the commands required to complete this flow.

Figure 4-5 Test-Ready Compile Flow



Before performing test-ready compile:

- Select a scan style

For information about selecting a scan style, see “[Specifying a Scan Style](#)” on page 4-12.

- Prepare for logic synthesis

For information about preparing for logic synthesis, see “[Preparing for Test-Ready Compile](#)” on page 4-25.

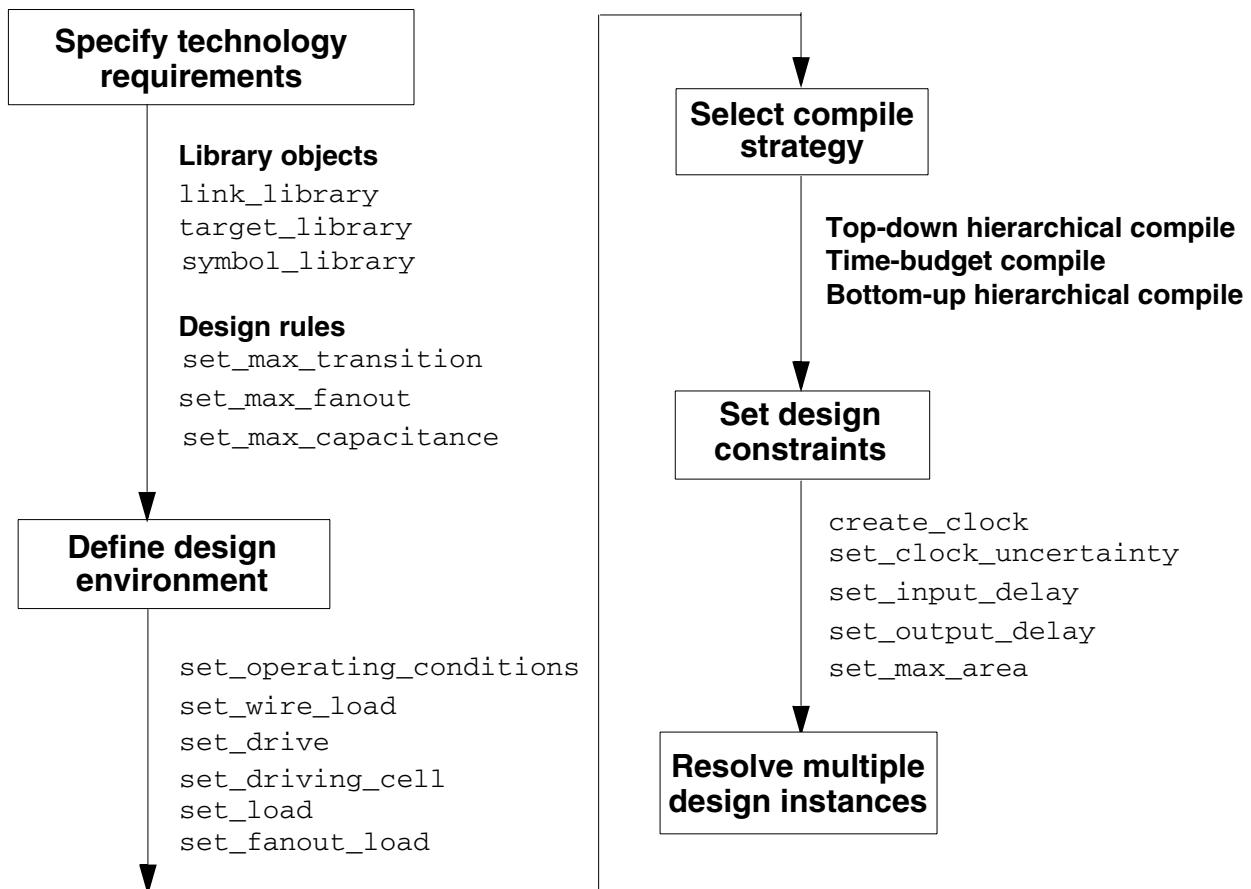
The result of test-ready compile is an optimized design that contains unrouted scan cells. The optimization performed during test-ready compile accounts for both the impact of the scan cells and the additional loading due to the scan chain routing. A design in this state is known as an *unrouted scan design*.

---

## Preparing for Test-Ready Compile

Figure 4-6 shows the synthesis preparation steps. For more information about these steps, see the *Design Compiler User Guide*.

Figure 4-6 Synthesis Preparation Steps



## Performing Test-Ready Compile in the Logical Domain

The `compile -scan` command invokes test-ready compile. You must enter this command from the `dc_shell-xg-t` command line; the Design Analyzer menus do not support the `-scan` option.

```
dc_shell> compile -scan
```

For details of how to constrain your design and for other compile options, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

---

## Controlling Test-Ready Compile

You can use the following variable and commands to control scan implementation by `compile -scan`:

- `test_default_scan_style` or  
`set_scan_configuration -style`
- `set_scan_element element_name true | false`
- `set_scan_register_type [-exact]  
-type scan_flip_flop_list [cell_or_design_list]`
- `set_scan_configuration -multibit_segments`

The `test_default_scan_style` variable determines which scan style the `compile -scan` command uses for scan implementation. You can also use the `set_scan_configuration -style` command for the same purpose.

You might not want to include a particular element in a scan chain. If this is the case, first analyze and elaborate the design. Then, use the `set_scan_element false` command in the generic technology (GTECH) sequential element. Subsequently, when you use the `compile -scan` command, this element is implemented as an ordinary sequential element and not as a scan cell. The following example shows a script that uses the `set_scan_element -false` command on generics:

```
analyze -format VHDL -library WORK switch.vhd
elaborate -library WORK -entity switch -arch rtl
set_scan_element false Q_reg
compile -scan
```

Note:

Use the `set_scan_element false` statement sparingly. For combinational ATPG, using nonscan elements generally results in lower fault coverage.

You might want to specify which flip-flop scan cells are to be used for replacing nonscan cells in the design. In that case, use the `set_scan_register_type` command as described in “[Specifying Scan Cells](#)” on page 18

---

## Comparing Default Compile and Test-Ready Compile

The following example shows the effect of test-ready compile on a small design. The Verilog description shown in [Example 4-2](#) and the VHDL description shown in [Example 4-3](#) each describe a small design containing two flip-flops: one a simple D flip-flop and one a flip-flop with a multiplexed data input.

### *Example 4-2 Verilog Design Example*

```
module example (d1,d2,d3,sel,clk,q1,q2);
  input d1,d2,d3,sel,clk;
  output q1,q2;
  reg q1,q2;
  always @ (posedge clk) begin
    q1 = d1;
    if (sel) begin
      q2=d2;
    end else begin
      q2=d3;
    end
  end
endmodule
```

The following command sequence performs the default compile process on the Verilog design example:

```
dc_shell> set target_library class.db
dc_shell> read_file -format verilog example.v
dc_shell> set_max_area 0
dc_shell> compile
```

### *Example 4-3 VHDL Design Example*

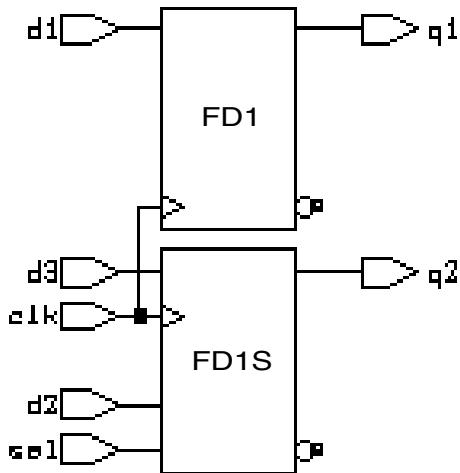
```
-----  
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_arith.all;  
-----  
entity EXAMPLE is  
    port( d1:in      STD_LOGIC;  
          d2:in      STD_LOGIC;  
          d3:in      STD_LOGIC;  
          sel:in     STD_LOGIC;  
          clk:in     STD_LOGIC;  
          q1:out     STD_LOGIC;  
          q2:out     STD_LOGIC  
    );  
end EXAMPLE;  
-----  
architecture RTL of EXAMPLE is  
begin  
    process  
    begin  
        wait until (clk'event and clk = '1');  
        q1 <= d1;  
        if (sel = '1') then  
            q2 <= d2;  
        else  
            q2 <= d3;  
        end if;  
    end process;  
end RTL;
```

The following command sequence performs the default compile process on the VHDL design example:

```
dc_shell> set target_library class.db  
dc_shell> analyze -format vhdl \  
           -library work example.vhd  
dc_shell> elaborate -library work EXAMPLE  
dc_shell> set_max_area 0  
dc_shell> compile
```

[Figure 4-7](#) shows the result of the default compile process on the design example. Design Compiler uses the D flip-flop (FD1) and the multiplexed flip-flop scan cell (FD1S) from the class technology library to implement the specified functional logic.

Figure 4-7 Gate-Level Design: Default Compile



Using default compile increases the scan replacement runtime and can result in sequential cells that do not have scan equivalents.

To invoke test-ready compile, specify the scan style before optimization and use the `-scan` option of the `compile` command.

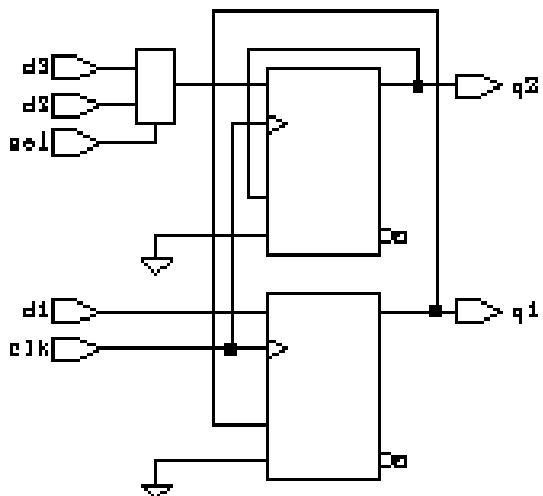
```
dc_shell> set test_default_scan_style \
           multiplexed_flip_flop
dc_shell> compile -scan
```

[Figure 4-8](#) shows the result of the test-ready compile process on the design example. During test-ready compile, DFT Compiler

- Implements the scan equivalent cells by using the multiplexed flip-flop scan cell (FD1S)
- Ties the scan enable pins (SE) to logic 0 to enable the functional logic
- Connects the Q output of the flip-flops to the scan input pins (SI) to reflect the scan loading effect

During scan routing, DFT Compiler replaces the temporary scan connections with the appropriate scan connections.

Figure 4-8 Gate-Level Design: Test-Ready Compile



A scan equivalent might not exist for the exact implementation defined, such as for the simple D flip-flop in the previous example. If the target library contains a scan cell that can be degenerated to the required implementation. For example, if the target library contains a scan cell with asynchronous pins that can be tied off, test-ready compile automatically uses that scan cell.

---

## Complex Compile Strategies

For larger designs or for designs with more aggressive timing goals, you might need to use more complex compile strategies, such as bottom-up compile, or you might need to use incremental compile a number of times. To include test-ready compile in your compile scripts, always use the `-scan` option of the `compile` command when compiling each current design, even if there are no sequential elements in the top level of the current design.

[Example 4-4](#) illustrates this guideline. It shows you how to perform a bottom-up compile for the a design, TOP, that has no sequential elements at the top level but instantiates two sequential modules A and B. (For clarity, details on how you might constrain the designs are omitted.) Note that the `compile -scan` command is used at the top level even though there are no sequential elements at the top level of the design.

#### *Example 4-4 Bottom-Up Compile Script*

```
dc_shell> current_design A
dc_shell> compile -scan

dc_shell> current_design B
dc_shell> compile -scan

dc_shell> current_design TOP
dc_shell> compile -scan
```

---

## Validating Your Netlist

Before you assemble the scan structures, you need to use the `link` and `check_design` commands to check the correctness of your design. You should fix any errors reported by these commands to guarantee the maximum fault coverage.

This section discusses the procedures for running the `link` and `check_design` commands.

---

### Running the link Command

The `link` command attempts to find models for the references in your design. The command searches the design files and library files defined by the `link_library` variable. If the `link_library` variable does not specify a path for a design file or library file, the `link` command uses the directory names defined in the search path. Specifying the asterisk character (\*) in the `link_library` variable forces the `link` command to search the designs in memory. See the man pages for more information about the `link` command.

If the `link` command reports unresolved references, such as missing designs or library cells in the netlist, resolve these references to provide a complete netlist to DFT Compiler. DFT Compiler operates on the complete netlist. DFT Compiler does not know the functional behavior of a missing cell, so it cannot predict the output of that cell. As a result, output from the missing reference is not observable. Each missing reference results in a large number of untestable faults in the vicinity of that cell and lower total fault coverage.

If the unresolved reference involves a simple cell, you can often fix the problem by adding the cell to the library or by replacing the reference with a valid library cell.

Handling a compiled cell requires a more complex solution. If the compiled cell does not contain internal gates, such as a ROM or programmable logic array, you can compile a behavioral model of the cell into gates and then run DFT Compiler on the equivalent gates.

For more information about missing references or link errors, see the *Design Compiler User Guide*.

---

## Running the `check_design` Command

The `check_design` command reports electrical design errors, such as port mismatches and shorted outputs that might lower fault coverage. For best fault coverage results, correct any design errors identified in your design. For more information about the `check_design` command, see the *Design Compiler User Guide*.

---

## Performing Constraint-Optimized Scan Insertion

During the scan replacement process, constraint-optimized scan insertion does the following:

- Inserts the scan cells
- Optimizes the scan logic, based on the design constraints
- Fixes all compile-related design rule violations

Scan insertion is the process of performing scan replacement and scan assembly in a single step. You use the `insert_dft` command to invoke constraint-optimized scan insertion. However, you can also perform scan replacement and scan assembly in separate steps.

This section contains the following subsections related to constraint-optimized scan insertion:

- [Supported Scan States](#)
  - [Locating Scan Equivalents](#)
  - [Preparing for Constraint-Optimized Scan Insertion](#)
  - [Scan Insertion](#)
- 

## Supported Scan States

Constraint-optimized scan insertion supports mixed scan states during scan insertion. Modules can have the following scan states:

- Nonscan

The design contains nonscan sequential cells. Constraint-optimized scan insertion must scan-replace and route these cells.

- Unrouted scan

The design contains unrouted scan cells. These unrouted scan cells can result from test-ready compile or from the scan replacement phase of constraint-optimized scan insertion. Constraint-optimized scan insertion must include these cells in the final scan architecture.

- Scan

The design contains routed scan chains. Constraint-optimized scan insertion must include these chains in the final scan architecture.

Because the focus of this chapter is the scan replacement process, this discussion assumes that

- The input to constraint-optimized scan insertion is an optimized nonscan gate-level design
- The output from constraint-optimized scan insertion is an optimized design that contains unrouted scan cells. Note that constraint-optimized scan insertion performs scan replacement only.)

Note:

When you do not route the scan chains, the optimization performed during constraint-optimized scan insertion accounts for the timing impact of the scan cell, but it does not take into account the additional loading due to the scan chain routing.

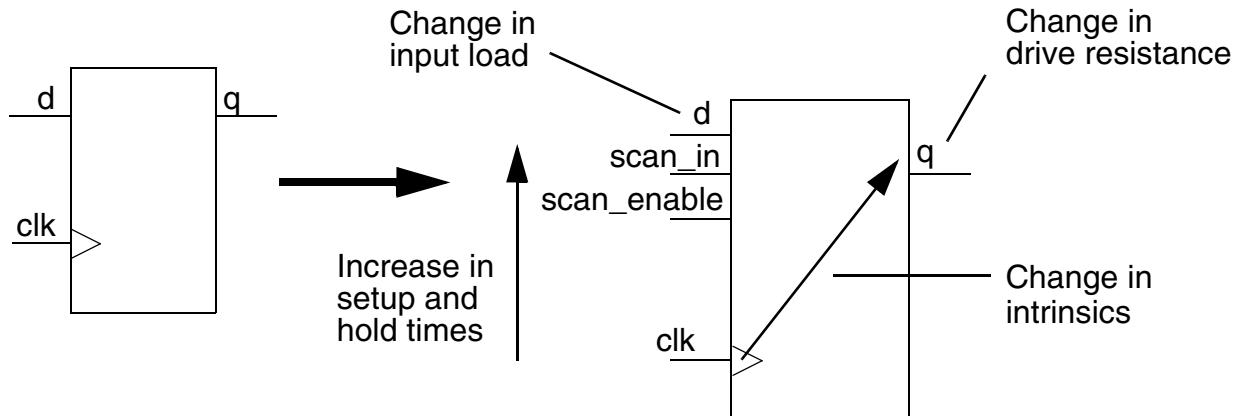
## Locating Scan Equivalents

To perform scan replacement, constraint-optimized scan insertion first locates simple scan equivalents by using the identical-function method. If the `insert_test_map_effort_enabled` variable is true and if constraint-optimized scan insertion does not achieve scan replacement using this method, it then uses sequential-mapping-based scan replacement. See the scan insertion information in the *Design Compiler Reference Manual: Optimization and Timing Analysis* for details about scan replacement methods.

Like test-ready compile, constraint-optimized scan insertion supports degeneration of scan cells to create the required scan equivalent functionality.

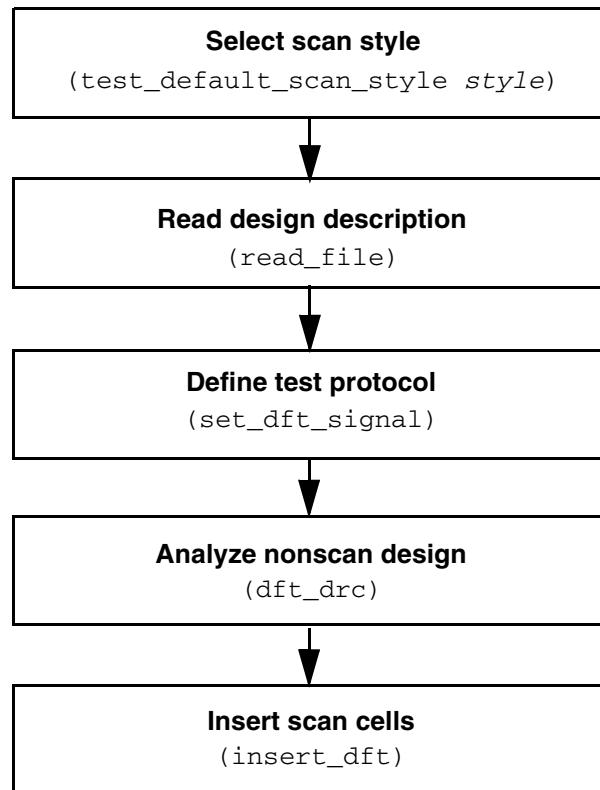
Replacing sequential cells with their scan equivalents modifies the design timing, as shown in [Figure 4-9](#). DFT Compiler performs scan-specific optimizations to reduce the timing impact of scan replacement. By using focused techniques, constraint-optimized scan insertion optimizes the scan logic faster than the incremental compile process could.

*Figure 4-9 Timing Changes Due to Scan Replacement*



[Figure 4-10](#) shows the flow used to insert scan cells with constraint-optimized scan insertion and the commands required to complete this flow.

*Figure 4-10 Constraint-Optimized Scan Insertion Flow  
(Scan Replacement Only)*



---

## Preparing for Constraint-Optimized Scan Insertion

Before performing constraint-optimized scan insertion,

- Verify the timing characteristics of the design.

Constraint-optimized scan insertion results in a violation-free design when the design has the following timing characteristics:

- The nonscan design does not have constraint violations.
- The time budget is good.
- You have properly applied realistic path constraints.
- You have described the clock skew.

Note:

If your design enters constraint-optimized scan insertion with violations, long runtimes can occur.

- Select a scan style.
- Identify barriers to scan replacement.

---

## Scan Insertion

To alter a default scan design, you must specify changes to the scan configuration. You can make specifications at any point before scan synthesis. This section describes the specification commands you can use.

With the DFT Compiler scan insertion capability, you can

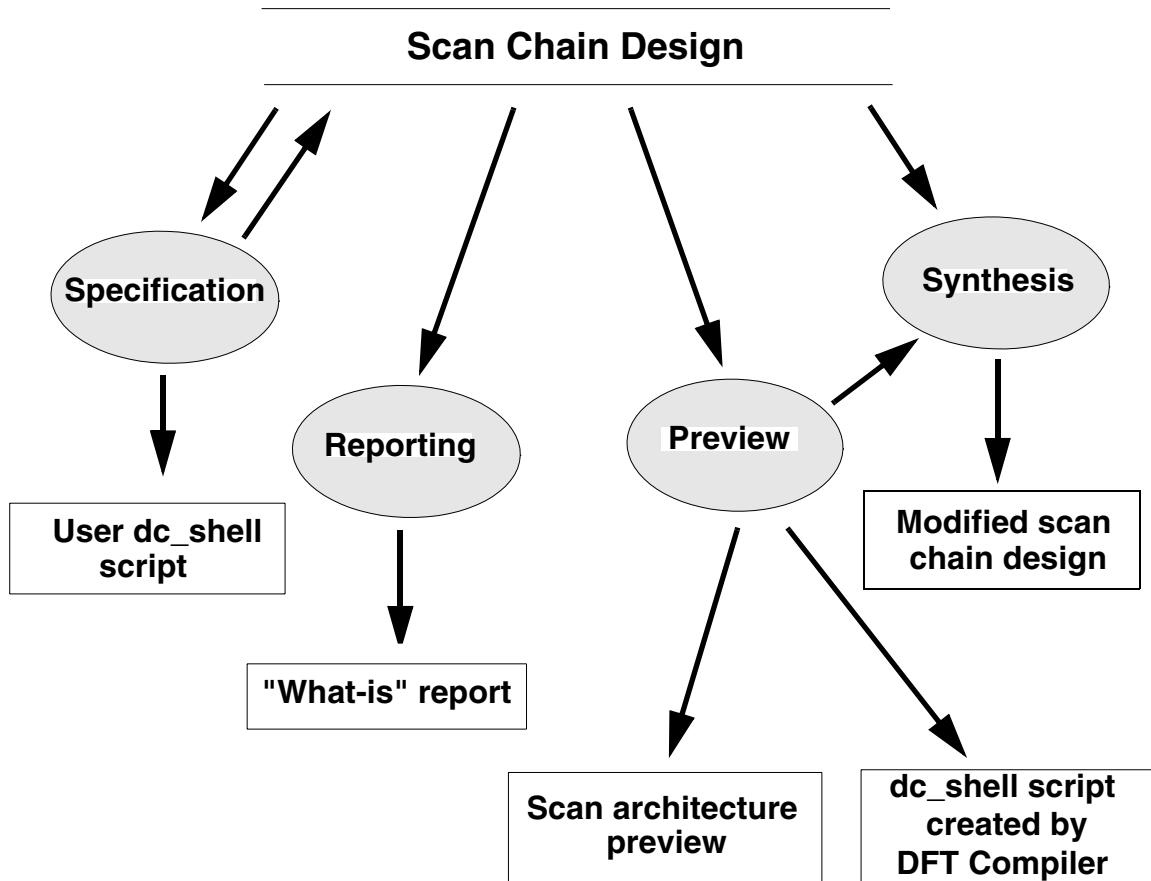
- Implement automatically balanced scan chains
- Specify complete scan chains
- Generate scan chains that enter and exit a design module multiple times
- Automatically fix certain scan rule violations
- Reuse existing modules that already contain scan chains
- Control the routing order of scan chains in a hierarchy
- Perform scan insertion from the top or the bottom of the design
- Implement automatically enabling or disabling logic for bidirectional ports and internal three-state logic

- Share functional ports as test data ports. DFT Compiler inserts enabling and disabling logic, as well as multiplexing logic, as necessary

You can design scan chains by using a specify-preview-synthesize process, which consists of multiple specification and preview iterations to define an acceptable scan design. After the scan design is acceptable, you can invoke the synthesis process to insert scan chains.

[Figure 4-11](#) shows this specify-preview-synthesize process.

*Figure 4-11 The Scan Insertion Process*



[Example 4-5](#) is a basic scan insertion script.

#### *Example 4-5 Basic Scan Insertion Script*

```
current_design Top
dft_drc
set_dft_configuration -fix_clock enable -fix_set enable \
    -fix_reset enable
set_scan_configuration -chain_count
create_test_protocol -infer_clock -infer_async
preview_dft
insert_dft
dft_drc -coverage_estimate
report_scan_path -v exist -chain all
report_constraints -all_violators
```

In this example, the following DFT configuration command enables AutoFix. For more information, see “[Using AutoFix](#)” on page 7-14.

```
set_dft_configuration -fix_clock enable -fix_reset enable \
    -fix_set enable
```

The scan specification command is `set_scan_configuration -chain_count 1`. It specifies a single scan chain in the design.

The `preview_dft` command is the preview command. It builds the scan chain and produces a range of reports on the proposed scan architecture.

The `insert_dft` command is the synthesis command. It implements the proposed scan architecture.

The following sections describe these steps in the design process.

## Specification Phase

During the specification phase, you use the scan and DFT specification commands to describe how the `insert_dft` command should configure the scan chains and the design. You can apply the commands interactively from the `dc_shell` or use them within design scripts. The specification commands annotate the database but do not otherwise change the design. They do not cause any logic to be created or any scan routing to be inserted.

The specification commands apply only to the current design and to lower-level subdesigns within the current design.

If you want to do hierarchical scan insertion by using a bottom-up approach, use the following general procedure:

1. Set the current design to a lower-level subdesign (`current_design` command).

2. Set the scan specifications for the subdesign (`set_scan_path`, `set_scan_element`, and so on).
3. Insert the scan cells and scan chains into the subdesign (`dft_drc`, `preview_dft`, and `insert_dft`).
4. Repeat steps 1, 2, and 3 for each subdesign, at each level of hierarchy, until you finish scan insertion for the whole design.

By default, the `insert_dft` command recognizes and keeps scan chains already inserted into subdesigns at lower levels. Thus, you can use different sets of scan specifications for different parts or levels of the design by using `insert_dft` separately on each part or level.

Note that each time you use the `current_design` command, any previous scan specifications no longer apply. This means that you need to enter new scan specifications for each newly selected design.

## **Scan Specification**

Using the scan specification commands, you can specify as little or as much scan detail as you want. If you choose not to specify any scan detail, the `insert_dft` command implements the default full-scan methodology. If you choose to completely specify the scan design that you require, you explicitly assign every scan element to a specific position in a specific scan chain. You can also explicitly define the pins to use as scan control and data pins.

Alternatively, you can create a partial specification, where you define some elements but do not issue a complete specification. If you issue a partial specification, the `preview_dft` command creates a complete specification during the preview process.

The scan specification commands are

- `set_scan_configuration`
- `set_scan_path`
- `set_dft_signal`
- `set_scan_element`
- `reset_scan_configuration`

These commands are described in detail later in this section.

## **DFT Configuration**

The DFT configuration commands are as follows:

- `reset_dft_configuration`

- `set_autofix_configuration`
- `set_autofix_element`
- `set_dft_configuration`
- `set_dft_signal`

## Preview

The `preview_dft` command produces a scan chain design that satisfies scan specifications on the current design and displays the scan chain design for you to preview. If you do not like the proposed implementation, you can iteratively adjust the specification and rerun preview until you are satisfied with the proposed design.

The scan preview process (the `preview_dft` command) performs two tasks:

- It checks the specification for consistency. For example, you cannot assign the same scan element to two different chains.
- It creates a complete specification if you have specified only a partial specification.

The DFT preview process (the `preview_dft` command) performs the above two tasks plus two more:

- It runs AutoFix.
- It produces a list of test points that are to be inserted into the design, based on currently enabled utilities.

When you use either of the preview commands, you create a `dc_shell` script that completely specifies the proposed implementation. You can edit this script and use the edited script as an alternative means of iterating to a scan design that meets your requirements.

**Limitation:**

Neither preview command annotates the database. If you want to annotate the database with the completed specification, use the `-script` option to create a specification `dc_shell` script and then run this script. The specification commands in this script add attributes to the database.

## Synthesis

You invoke the synthesis process by using the `insert_dft` command, which implements the scan design determined by the preview process. If you issue this command without explicitly invoking the preview process, the `insert_dft` command transparently runs `preview_dft`.

Execute the `dft_drc` command at least once before executing `insert_dft`. Executing `dft_drc` provides information on circuit testability before inserting scan into your design.



# 5

## Pre-Scan Test Design Rule Checking

---

This chapter describes the process for preparing for and running test design rule checking (DRC), and checking violations prior to scan insertion.

This chapter includes the following sections:

- [Test DRC Basics](#)
- [Classifying Sequential Cells](#)
- [Checking for Modeling Violations](#)
- [Setting Timing Attributes](#)
- [Creating Test Protocols](#)
- [Masking DRC Violations](#)

---

## Test DRC Basics

This section discusses the test DRC flow, the types of messages generated as a result of running the process, and the effects of violations on scan replacement.

---

### Test DRC Flow

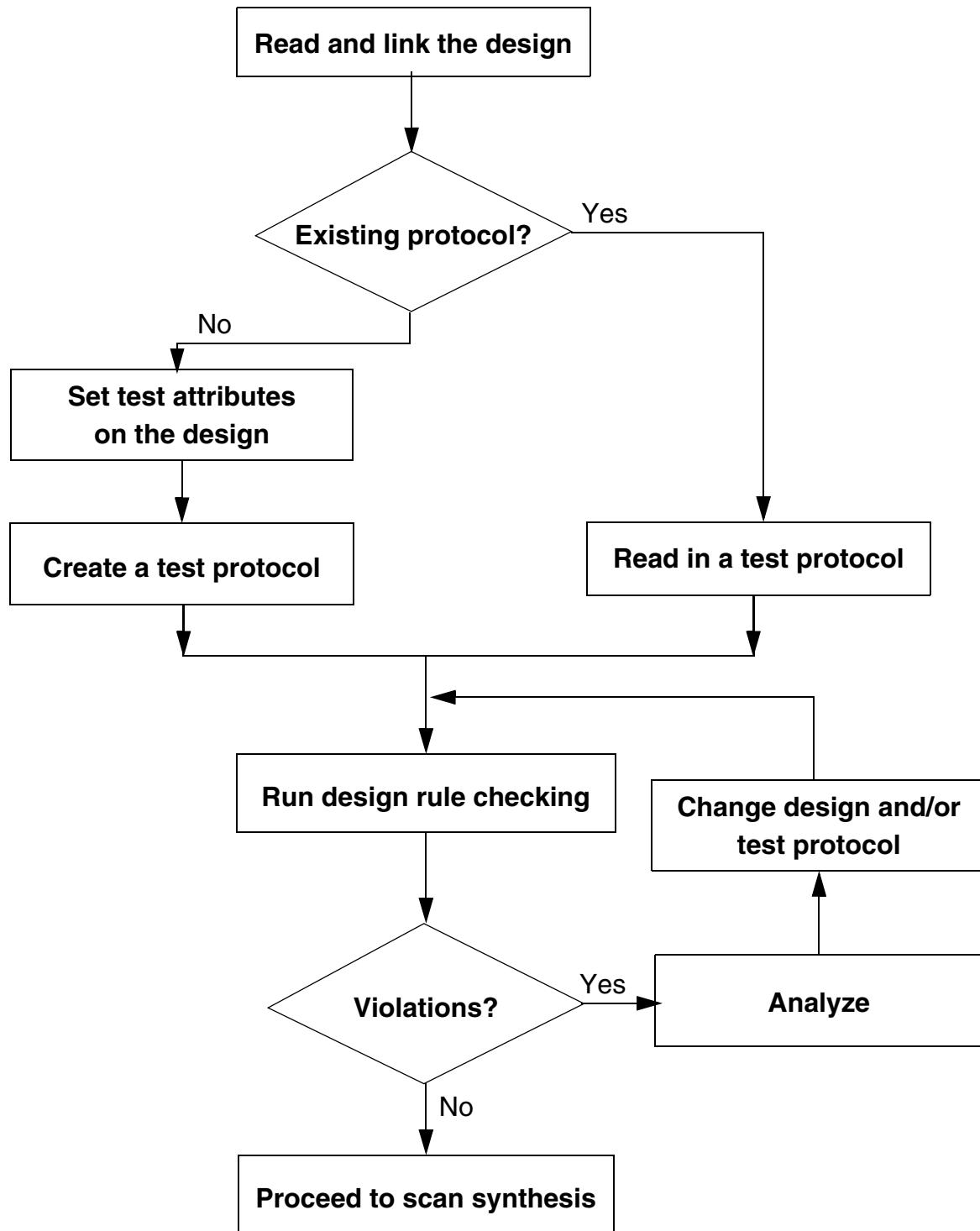
You use the `dft_drc` command to activate test design rule checking. However, before running this process, you must first create a test protocol that includes timing attributes (see “[Creating the Test Protocol](#)” on page 5-5 for information on creating test protocols).

After running the `dft_drc` command, violation messages are reported in three categories:

- Information messages – no action is required.
- Warnings – you should analyze the violations; however, you can still run certain DFT commands.
- Error messages – these indicate fatal errors that you need to correct before you can use the DFT commands.

[Figure 5-1](#) illustrates a general test design rule checking flow.

*Figure 5-1 Test DRC Flow*



The following steps outline the test DRC process:

1. Read and link the design into DFT Compiler.

For details, see the [“Preparing Your Design” on page 5-4](#).

2. Determine if you have an existing test protocol for the design.

If so, read the test protocol into DFT Compiler.

If not, do the following:

- Set the appropriate test attributes on the design.
- Create the test protocol.

3. Run design rule checking.

- If test DRC reports no violations, you can insert DFT structures into your design.
- If test DRC reports violations, you can graphically analyze the violations by using the DFT GUI, as described in Chapter 3, “Running the Test DRC Debugger.” (Note that the DFT GUI runs in the Design Vision environment.)

To fix the violations, either change your design, change your test protocol, or do both.

## Preparing Your Design

To prepare your design for test DRC, follow these steps:

1. Set the `search_path` variable to point to directory paths that contain your design and library files.
2. Set the `link_library` variable to point to the technology library files referred to by your design.
3. Set the `target_library` variable to point to technology library files you want mapped to your design.
4. Use the `read_file` command to read your design into DFT Compiler.
5. Run the `link` command to link your design with your technology library.

See the Design Compiler documentation for more information on how to read and link your design.

## **Creating the Test Protocol**

If you have an existing test protocol, read the test protocol into DFT Compiler by using the `read_test_protocol` command. If you do not have an existing test protocol, create it by following these steps:

1. Identify the test-related ports in your design. Such signals include
  - Clocks
  - Asynchronous sets and resets
  - Scan inputs
  - Scan outputs
  - Scan enables
2. Set the test attributes on these ports.
3. Run the `create_test_protocol` command to create the test protocol for your design.

## **Assigning a Known Logic State**

You can use the `set_test_assume` command to assign a known logic state to output pins of black box sequential cells. The command syntax is

```
set_test_assume value pin_list
```

The `value` argument specifies the assumed value, 1 or 0, on this output pin or pins.

The `pin_list` argument specifies the names of output pins of unknown black-box cells, including nonscan sequential cells in full-scan designs. The hierarchical path to the pin should be specified for pins in subblocks of the current design.

The `dft_drc` command takes into account the conditions you define with the `set_test_assume` command.

## **Performing Test Design Rule Checking**

After you create or read in a test protocol, perform test design rule checking by running the `dft_drc` command.

By default, `dft_drc` reports only the first instance of any violation that exists in the design. If you want to see all instances of the same violation, use the `dft_drc -verbose` command to report all instances of all the violations.

## Analyzing and Debugging Violations

You can graphically analyze the cause of a violation by using the DFT GUI, as described in Chapter 3, “Running the Test DRC Debugger.” (Note that the DFT GUI runs in the Design Vision environment.)

After you have located the cause of the violation, you can either change the design, change the test protocol, or do both. Then rerun the steps outlined above to see if the violations have been fixed.

You can also use AutoFix to fix uncontrollable clocks and asynchronous sets and resets. For more information on AutoFix, see [“Using AutoFix” on page 7-14](#).

## Summary of Violations

At the completion of design rule checking, the `dft_drc` command displays a violation summary. [Example 5-1](#) shows the format of the violation summary.

### *Example 5-1 Violation Summary*

```
-----  
DRC Report  
Total violations: 6  
-----  
6 PRE-DFT VIOLATIONS  
 3 Uncontrollable clock input of flip-flop violations (D1)  
 3 DFF set/reset line not controlled violations (D3)  
  
Warning: Violations occurred during test design rule  
checking. (TEST-124)  
-----  
Sequential Cell Report  
 3 out of 5 sequential cells have violations  
-----  
SEQUENTIAL CELLS WITH VIOLATIONS  
 * 3 cells have test design rule violations  
SEQUENTIAL CELLS WITHOUT VIOLATIONS  
 * 2 cells are valid scan cells
```

The total number of violations for the circuit appears in the header. If there are no violations in the circuit, the `dft_drc` command displays only the violation summary header. Within the summary, violations are organized by category. A violation category appears in the summary only if there are violations in that category. For each category, the `dft_drc` command displays the number (n) of violations, along with a short description of each violation and the corresponding error message number. Using the error message number, you can find the violation in the `dft_drc` run.

Unknown cell violations have message numbers in the TEST-451-to -TEST-456 range. Unsupported cell violations have message numbers in the TEST-464-to-TEST-469 range. The following is an excerpt from a violation summary for unsupported and unknown cells:

```
DRC Report
```

```
Total violations: 4
```

---

```
-----
```

### 3 MODELING VIOLATIONS

```
1 Cell has unknown model violation (TEST-451)
```

## Enhanced Reporting Capability

You can choose the enhanced reporting feature with the command  
set test\_disable\_enhanced\_dft\_drc\_reporting false. (The default value of  
the variable is true.) When enhanced reporting is enabled, the reporting and formatting of  
rule violations are changed to provide a better understanding of the respective rules.

A typical enhanced report can look like the following:

### *Example 5-2 Enhanced DRC Report Example*

```
In mode: all_dft...
Pre-DFT DRC enabled
Information: Starting test design rule checking. (TEST-222)
    Loading test protocol
    ...basic checks...
    ...basic sequential cell checks...
        ...checking for scan equivalents...
    ...checking vector rules...
    ...checking pre-dft rules...
Simulation library files used for DRC
-----
./core_slow_lvds_pads.v
./core_slow_special_cells.v
Cores and modes used for DRC in mode: all_dft
-----
SUB_1: U1, U3, U4 mode: Internal_scan
SUB_2: U5, U6 mode: Internal_scan
Modeling and user constraints that will prevent scan insertion
-----
Warning: Cell U34 will not be scanned due to set_scan_element command. (TEST-202)
DRC violations which will prevent scan insertion
-----
Warning: Cell U1 has constant 1 value. (TEST-505)
Warning: Reset input RN of DFF U53 was not controlled. (D3-1)
Information: There are 10 other cells with the same violation. (TEST-171)
DRC Violations which can affect ATPG coverage
-----
Warning: Clock CCLK can capture new data on LS input of DFF U25. (D13-1)
    Source of violation: input CLK of DLAT U13/clk_gate_flop/latch.
Warning: CCLK clock path affected by new capture on LS input of DFF U17 (D15-1)
    Source of violation: input CLK of DLAT U18/clk_gate_flop/latch.
-----
DRC Report
Total violations: 14
```

```

-----
1 MODELING AND USER VIOLATIONS AFFECTING SCAN INSERTION
  1 cell with set_scan_element constraint (TEST-202)
11 DRC VIOLATIONS AFFECTING SCAN INSERTION
  1 Constant cell (TEST-505)
11 DFF reset line not controlled violations (D3)
2 DRC VIOLATIONS AFFECTING ATPG coverage
  1 Data path affected by clock captured by clock in level sensitive
    clock_port violations (D13)
  1 Clock path affected by clock captured by clock in level sensitive
    clock_port violations (D15)
-----
Sequential Cell Report
      Cells   Core  core_cells
-----
Sequential elements detected:  50     5     50
Clock gating cells:          0
Synchronizing cells:         0
Non scan elements:           1     0     0
Excluded scan elements:      0     0     0
Violated scan elements:      11    1     10
Scan elements:                39    4     40
-----
```

---

## Test Design Rule Checking Messages

When you invoke `dft_drc`, it generates messages to assist you in determining problems with your scan design. These messages fall into three categories:

- Information

Information messages give you the status of the design rule checker or more detail about a particular rule violation.

- Warning

A warning message indicates a testability problem that lowers the fault coverage of the design. Most of the violations reported by the `dft_drc` command are warning messages. The warnings allow you to evaluate the effect of violations and determine acceptable violations, based on your test requirements.

Many warnings reported by `dft_drc` reduce fault coverage. Try to correct all violations, because a cell that violates a design rule, as well as the cells in its neighborhood, is not testable. A cell's neighborhood can be as large as its transitive fanin and its transitive fanout.

- Error

An error message indicates a serious problem that prevents further processing of the design in DFT Compiler until you resolve the problem.

## Test Design Rule Checking Message Generation

By default, the `dft_drc` command generates a message only for the first violation of a given type. To see all violations, use the `dft_drc -verbose` command.

The `dft_drc` command reports only the first instance of a given violation type.

## Understanding Test Design Rule Checking Messages

You can access online help for most warning messages generated by `dft_drc`. Online help provides information about the violation and information about how to proceed. Use the `help` command to access online help:

```
dc_shell> man message_id
```

Replace the `message_id` argument with the string shown in the parentheses that follow the warning text.

To keep a record of the information, warning, and error messages for your design, direct the output from the `dft_drc` command to a file with a command such as

```
dc_shell> dft_drc > my_drc.out
```

In this example, `my_drc.out` is the name of the output file.

---

## Effects of Violations on Scan Replacement

For designs that are synthesized with the `compile -scan` command, the default behavior is for violations on scan-replaced cells to cause the `insert_dft` command to unscan scan elements.

For designs that are not synthesized with the `compile -scan` command, violations on sequential cells cause the `insert_dft` command not to perform scan replacement.

If you have previously run the `dft_drc` command, the `insert_dft` command uses the results. In the AutoFix flow, the `insert_dft` command does not use these results and it runs `dft_drc` again.

The `insert_dft` command issues the following message:

```
Using test design rule information from previous dft_drc run
```

If you have not explicitly run the `dft_drc` command, the `insert_dft` command runs `dft_drc` as a preprocessor to determine which sequential elements can be included in scan chains. The `insert_dft` command issues the following message:

```
Information: Starting test design rule checking. (TEST-222)
```

In both cases, when violations occur, the `insert_dft` command issues the following message:

```
Warning: Violations occurred during test design rule  
checking. (TEST-124)
```

You should determine the causes of these violations before proceeding. Sequential cells with violations are not included in a scan chain because they would probably prevent the scan chain from working as intended.

---

## Viewing the Sequential Cell Summary

At the end of the `dft_drc` output, DFT Compiler provides a summary of the test status of the sequential cells in your design. [Example 5-3](#) shows the syntax of the sequential cell summary generated by DFT Compiler.

### *Example 5-3 Sequential Cell Summary*

---

```
-----  
Sequential Cell Report  
-----  
2 out of 133721 sequential cells have violations  
-----  
SEQUENTIAL CELLS WITH VIOLATIONS  
* 2 cells have capture violations  
SEQUENTIAL CELLS WITHOUT VIOLATIONS  
*133719 cells are valid scan cells
```

To get a complete listing of all the cells in each category, run the `dft_drc -verbose` command.

For information about classifying sequential cells, see the next section.

---

## Classifying Sequential Cells

After the violation summary, the `dft_drc` command displays a summary of sequential cell information.

[Example 5-4](#) shows the syntax of the sequential cell summary.

#### *Example 5-4 Sequential Cell Summary*

---

```
Sequential Cell Report  
2 out of 133721 sequential cells have violations  
  
-----  
SEQUENTIAL CELLS WITH VIOLATIONS  
* 2 cells have capture violations  
SEQUENTIAL CELLS WITHOUT VIOLATIONS  
*133719 cells are valid scan cells
```

The number of sequential cells with violations appears in the header. This number is the sum of the cells with scan shift violations, capture violations, and constant values, along with the cells that are black boxes. If a design has no sequential cells, only a header with the following message appears:

There are no sequential cells in this design

Within the summary, the sequential cells are divided into two groups: those with violations and those without. Only the categories of sequential cells that are found in the design are listed in the summary. In verbose mode, cell names are listed within each category. More information about the sequential cell categories is provided in the following sections.

---

## **Sequential Cells With Violations**

This section of the sequential cell summary points to problematic sequential cells. The cells in this group have corresponding violations that can be found in the `dft_drc` output.

### **Cells With Scan Shift Violations**

This category includes cells with scan-in and scan connectivity violations. Within this category, cells are listed by the type of scan shift violation.

- Not scan-controllable

The `dft_drc` command cannot transport data from a scan-in port into the cell.

- Not scan-observable

The `dft_drc` command cannot transport data from the cell to a scan-out port.

Note:

Cells in multibit components are homogeneous. If a cell in a multibit component has violations, all of the cells in that multibit component have violations.

Once `dft_drc` has run, you can invoke the `report_scan_path -view existing_dft -chain all` command to observe the scan chains as extracted by the `dft_drc` command.

## Black-Box Cells

Included in the black-box cells category are sequential cells that cannot be used for scan shift. Unknown cells and unsupported cells are classified as black boxes. These cells are not scan-replaced when you run the `insert_dft` command.

## Constant Value Cells

The constant value category includes sequential cells that are constant during scan testing. These cells are assumed to hold constant values; they are not scan-replaced by `insert_dft`. For every constant value sequential cell, there is a corresponding TEST-504 or TEST-505 violation.

---

## Sequential Cells Without Violations

The valid scan cells category displays the number of sequential cells that have no test design rule violations. ATPG tools can use these cells for scan shift and for measuring circuit response data. Valid scan cells can be scan-replaced by `insert_dft`.

Note:

Valid scan cells can have capture violations. Valid cells with capture violations only are scan-replaced.

The number of synchronization latches is listed in the last category.

---

## Checking for Modeling Violations

If you instantiate a cell that DFT Compiler doesn't understand, you can get modeling violations. The `dft_drc` command performs modeling checks locally, one cell at a time.

Modeling violations are discussed in the following subsections:

- [Black-Box Cells](#)
- [Unsupported Cells](#)
- [Generic Cells](#)
- [Scan Cell Equivalents](#)
- [Latches](#)

---

## Black-Box Cells

A cell whose output is considered unknown is classified as a black-box cell. These cells might lack a functional description in the technology library. Such cells marked as black box in the `report_lib` command. Also, the `dft_drc` command identifies black-box sequential cells.

The `dft_drc` command requires that you have a functional model in your library for each leaf cell in your design. If you use cells that do not have functional models, the `dft_drc` command displays the following warning:

```
Cell %s (%s) is unknown (black box) because functionality  
for output pin %s is bad or incomplete (TEST-451)
```

See the Library Compiler reference manuals for more information on modeling the behavior of cells.

## Correcting Black Box Cells

DFT Compiler models a cell as a black box in these cases:

- The `link` command cannot resolve the cell reference by using the technology libraries or designs in the `search_path` (unresolved reference).
- The technology library model for the cell reference does not contain a functional description (black box library cell).

In the following cases, a black box cell can have a severe impact on fault coverage:

- The black box cells are pad cells.

The `dft_drc` command completely fails and prevents `insert_dft` from working. This occurs during scan stitching at the top level.

- A black box cell controls the enable signal of an internal three-state driver or a bidirectional signal.

The `insert_dft` command inserts three-state and bidirectional control logic if the existing control logic is a black box, even if doing so is unnecessary.

DFT Compiler generates this message when it models a cell as a black box:

```
Warning: Cell %s (%s) is unknown (black box) because  
functionality for output pin %s is bad or incomplete.  
(TEST-451)
```

The method for correcting the violation depends on the source of the violation and the complexity of the cell.

Note:

Use the `link` command to correct unresolved references.

### Black Box Library Cell

If no functional description of the cell exists in the technology library, you need to obtain either a functional model or a structural model of the cell.

If the cell can be functionally modeled by Library Compiler, obtain an updated technology library that includes a functional model of the cell.

If you have a simulation model for the black box, declare it by using the following variable:

```
% set test_simulation_library simulation_library_path
```

- If you do not have a Library Compiler license or library source code, ask your semiconductor vendor for a library that contains a functional model of the cell.
- If you have a Library Compiler license and the library source code, add the functional description to the library cell model.

See the Library Compiler documentation for information about cell modeling.

If Library Compiler syntax does not support functional modeling of the cell, create a structural model for the cell and link the design to this structural model instead of the library cell model.

Note:

You should only use the `test_simulation_library` variable to replace leaf cells that do not have functional models. Do not use the variable to replace any arbitrary module in the design. If you want to replace the entire design module that consists of leaf cells, you should use the `remove_design` command to remove the module and then read the Verilog netlist description of that module into memory.

---

## Unsupported Cells

Cells can have a functional description and still not be supported by the `dft_drc` command. Using state table models, library developers can describe cells that violate the current assumptions for test rule checking. The `dft_drc` command detects those cells and flags them as black boxes.

DFT Compiler supports single-bit cells or multibit cells that have identical functionality on each pin; these cells have the following characteristics:

- The functional view, which Design Compiler understands and manipulates, is either a flip-flop, a latch, or a master-slave cell with `clocked_on` and `clocked_on_also` attributes.
- The test view, used for scan shifting, is either a flip-flop or a master-slave cell.
- The functional view and the test view each have a single clock per internal state.

The multibit library cell interfaces must be either fully parallel or fully global. For cells that do not meet these criteria, DFT Compiler uses single-bit cells.

For example, if you want to infer a 4-bit banked flip-flop with an asynchronous clear signal, the clear signal must be either different for each bit or shared among all 4 bits. If the first and second bits share one asynchronous reset but the third and fourth bits share another reset, DFT Compiler does not infer a multibit flip-flop. Instead, DFT Compiler uses 4 single-bit flip-flops. For more information about multibit cells and multibit components, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

DFT Compiler does not support registers or duplicate sequential logic within a cell. The nonscan equivalent of a scan cell must have only one state. A scan cell can have multiple states in shift mode.

If the `dft_drc` command detects such a cell, it issues the following warning:

```
Cell %s (%s) is not supported because it has too many
states (%d states). This cell is being black-boxed. (TEST-462)
```

When the `dft_drc` command recognizes part of a cell as a master-slave latch pair but finds extra states, it issues one of the following warnings, depending on the situation:

```
Master-slave cell %s (%s) is not supported because the state
pin %s is neither a master nor a slave. This cell is being
black-boxed, (TEST-463)
```

```
Master-slave cell %s (%s) is not supported because there
are two or more master states. This cell is being
black-boxed, (TEST-464)
```

```
Master-slave cell %s (%s) is not supported because there
are two or more slave states. This cell is being
black-boxed, (TEST-465)
```

If the `dft_drc` command detects a state with no clocks or with multiple clocks, it issues one of the following warnings:

Cell %s (%s) is not supported because the state pin %s has no clocks. This cell is being black-boxed, (TEST-466)

Cell %s (%s) is not supported because the state pin %s is multi-port. This cell is being black-boxed. (TEST-467)

In addition, the `dft_drc` command detects and rejects sequential cells with three-state outputs and issues the following warning:

Cell %s (%s) is not supported because it is a sequential cell with three-state outputs. This cell is being black-boxed, (TEST-468)

Black box cells have an adverse effect on fault coverage. To avoid this effect, you must replace unsupported cells with cells that DFT Compiler can support.

Note:

Unsupported cells can originate only from explicit instantiation. They are not used by Design Compiler or by DFT Compiler. For more information on modeling sequential cells, see the Library Compiler User Guide: *Methodology and Modeling Functionality in Technology Libraries* and Library Compiler User Guide: *Modeling Timing, Signal Integrity, and Power in Technology Libraries*.

---

## Generic Cells

Your design should be a mapped netlist. In the RTL stage, `dft_drc` will map your design into an internal representation

Some generic cells, such as unimplemented DesignWare parts and operators, have implicit functional descriptions. The `dft_drc` command treats them as black box cells and displays the following message:

Warning: Cell %s (%s) is unknown (black box) because functionality for output pin %s is bad or incomplete. (TEST-451)

If you instantiate generic cells after running `compile -scan`, you must recompile your design.

---

## Scan Cell Equivalents

When checking test design rules in a design without scan chains, the `dft_drc` command verifies that each sequential element has not been explicitly marked by using the `set_scan_element false` command. If a scan cell equivalent does not exist, the `dft_drc` command issues the following message:

```
Warning: No scan equivalent exists for cell %s (%s).  
(TEST-120)
```

Note:

Use the `set_scan_element false` command to prevent scan replacement.

The cells in violation are marked as nonscan. In the full-scan methodology, these cells are black boxes. If these cells are not valid nonscan, they are in violation and are black boxes. You can suppress the TEST-120 warning with the `set_scan_element` command. For example, to ensure that a nonscan latch cell is not made scannable, enter the command

```
dc_shell> set_scan_element false latch_name
```

If you use the `set_scan_element` command, the `dft_drc` command issues the following informational message:

```
Information: Cell %s (%s) will not be scanned due to a  
set_scan_element command. (TEST-202)
```

If the `dft_drc` command cannot find scan cell equivalents in the target library, the probable reason is that the target library does not contain test cells. In such cases, the `dft_drc` command issues the following warning:

```
Warning: Target library for design contains no scan-cell  
models. (TEST-224)
```

## Scan Cell Equivalents and the `dont_touch` Attribute

If you set the `dont_touch` attribute on a cell in your design before scan cell replacement, that cell is not modified or replaced when you optimize the design.

If you apply the `dont_touch` attribute to a scan cell after running `compile -scan` on your design, it will not be added to a scan chain.

If your design contains a sequential cell that has the `dont_touch` attribute assigned, the `dft_drc` command produces the following warning:

```
Warning:Cell %s (%s) can't be made scannable because it is  
dont_touched. (TEST-121)
```

Note:

Use the `dont_touch` attribute carefully, because it increases the number of nonscan cells and nonscan cells lower fault coverage.

You can use `set_scan_element false` if you do not want to make a sequential cell scannable but you do want to be able to modify the cell during optimization.

---

## Latches

DFT Compiler replaces latches with scannable latches whenever possible. If the `dft_drc` command cannot find scan cell equivalents for the latches, it marks the latches as nonscan and issues the TEST-120 warning as previously explained.

### Nonscan Latches

DFT Compiler models nonscan latches in two ways:

- As black boxes
- As synchronization elements

If you do not scan replace your latches, you can ignore “no-scan equivalent” messages for latches.

A nonscan latch is treated by default as a black box. However, if the latch satisfies the requirements for a synchronization element, the `dft_drc` command treats the latch as a synchronization element.

Note:

In `dft_drc`, synchronous elements can be on the scan chain.

---

## Setting Timing Attributes

This section discusses the process for setting timing attributes in your design. Timing attributes are used by the test protocol for design rule checking and for DFT preview and insertion.

This section has the following subsections:

- [Protocols for Common Design Timing Requirements](#)
- [Setting Timing Attributes](#)

---

## Protocols for Common Design Timing Requirements

Before creating a test protocol and checking test design rules, you need to identify the timing information for your design. You do this by setting a number of timing attributes and, if necessary, by defining test clock requirements. Timing attributes are discussed in detail in “[Setting Timing Attributes](#)” on page 5-20.

Defining test clock requirements is discussed in detail in Chapter 6, “[Architecting Your Test Design](#).”

If you intend to use postclock strobing, you need to change the default variable values. If your design’s timing attributes are the same as the variables’ default values, you do not need to make any changes.

### Strobe-Before-Clock Protocol

The timing requirements for a strobe-before-clock protocol are shown in the following example. Check with your semiconductor vendor for specific timing information.

```
test_default_period : 100.00 ;
test_default_delay : 0.00 ;
test_default_bidir_delay : 0.00 ;
test_default_strobe : 40.00 ;
test_default_strobe_width : 0.00;
```

### Strobe-After-Clock Protocol

To use a strobe-after-clock protocol, set the timing values described in the following example. Although the strobe-after-clock protocol works with TetraMAX ATPG, the strobe-before-clock protocol is more efficient. Use the timing attributes shown in this example.

```
test_default_period : 100.00 ;
test_default_delay : 5.00 ;
test_default_bidir_delay : 55.00 ;
test_default_strobe : 95.00 ;
test_default_strobe_width : 0.0;
```

If you intend to use a strobe-after-clock protocol with TetraMAX ATPG, use the timing attributes shown in the following example:

```
test_default_period : 100.00 ;
test_default_delay : 0.00 ;
test_default_bidir_delay : 0.00 ;
test_default_strobe : 90.00 ;
test_default_strobe_width : 0.00;
```

---

## Setting Timing Attributes

Before you run `create_test_protocol`, you need to define timing attributes. The command uses the following test variables to determine the values in the test protocol timing attributes:

```
test_default_period  
test_default_delay  
test_default_bidir_delay  
test_default_strobe  
test_default_strobe_width
```

Your semiconductor vendor's requirements, together with the basic scan test requirements, drive the specification of test timing parameters. If you intend to use postclock strobing, you need to change the default variable values. You can do this every time you create a new design, or you can add these variable values to your local `.synopsys_dc.setup` file.

### **test\_default\_period Attribute**

The `test_default_period` variable defines the default value, in ns, for the period in the test protocol. The period value must be a positive real number.

By default, DFT Compiler uses a 100-ns test period. If your semiconductor vendor uses a different test period, specify the required test period by using the `test_default_period` variable.

The syntax for setting the variable is

```
set test_default_period period
```

For example,

```
dc_shell> set test_default_period 100.0
```

In the `.synopsys_dc.setup` file, `test_default_period` is 100.0 ns.

### **test\_default\_delay Variable**

The `test_default_delay` variable defines the default value, in ns, for the input delay in the inferred test protocol. The delay value must be a nonnegative real number less than the strobe value. See the default timing in [Figure 5-2 on page 5-24](#).

By default, DFT Compiler applies data to all nonclock input ports 5.0 ns after the start of the cycle. If your semiconductor vendor requires different input timing, specify the required input delay by using the `test_default_delay` variable.

The syntax for setting the variable is

```
set test_default_delay delay
```

For example

```
dc_shell> set test_default_delay 5.0
```

In the .synopsys\_dc.setup file, test\_default\_delay is 5.0 0 ns.

## **test\_default\_bidir\_delay Attribute**

The test\_default\_bidir\_delay variable defines the default value, in ns, for the bidirectional delay in the inferred test protocol. The *bidir\_delay* must be a positive real number less than the strobe value and can be less than, greater than, or equal to the delay value. See the default timing in [Figure 5-2 on page 5-24](#).

By default, DFT Compiler applies data to all bidirectional ports in input mode 0 ns after the start of the parallel measure cycle. In any cycle where a bidirectional port changes from input mode to output mode, DFT Compiler releases data from the bidirectional port 0 ns after the start of the cycle. If your semiconductor vendor requires different bidirectional timing, specify the required bidirectional delay by using the test\_default\_bidir\_delay variable.

The risks associated with incorrect specification of the bidirectional delay time include

- Test design rule violations
- Bus contention
- Simulation mismatches

Minimize these risks by carefully specifying the bidirectional delay time.

DFT Compiler uses the bidirectional delay time as

- The data application time for bidirectional ports in input mode during the parallel measure cycle and during scan-in for bidirectional ports used as scan inputs or scan-enable signals
- The data release time for bidirectional ports in input mode during cycles in which the bidirectional port changes from input mode to output mode

DFT Compiler performs relative timing checks during test design rule checking. The following requirements must be met:

- The bidirectional delay time must be less than the strobe time.

If you change the strobe time from the default value, confirm that the bidirectional delay value meets this requirement.

- If the bidirectional port drives sequential logic, the bidirectional delay time must be equal to or greater than the active edge of the clock.

The syntax for setting the variable is

```
set test_default_bidir_delay bidir_delay
```

For example

```
dc_shell> set test_default_bidir_delay 40.0
```

In the .synopsys\_dc.setup file, test\_default\_bidir\_delay is 0 ns.

### **test\_default\_strobe Variable**

The test\_default\_strobe variable defines the default value, in ns, for the strobe in the inferred test protocol. The strobe value must be a positive real number less than the period value and greater than the test\_default\_delay value (see the default timing in [Figure 5-2 on page 5-24](#)).

By default, DFT Compiler compares data at all output ports 40 ns after the start of the cycle. If your semiconductor vendor requires different strobe timing, specify the strobe time by using the test\_default\_strobe variable.

The syntax for setting the variable is

```
set test_default_strobe strobe
```

For example:

```
dc_shell> set test_default_strobe 100.0
```

In the .synopsys\_dc.setup file, test\_default\_strobe is 40 ns.

### **test\_default\_strobe\_width Variable**

The test\_default\_strobe\_width variable defines the default value, in ns, for the strobe width in the inferred test protocol. The strobe width value must be a positive real number. The strobe value plus the strobe width value must be less than or equal to the period value. See the default timing in [Figure 5-2 on page 5-24](#).

Clocking requirements specified by semiconductor vendors include

- Clock waveform timing
- Maximum number of unique clock waveforms
- Minimum delay between different clock waveforms, which allows for clock skew on the tester

DFT Compiler provides the capability to specify clock waveform timing but does not place any restrictions on the number of unique waveforms that can be defined or the minimum time between clock waveforms. By determining what restrictions the semiconductor vendor places on these timing parameters, you can define clock waveforms that meet the restrictions.

When DFT Compiler infers clock ports during `dft_drc`, the clock type determines the default timing for each clock edge. [Table 5-1](#) provides the default clock timing for each clock type.

*Table 5-1 Default Clock Timing for Each Clock Type*

Clock type	First edge	Second edge
Edge-triggered or D-latch enable	45.0	55.0
Master clock	30.0	40.0
Slave clock	60.0	70.0
Edge-triggered	45.0	60.0
Master clock1	50.0	60.0
Slave clock	40.0	70.0

DFT Compiler determines the polarity of the first edge (rise or fall) so that the first clock edge triggers the majority of cells on a clock. The timing arcs in the technology library specify each cell's trigger polarity. The polarity of the second edge is opposite the polarity of the first edge, that is, if the first edge is rising (falling), the second edge is falling (rising).

Use the `set_dft_signal` command to specify clock waveforms if your semiconductor vendor's requirements differ from the default timing.

The `set_dft_signal` command has a time period associated with it. That period has to be identical to the `test_default_period` value. If you change the value of one, you must check the value of the other.

The syntax for setting the variable is

```
set test_default_strobe_width strobe_width
```

If you need a window strobe in your STIL procedure file (SPF) or STIL patterns, set the default value of `test_default_strobe_width` to 1.0, as shown in the following command:

```
dc_shell> set test_default_strobe_width 1.0
```

In the .synopsys\_dc.setup file, test\_default\_strobe\_width is 0.0 ns.

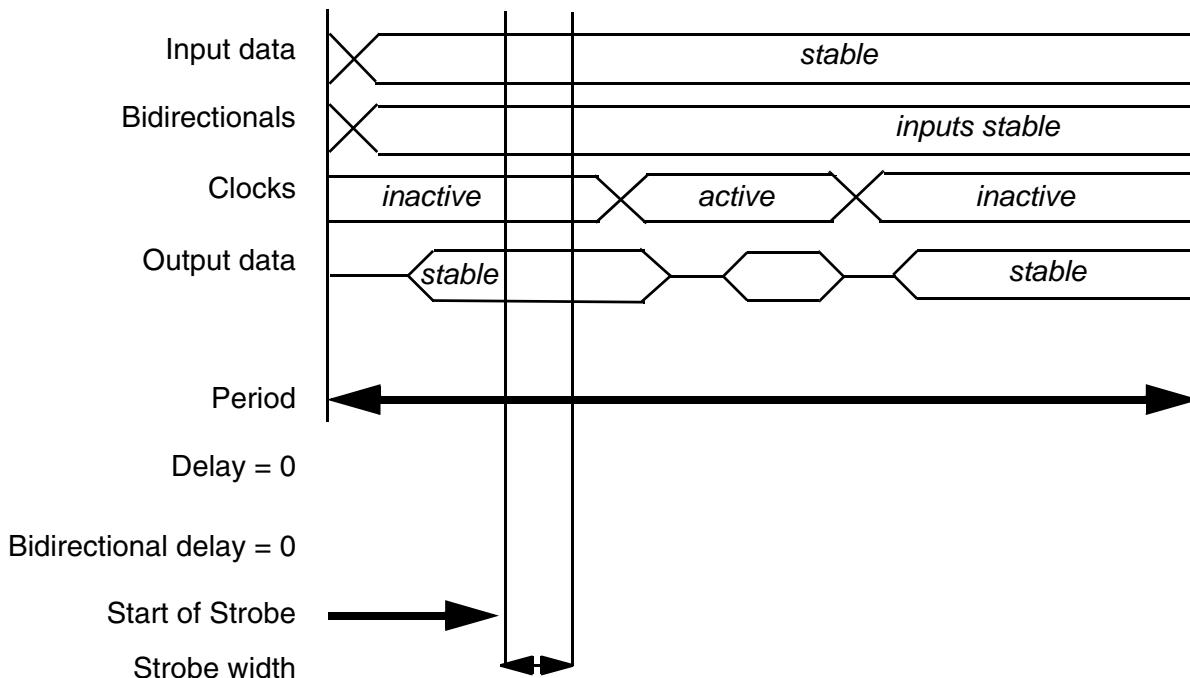
Note:

When test\_default\_strobe\_width is 0.0 ns, the strobe width is equal to one of two values: the difference between the strobe time and the end of the period, or the difference between the strobe time and the first input event after the strobe occurs, whichever occurs first.

## The Effect of Timing Attributes on Vector Formatting

[Figure 5-2](#) shows a timing diagram for a strobe-before-clock scheme.

*Figure 5-2 Effect of Timing Attributes on Vector Formatting*



---

## Creating Test Protocols

Test protocols are an intrinsic part of your design-for-test process and must be created before you run `dft_drc`. This section has the following subsections related to creating test protocols:

- [Design Characteristics for Test Protocols](#)
  - [STIL Test Protocol File Syntax](#)
  - [Defining an Initialization Protocol](#)
  - [Scan Shift and Parallel Cycles](#)
  - [Examining a Test Protocol File](#)
- 

### Design Characteristics for Test Protocols

A test protocol is based on certain characteristics of a design. How a protocol is affected by these characteristics is discussed in the following subsections:

- [scan\\_style Attribute](#)
- [signal\\_type Attributes](#)
- [Clock Ports](#)
- [Asynchronous Control Ports](#)
- [Bidirectional Ports](#)

#### **scan\_style Attribute**

Each scan style has a unique method of performing scan shift, which must be reflected in the test protocol. [“Scan Shift and Parallel Cycles” on page 5-32](#) describes how scan style influences the scan shift process.

#### **signal\_type Attributes**

The `signal_type` attributes for each test port are set automatically by `insert_dft` and preserved if you save the design in Synopsys .ddc format. If you have an existing scan design that is not saved in Synopsys .ddc format, you must identify each test port with the appropriate `signal_type` attribute. You can do this by using the `set_dft_signal` command.

## Clock Ports

You specify clock ports (and their timing attributes) by using the `set_dft_signal` command.

Tracing back from the clock pins on all sequential elements to the ports driving these pins interesting Clock ports can also be inferred by. The default timing for the clock signals is determined by the `set_scan_configuration -style` setting.

## Asynchronous Control Ports

You specify asynchronous control ports by using the following command:

```
dc_shell> set_dft_signal -view existing_dft \
    -type Reset -port Rst -active_state 1
```

Asynchronous control ports can also be inferred by tracing back from the asynchronous pins on all sequential elements to the ports controlling these pins. Asynchronous control ports must be identified because all asynchronous inputs must be disabled during scan shift to allow predictable loading and unloading of the scan data.

## Bidirectional Ports

In all cycles except parallel measure and capture, all nondegenerated bidirectional ports are assumed to be in output (driving) mode and are appropriately masked. During parallel measure and capture cycles, ATPG data controls the bidirectional ports as normal input or output ports but `test_default_bidir_delay` controls the timing.

---

## STIL Test Protocol File Syntax

DFT Compiler reads test protocols written in the Standard Test Interface Language (STIL). The STIL format is also used by TetraMAX ATPG.

Although the STIL procedure file syntax is the same as that used by TetraMAX ATPG, DFT Compiler cannot read some of the STIL elements that are available in TetraMAX ATPG.

The following STIL elements are *not* available in DFT Compiler:

- Post `load_unload` vectors
- Multiple scan groups in the `load_unload` procedure
- Multiple waveforms in the timing section

For general information on STIL standards (IEEE Std. 1450.0-1999), see the STIL home page at

<http://grouper.ieee.org/groups/1450/index.html>

Note that both DFT Compiler and TetraMAX use the IEEE P1450.1 extensions to STIL. For details, see Appendix E, “STIL IEEE P1450.1 Extensions,” in the *TetraMAX ATPG User Guide*.

## Defining the test\_setup Macro

The `test_setup` macro is optional. It defines any initialization sequences that the design might need for test mode or to ensure that the device is in a known state. A sample `test_setup` macro is shown in [Example 5-5](#).

*Example 5-5 Defining the test\_setup Macro in the SPF*

```
STIL;
    ScanStructures {
        ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
        ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
        ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
        ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
    }
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si=####; _so=####; CLOCK=P; }
        }
    }
}
MacroDefs {
    test_setup {
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}
```

If you need to initialize a port to X in the `test_setup` macro, the STIL assignment character for this is N. An X indicates that outputs are measured and the result is masked.

## Defining Basic Signal Timing

If you do not define the signal timing explicitly, DFT Compiler uses its own default values.

[Example 5-6](#) contains many additions to define signal timing. Line numbers have been added for reference.

- Lines 6–9. Define some additional signal groups so that timing for all inputs or outputs can be defined in just a few lines, instead of explicitly naming each port and its timing.

- Lines 12–28. This is a waveform table with a period of 1,000 ns that defines the timing to be used during nonshift cycles.
- Line 37. Addition of the W statement ensures that BROADSIDE\_TIMING is used for V cycles during the load\_unload procedure.
- Line 48. Causes the test\_setup macro to use BROADSIDE\_TIMING.

*Example 5-6 Defining Timing in the SPF*

```

1. STIL;
2. UserKeywords PinConstraints;
3. PinConstraints { "TEST_MODE" 1; "PLL_TEST_MODE" 1; }
4. SignalGroups {
5.     bidi_ports = '"D[0]" + "D[1]" + "D[2]" + "D[3]" + "D[4]" +
+ "D[5]" + "D[6]" + "D[7]" + "D[8]" + "D[9]" + "D[10]" +
"D[11]" + "D[12]" + "D[13]" + "D[14]" + "D[15]" ';
6.     input_grp1 = 'SCAN_ENABLE + BIDI_DISABLE + TEST_MODE +
PLL_TEST_MODE' ;
7.     input_grp2 = 'SDI1 + SDI2 + DIN + "IRQ[4]"' ;
8.     in_ports = 'input_grp1 + input_grp2';
9.     out_ports = 'SDO2 + D1 + YABX + XYZ';
10. }
11. Timing {
12.     WaveformTable "BROADSIDE_TIMING" {
13.         Period '1000ns';
14.         Waveforms {
15.             CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
// clock
16.             CLOCK { 01Z { '0ns' D/U/Z; } }
17.             RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
// async reset
18.             RESETB { 01Z { '0ns' D/U/Z; } }
19.             input_grp1 { 01Z { '0ns' D/U/Z; } }
20.             input_grp2 { 01Z { '10ns' D/U/Z; } }
// outputs are to be measured at t=350
21.             out_ports { HLT { '0ns' X; '350ns' H/L/T/X; } }
// bidirectional ports as inputs are forced at t=20
22.             bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
23.             // bidirectional ports as outputs are measured at
t=350
24.             bidi_ports { X { '0ns' X; } }
25.             bidi_ports { HLT { '0ns' X; '350ns' H/L/T; } }
26.         }
27.     } // end BROADSIDE_TIMING
28. }
29. ScanStructures {
30.     ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
31.     ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
32.     ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
33.     ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
34. } // end scan structures
35. Procedures {
36. "load_unload" {
37.     W "BROADSIDE_TIMING" ;
38.     V {CLOCK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1;
bidi_ports = \r16 Z;}

```

```

39.      V {}
40.      V { bidi_ports = \r4 1010 ; }
41.      Shift {
42.          V { _si=#####; _so=#####; CLOCK=P; }
43.      }
44. } // end load_unload
45. } //end procedures
46. MacroDef {
47.     "test_setup" {
48.         W "BROADSIDE_TIMING" ;
49.         V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
50.             BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZZ; }
51.             V {PLL_RESET = 0; }
52.             V {PLL_RESET = 1; }
53.     } // end test_setup
54. } //end procedures

```

## Defining the load\_unload Procedure

The `load_unload` procedure contains information about placing the scan chains into a shiftable state and shifting 1 bit through them. DFT Compiler creates this procedure if you define the scan-enable information before you write out the STIL file. [Example 5-7](#) shows the syntax used to define scan chains.

### *Example 5-7 Defining Scan Chain Loading and Unloading in the SPF*

```

STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE=1; }
    }
}

```

## Defining the Shift Procedure

The shift procedure specifies how to shift the scan chains within the definition of the `load_unload` procedure. The bold text shown in [Example 5-8](#) defines the shift procedure.

### *Example 5-8 Defining the Scan Chain Shift Procedure in the SPF*

```

STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {

```

```

    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si #####; _so #####; CLOCK=P; }
        }
    }
}

```

---

## Defining an Initialization Protocol

If your design requires an initialization sequence to configure it for scan testing, you can provide the initialization vectors through an initialization protocol. With an initialization protocol, you provide specific vectors to initialize the design while letting the `create_test_protocol` command complete the scan shifting steps of the protocol.

Use the following process to generate an initialization protocol:

1. Analyze the design to determine its test configuration requirements.
  - Determine the initial state required and the initialization sequence necessary to achieve this state.
  - Determine the test configuration required to maintain this initial condition throughout scan testing.
2. Generate a default test protocol file.
  - Specify timing parameters if you require values other than the default.
  - Specify test configuration requirements determined in the analysis step by using the `set_dft_signal` command.
  - Run `create_test_protocol` to generate the default protocol.
  - Use the `write_test_protocol` command to write the ASCII protocol file.
3. Create the initialization protocol file.

Modify the initialization sequence in the `test_setup` section of the test protocol file.

4. Read in the initialization protocol.

First remove the existing protocol by using the `remove_test_protocol` command, and then read the initialization protocol using the following command.

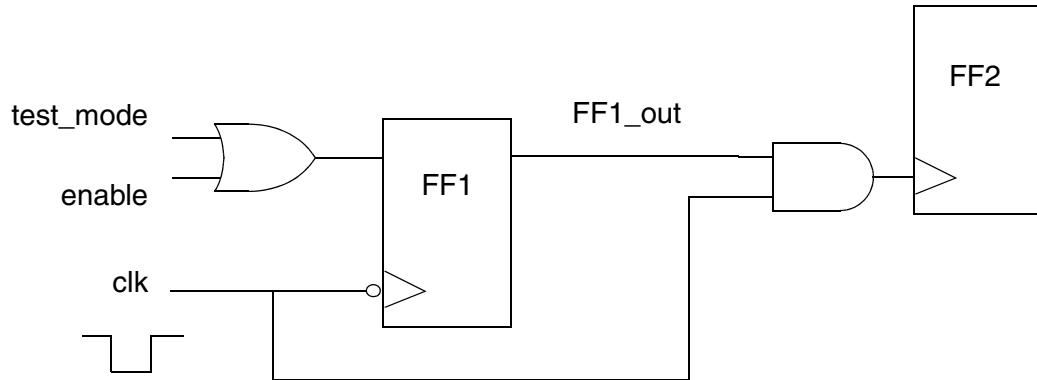
```
read_test_protocol -section test_setup
```

5. Rerun `create_test_protocol` to complete the test protocol.

Run test DRC.

See the design in Figure 5-3 for an illustration of the use of an initialization protocol.

*Figure 5-3 Design That Needs an Initialization Protocol*



In this design, the clock signal, clk, is active low. For the clock signal to reach FF2, you need to initialize it by pulsing clk once so that the enable signal FF1\_out is asserted. Because the `create_test_protocol` command has no knowledge of this requirement, you need to modify the generated protocol to include this special initialization sequence.

The initialization sequence generated by the `create_test_protocol` command looks like the following:

```
"test_setup" {
    W "_default_WFT_";
    V {   "CLK"=1; }
    V {   "CLK"=1; "test_mode"=1; }
}
```

If this initialization sequence has not been modified, test DRC gives the following violations:

4 PRE-DFT VIOLATIONS

- 3 Uncontrollable clock input of flip-flop violations (D1)
- 1 Clock not able to capture violation (D8)

The initialization sequence that is necessary to initialize the circuit is the following:

```
"test_setup" {
    W "_default_WFT_";
    V { "CLK"=1; }
    V { "CLK"=1; "test_mode"=1; }
    V { "CLK"=P; "test_mode"=1; }
    V { "CLK"=1; "test mode"=1; }
```

Test DRC requires that all clock signals are in their inactive state at the end of the initialization sequence. When this initialization sequence is applied, test DRC indicates that there are no test design rule violations.

However, after `insert_dft`, this initialization sequence is lost. You must reapply the same initialization sequence to ensure that post-scan insertion test DRC reports no violations.

The following table shows the flows you should use with various types of test protocols.

If you have	Use this flow
No test protocol	<code>set_dft_signal...</code> <code>create_test_protocol</code> <code>dft_drc</code>
Only the <code>test_setup</code> section in the protocol	<code>set_dft_signal...</code> <code>read_test_protocol -section test_setup</code> <code>create_test_protocol</code> <code>dft_drc</code>
Full protocol	<code>read_test_protocol (no -section test_setup)</code> <code>set_dft_signal (for clocks and asynchronous signals)</code> <code>dft_drc</code>

---

## Scan Shift and Parallel Cycles

Both the standard protocol and the strobe-before-clock protocol access scan chains in parallel. Scan output for the current pattern and scan input for the next pattern occur simultaneously.

The process DFT Compiler uses to perform scan shift is determined by the scan style you selected with the `set_scan_configuration -style` command or with the `test_default_scan_style` environment variable.

## Multiplexed Flip-Flop Scan Style

For the multiplexed flip-flop scan style, scan shift is performed by execution of the following steps  $n$  times, where  $n$  is the number of bits in the longest scan chain:

1. Assert the scan-enable signals.
2. Apply scan data at the scan input ports.
3. Pulse the system clocks.
4. Compare scan data at the scan output ports.

During the parallel measure and capture cycles, test design rule checking treats the scan-enable signal like any other parallel input; in some capture cycles, the captured data can be from the scan path rather than the functional path. Because fault detection occurs only during the parallel measure cycle and during comparison of captured data at the scan output ports, treating the scan-enable signal as a parallel input allows inclusion of scan logic and clock logic in the fault list and detection of faults on these nodes.

Note:

```
Define clocks by using the set_dft_signal -view exist \
-type ScanClock command.
```

## Clocked-Scan Scan Style

For the clocked-scan scan style, scan shift is performed by execution of the following steps  $n$  times, where  $n$  is the number of bits in the longest scan chain:

1. Apply scan data at the scan input ports.
2. Pulse the scan clock ports. (Scan clock ports are identified with the `test_scan_clock` attribute.)
3. Compare scan data at the scan output ports.

## LSSD Scan Style

For the LSSD scan style, scan shift is performed by execution of the following steps  $n$  times, where  $n$  is the number of bits in the longest scan chain:

1. Apply scan data at the scan input ports.
2. Pulse the test master clock, and then pulse the slave clock. (Test master clock ports are identified with the `test_scan_clock_a` attribute; test slave clock ports are identified with the `test_scan_clock_b` attribute.)
3. Compare scan data at the scan output ports.

For all scan styles, including the LSSD scan style, the parallel measure cycle is performed by application of data to nonclock input ports, holding clocks inactive, and comparing data at output ports. The capture cycle involves pulsing a clock. Nonclock input ports remain unchanged from the parallel measure cycle; output ports and bidirectional ports are masked.

Note:

Make sure the `ScanMasterClock`, `ScanSlaveClock`, `MasterClock`, and `SlaveClock` signal types are specified relatively.

---

## Examining a Test Protocol File

You can convert a test protocol file into an ASCII file that you can view and edit. To print this test protocol file to a file, use the `write_test_protocol` command. The command syntax is

```
write_test_protocol [-output test_protocol_file_name]
                    [-test_mode mode_name]
                    [-names verilog | verilog_single_bit]
```

Option	Description
<code>-out test_protocol_file_name</code>	Specifies the name of the ASCII output file. The default file name is <i>design_name.spf</i> , where <i>design_name</i> is the current design, and the <i>.spf</i> extension identifies the file type as a STIL format test protocol file.
<code>-test_mode <i>mode_name</i></code>	Specifies the CTL model test mode from which the protocol is generated.
<code>-names verilog   verilog_single_bit</code>	Specifies the form of the names used in the STIL protocol. Names can be unchanged from internal representation (the default). They can also be modified as Verilog names or as Verilog names compatible with the usage of the <code>verilogout_single_bit</code> environment variable. In all cases, the internal representation is not changed. This option takes effect only in conjunction with <code>-test_mode</code> options, when HSS is used. In all other cases, the form of the names is determined by the setting of the <code>test_stil_netlist_format</code> variable.

Note:

Do not use the `write_test_protocol` command before you run `create_test_protocol`. If you do, you will get an error message to the effect that no test protocol exists.

[Example 5-9](#) shows the test protocol file for a multiplexed flip-flop design. This file was generated by use of the `write_test_protocol` command after execution of test design rule checking on the design.

*Example 5-9 Test Protocol for Multiplexed Flip-Flop Design Example*

```
STIL 1.0 {
    Design P2000.9;
```

```

}

Header {
    Title DFT Compiler 2003.06 STIL output;
    Date Thu Apr 10 14:30:34 2003 ;
    History {
    }
}
Signals {
    CDN In; CLK In; DATA In; IN1 In; TEST_SE In;
TEST_SI In;
    OUT1 Out; OUT2 Out;
}
SignalGroups {
    all_inputs = `CDN + CLK + DATA + IN1 + TEST_SE +
TEST_SI'; // #signals=6
    all_outputs = `OUT1 + OUT2'; // #signals=2
    all_ports = `all_inputs + all_outputs'; // #signals=8
    _pi = `all_inputs'; // #signals=6
    _po = `all_outputs'; // #signals=2
}
Timing {
    WaveformTable _default_WFT_ {
        Period '100ns';
        Waveforms {
            all_inputs { 0 { '5ns' D; } }
            all_inputs { 1 { '5ns' U; } }
            all_inputs { Z { '5ns' Z; } }
            all_inputs { N { '5ns' N; } }
            all_outputs { X { '0ns' X; } }
            all_outputs { H { '0ns' X; '95ns' H; } }
            all_outputs { T { '0ns' X; '95ns' T; } }
            all_outputs { L { '0ns' X; '95ns' L; } }
            CLK { P { '0ns' D; '45ns' U; '55ns' D; } }
            CDN { P { '0ns' U; '45ns' D; '55ns' U; } }
        }
    }
}
PatternBurst __burst__ {
    PatList {
        __pattern__ {
        }
    }
}
PatternExec {
    PatternBurst __burst__;
}
Procedures {
    capture {
        W _default_WFT_;
        V { _pi =\r6 #; _po =\r2 #; }
    }
    capture_CLK {
        W _default_WFT_;
        forcePI : V { _pi =\r6 #; }
        measurePO : V { _po =\r2 #; }
        pulse : V { CLK =P; }
    }
    capture_CDN {
}

```

```

        W _default_WFT_ ;
        forcePI : V { _pi =\r6 #; }
        measurePO : V { _po =\r2 #; }
        pulse : V { CDN =P; }
    }
}
MacroDefs {
    test_setup {
        W _default_WFT_ ;
        V { CLK =0; }
        V { CDN =1; CLK =0; }
    }
}

```

## Updating a Protocol in a Scan Chain Inference Flow

If you import an existing-scan netlist without any test attributes, test DRC can infer the scan structures if you perform the following steps:

1. Specify test clocks and other test attributes in the design.
2. Create a test protocol.
3. Run the `dft_drc` command to infer scan structures.

If scan chain inference is successful, the protocol is updated to contain procedures to shift the scan chain.

## Masking DRC Violations

DFT Compiler includes three commands that are used in support of setting, resetting, or reporting the severity of DRC violations:

- `set_dft_drc_rules`
- `reset_dft_drc_rules`
- `report_dft_drc_rules`

This section describes how to use each of these commands.

## Setting the Severity of DRC Violations

The `set_dft_drc_rules` command is used to modify the severity of selected DRC violations to either ERROR, WARNING, or IGNORE. The following violation codes are supported:

- TEST-504 : Constant 0 violations

- TEST-505 : Constant 1 violations
- D17 : Clock i/p *type cell\_name* cannot capture the data.

The syntax for the `set_dft_drc_rules` command is as follows:

```
set_dft_drc_rules
[-error {drc_error_ID}]
[-warning {drc_error_ID}]
[-ignore {drc_error_ID}]
[-cell {cell_list}]
```

Option	Description
-error {drc_error_ID}	Explicitly changes the severity of the specified violations to ERROR. If you change the severity to ERROR, the <code>dft_drc</code> command will report the specified violations as error and the <code>insert_dft</code> command will fail.
-warning {drc_error_ID}	Explicitly changes the severity of the specified violations to WARNING. If you change the severity to WARNING, the <code>insert_dft</code> command will ignore the violation. However, the <code>dft_drc</code> command will still report the specified violations.
-ignore {drc_error_ID}	Explicitly changes the severity of the specified violations to IGNORE. If you change the severity to IGNORE, the <code>dft_drc</code> and <code>insert_dft</code> commands will ignore the violation.
-cell {cell_list}	Specifies the cells for which the violation severity is to be changed. If no cells are specified, the violation severity is changed for all the cells in the design.

To use this command, you first need to specify the violation by using the `-error`, `-warning`, or `-ignore` options, and then enter the applicable DRC violation ID. You can specify the cells you want to change by using the `-cell` option and entering the names of the applicable cells. If you don't specify the `-cell` option, the `set_dft_drc_rules` command will apply to all cells in the design.

The following set of examples shows how you set the severity:

- The severity of DRC violation TEST-504 for all cells to WARNING
 

```
dc_shell> set_dft_drc_rules -warning {TEST-504}
```
- The severity of DRC violation TEST-505 for cells reg3 and reg4 to WARNING

```
dc_shell> set_dft_drc_rules -warning {TEST-505} \
           -cell {reg3 reg4}
```

- The severity of DRC violation TEST-505 for cell reg6 to IGNORE

```
dc_shell> set_dft_drc_rules -ignore {TEST-505} \
           -cell {reg6}
```

---

## Resetting the Severity of DRC Violations

The `reset_dft_drc_rules` command can be used to change the severity of the violations reported by the `dft_drc` command to the default value. The `reset_dft_drc_rules` command can be used to change the severity only if the severity of that violation has been changed by use of the `set_dft_drc_rules` command. The following violations can be changed with the `reset_dft_drc_rules` command:

- TEST-504 : Constant 0 violations
- TEST-505 : Constant 1 violations
- D17 : Clock i/p *type cell\_name* cannot capture the data.

The syntax for the `reset_dft_drc_rules` command is

```
reset_dft_drc_rules
-violation {drc_error_ID}
[-cell {cell_list}]
```

---

Option	Description
<code>-violation {drc_error_ID}</code>	Explicitly changes the severity of the specified violations to their default value.
<code>-cell {cell_list}</code>	Specifies the cells for which the violation severity is to be changed to their default value. If no cells are specified, the violation severity is changed for all the cells in the design.

---

The following set of examples changes the severity of DRC violation TEST-504 for all cells to default severity ERROR and changes the severity of TEST-505 for cells reg3, reg4 to default the severity ERROR:

```
dc_shell> reset_dft_drc_rules -violation {TEST-504}
```

```
dc_shell> reset_dft_drc_rules -violation {TEST-505} \
           -cell {reg3 reg4}
```

---

## Reporting the Severity of DRC Violations

The `report_dft_drc_rules` command can be used to check the violation severity status on cells in a design.

The syntax for the `reset_dft_drc_rules` command is

```
report_dft_drc_rules
[-all]
-violation {drc_error_ID}
[-cell {cell_list}]
```

Argument	Description
-all	Reports all the cells for which the violation severity has been changed by the user.
-violation {drc_error_ID}	Reports the cells for which the severity of the specified violation(s) has been changed.
-cell {cell_list}	Specifies the cell(s) for which the violation severity is to be reported.

The following example set first changes the severity of DRC violation TEST-504 for all cells to WARNING; TEST-505 for cells reg3, reg4 to WARNING; and TEST-505 for cell reg6 to IGNORE. The corresponding outputs for the `report_dft_drc_rules` command are shown below:

```
dc_shell> set_dft_drc_rules -warning {TEST-504}

dc_shell> set_dft_drc_rules -warning {TEST-505} \
           -cell {reg3 reg4}

dc_shell> set_dft_drc_rules -ignore {TEST-505} \
           -cell {reg6}

dc_shell> report_dft_drc_rules
```

Severity of the following violations have been changed by the user.

```
-----
TEST-504 : Constant 0 violation
TEST-505 : Constant 1 violation
```

```

dc_shell> report_dft_drc_rules -all

Violation Default Specified Range/
  Name      Severity Severity Cell list
-----
TEST-504    error     warning   all cells
TEST-505    error     warning   reg3
              error     warning   reg4
              error     ignore    reg6

dc_shell> report_dft_drc_rules -violation \
          {TEST-504 TEST-505}

Violation Default Specified Range/
  Name      Severity Severity Cell list
-----
TEST-504    error     warning   all cells
TEST-505    error     warning   reg3
              error     warning   reg4
              error     ignore    reg6

dc_shell> report_dft_drc_rules -cell {reg2 reg4}

Cell Violation Default Specified
  Name  Name      Severity Severity
-----
reg2  TEST-504    error     warning
reg4  TEST-504    error     warning
      TEST-505    error     warning

```

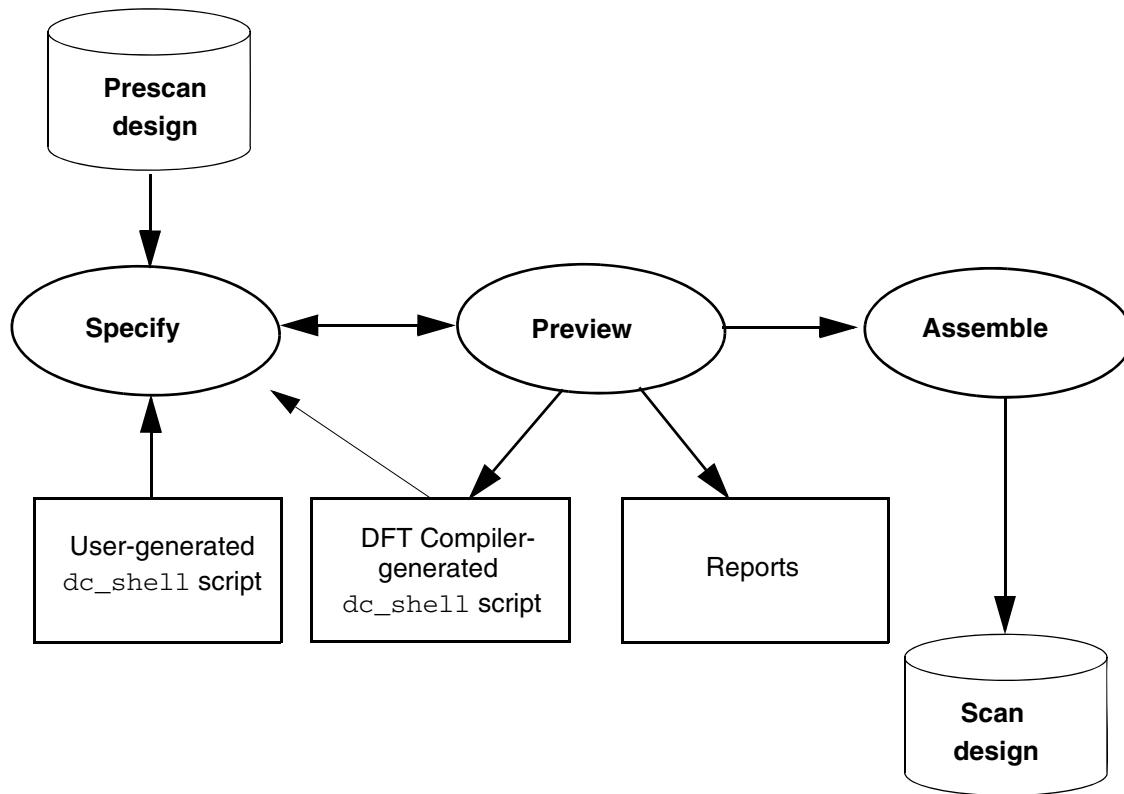
# 6

## Architecting Your Test Design

This chapter describes the basic processes involved in configuring and architecting your test design for scan insertion. The standard DFT architecture process consists of configuring your architecture, building scan chains, connecting test signals, setting test clocks, and analyzing your configurations before and after scan insertion.

[Figure 6-1](#) shows the basic flow of the scan chain generation process.

*Figure 6-1 Scan Chain Generation Process*



This chapter includes the following sections:

- [Configuring Your DFT Architecture](#)
- [Architecting Scan Chains](#)
- [Architecting Scan Signals](#)
- [Architecting Test Clocks](#)
- [Modifying Your Scan Architecture](#)
- [Post-Scan Test Design Rule Checking](#)

---

## Configuring Your DFT Architecture

Before you run scan insertion, you need to configure your DFT architecture. This section includes the following subsections related to the configuration process:

- [Defining Your Scan Architecture](#)
  - [Specifying Individual Scan Paths](#)
  - [Previewing Your Scan Design](#)
- 

### Defining Your Scan Architecture

To define your scan architecture, you need to set design constraints, define any test modes, specify test ports, and identify and mark any cells that you do not want to have scanned.

Use the following script for the basic scan assembly flow:

```
current_design top

/* specify the scan architecture */
set_scan_configuration -chain_count 4

/* create the test protocol */
create_test_protocol

/*check test design rules */
dft_drc

/* preview the scan structures */
preview_dft

/* assemble the scan structures */
insert_dft

/* verify */
dft_drc
report_constraint -all_violators
```

Scan configuration is the specification of global scan properties for the current design. Use the `set_scan_configuration` command to specify global scan properties such as

- Scan style and methodology
- Length and number of scan chains
- Handling of multiple clocks
- Internal and external three-state nets

- Bidirectional ports

Note:

This list of the `set_scan_configuration` command's functionality is not exhaustive. For a complete listing, as well as a description of each option's purpose, see the man page.

DFT Compiler automatically generates a complete scan architecture from the global properties that you have defined.

## Setting Design Constraints

You should set constraints prior to running `insert_dft` because it minimizes constraint violations. Use Design Compiler commands to set area and timing constraints on your design. If you have already compiled your design, you do not need to reset your constraints. For more information about setting area and timing constraints on your design, see the *Design Compiler Reference Manual: Constraints and Timing*.

## Defining Constant Input Ports During Scan

If your design requires a signal to be held constant to satisfy design rules or to enable circuit paths, you need to use the `set_dft_signal` command. For more information, see [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

## Specifying Test Ports

The `insert_dft` command adds scan signals that use existing ports. These ports are identified by using the `set_dft_signal` command. If the tool cannot find existing ports that it can use as test ports, it adds new ports to the design. The `insert_dft` command names the new ports according to the following variables:

- `test_clock_port_naming_style`
- `test_scan_clock_a_port_naming_style`
- `test_scan_clock_b_port_naming_style`
- `test_scan_clock_port_naming_style`
- `test_scan_enable_inverted_port_naming_style`
- `test_scan_enable_port_naming_style`
- `test_clock_in_port_naming_style`
- `test_clock_out_port_naming_style`

---

## Specifying Individual Scan Paths

DFT Compiler supports detailed specification of individual scan paths. Use the following commands to specify the scan architecture:

- `set_scan_element`

Use this command to specify sequential elements that are to be excluded or included in the scan chains. DFT Compiler supports the following types of sequential elements: leaf cells, hierarchical cells, references, library cells, and designs.

Use the `set_scan_element` command sparingly. For best results, use the command only on instances.

- `set_scan_path`

Use this command to specify properties specific to a scan chain, such as name, membership, chain length, clock association, and ordering.

- `set_dft_signal`

Use this command to specify desired port connections and scan chain assignments for test signals.

- `set_autofix_element`

Use this command to control particular bidirectional ports on the top level of the current design.

In case you are unfamiliar with some of the scan path components used in the scan specification commands, [Figure 6-2](#) illustrates the scan path components.

**Figure 6-2 Scan Path Components**

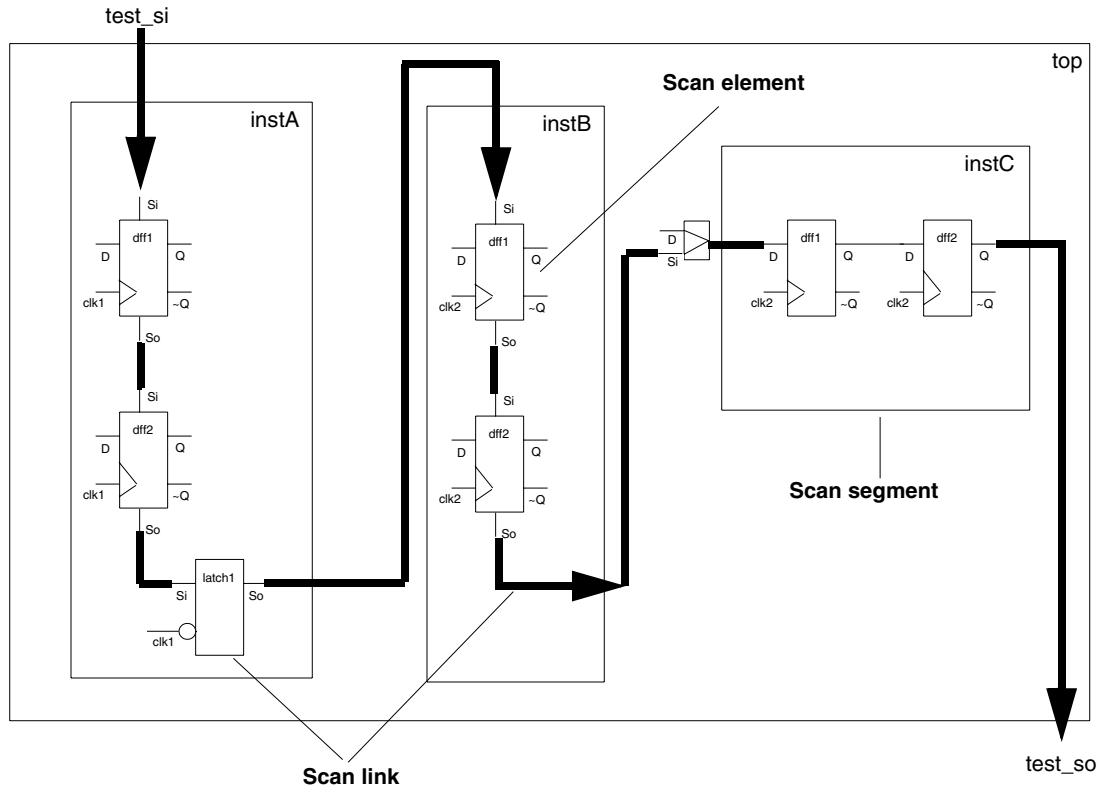


Figure 6-2 shows a single scan path that starts at port `test_si`, which receives the `test_scan_in` scan signal, and ends at port `test_so`, which drives the `test_scan_out` scan signal. Cells `instA/dff1`, `instA/dff2`, `instB/dff1`, and `instB/dff2` are examples of scan elements. The shift register in `instC` is a defined scan segment. In the bottom-up flow, the scan chains in `instA` and `instB` are considered subchains or inferred scan segments. The thick lines represent scan links. The latch (instance `latch1`) is also a scan link.

The following sections discuss some of the situations you might encounter during scan specification. See [Chapter 4, “Performing Scan Replacement,”](#) for scan style selection considerations.

## Previewing Your Scan Design

Use the `preview_dft` command to preview your scan design. The command generates a scan chain design that satisfies scan specifications on the current design and displays the scan chain design. This allows you to preview your scan chain designs without synthesizing them and to change your specifications to explore the design space as necessary.

When you are using `dft_drc`, a valid protocol must exist before you run `preview_dft`. This is created by `create_test_protocol`. If no `dft_drc` run has occurred, `preview_dft` will run it automatically. The `preview_dft` and `insert_dft` commands use the same algorithms to design scan chains.

[Example 6-1](#) shows an example display generated by the `preview_dft` command.

#### *Example 6-1 Display Generated by the preview\_dft Command*

```
*****
Preview DFT report
Design: P
Version: 1998.02
Date: Wed Apr 21 11:25:53 1999
*****
Number of chains: 1
Test methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix
Scan chain '1' (test_so) contains 4 cells
```

### **Using `preview_dft` Versus `report_scan_path`**

The `preview_dft` command provides a summary of the scan chains you are going to get. The `report_scan_path -view existing_dft -chain all` command presents a summary of the scan chains you currently have.

---

## **Architecting Scan Chains**

The `set_scan_configuration` command enables you to specify the scan chain design. This command controls almost all aspects of how the `insert_dft` command makes designs scannable. Exceptions are specific to particular scan chains and are specified in the `set_scan_path` command options.

This section has the following subsections related to architecting scan chains:

- [Specifying a Scan Chain for the Current Design](#)
- [Controlling the Scan Chain Length](#)
- [Determining the Scan Chain Count](#)
- [Balancing Scan Chains](#)
- [Controlling the Routing Order](#)
- [Routing Scan Chains and Global Signals](#)

- Rerouting Scan Chains
  - Stitching Scan Chains Without Optimization
  - Using Existing Subdesign Scan Chains
  - Uniquifying Your Design
  - Reporting Scan Path Information on the Current Design
- 

## Specifying a Scan Chain for the Current Design

Use the `set_scan_path` command to specify a scan chain for the current design.

The `set_scan_path` command enables you to

- Specify a name for a scan chain
- Allocate scan cells, scan segments, scan links, and subdesign scan chains to scan chains and specify the ordering of the scan chain
- Specify a dedicated scan-out port
- Limit a scan chain's elements to only those components you specify or enable DFT Compiler to balance scan chains by adding more elements
- Specify scan chain length
- Assign scan chains to clock domains

Scan chain elements cannot belong to more than one chain. The command options are not incremental. Where `set_scan_path` commands conflict, the preview command (`preview_dft`) and scan insertion command (`insert_dft`) execute the most recent command.

---

## Controlling the Scan Chain Length

DFT Compiler supports controlling the length of scan chains. Controlling the length of the scan chain can help to balance the scan configuration in a design that has bottom-up or system-on-a-chip (SoC) scan insertion.

### Specifying Limits for Individual Scan Chain Length

You can set the length of an individual scan chain by using the `set_scan_path` command.

```
dc_shell> set_scan_path chain_name -exact_length n
```

This command enables you to identify a specific scan chain and define its length. Scan chain length is defined in terms of the number of scan cells in the chain.

## Specifying the Global Scan Chain Exact Length

You can specify an exact length for all scan chains by using the `-exact_length` option of the `set_scan_configuration` command.

For example, suppose your design has 420 flip-flops, and you want an exact length of 80 flip-flops per scan chain. In this case, specifying `set_scan_configuration -exact_length 80` creates five chains with 80 flip-flops and one chain with 20 flip-flops.

### Caution!

The exact length feature is meant to be used only with standard scan, including multimode standard scan. It is not currently supported in DFT MAX adaptive scan. Make sure you do not use this feature with adaptive scan.

Note the following properties of this feature:

- The `-exact_length` option takes precedence over both the `-max_length` option and the `-chain_count` option.
- The `report_scan_configuration` command reports the value of the exact length configuration.
- The user-specified chain configuration is preserved.
- The quality of the result of using this option on designs containing complex segments cannot be guaranteed.

## Specifying the Global Scan Chain Length Limit

Setting the scan chain length limit helps with bottom-up scan insertion by balancing scan chains more efficiently at the top level. Setting a limit on the length of scan chains allows for design constraints related to pin availability or test vector depth.

Use the `set_scan_configuration -max_length` command to specify the length of a scan chain:

```
dc_shell> set_scan_configuration \
           -max_length 7
```

For example, if you set the scan chain length limit to 7 registers for a single-clock, single-edge design with 29 flip-flops, `insert_dft` creates five scan chains with lengths of 6, 6, 6, 6, and 5 registers. This scan chain allocation meets the scan chain length limit while also balancing each scan chain's length as closely as possible.

Note:

Specifying both the `-max_length` option and the `-chain_count` option (described in the next section) might result in conflicting scan chain allocations. In such a case, the `-max_length` option takes precedence.

---

## Determining the Scan Chain Count

Use these questions to decide how many scan chains to request:

- How many scan chains does your semiconductor vendor allow?

Many semiconductor vendors restrict the maximum number of scan chains due to software or tester limitations. Before performing scan specification, check with your semiconductor vendor for the maximum number of scan chains supported.

- How many clock domains exist in your design?

To prevent timing problems on the scan path in multiplexed flip-flop designs, allocate a scan chain for each clock domain (DFT Compiler default behavior). DFT Compiler considers each edge of a clock a unique clock domain. Multiple clock domains do not affect the number of scan chains in scan styles other than multiplexed flip-flop.

- How much time will it take to test your design?

Because the test time is proportional to the length of the longest scan chain, increasing the number of scan chains reduces the test time for a design.

Use the `set_scan_configuration -chain_count` command to specify the number of scan chains.

```
dc_shell> set_scan_configuration -chain_count 7
```

By default, DFT Compiler generates

- One scan chain per clock domain if you select the multiplexed flip-flop scan style
- One scan chain if you select any other scan style

Note:

The `-max_length` argument and the `-chain_count` argument are mutually exclusive. If you use both arguments, `-max_length` takes precedence over `-chain_count`.

---

## Balancing Scan Chains

By default, the `insert_dft` command balances the number of cells in each scan chain, that is, DFT Compiler always rebalances scan chains. If you want to create unbalanced scan chains, you have to use the `use_scan_path` command or the test model flow.

If unbalanced scan chains occur due to multiplexed flip-flop design with multiple clock domains or a bottom-up design flow, you can override DFT Compiler default behavior to force balanced scan chains. When overriding the default behavior, always use `preview_dft` to verify that the result meets your requirements.

## Multiple Clock Domains

For multiplexed flip-flop designs, DFT Compiler allocates cells to scan chains based on clock domain. You can override this default behavior by using the `set_scan_configuration -clock_mixing` command.

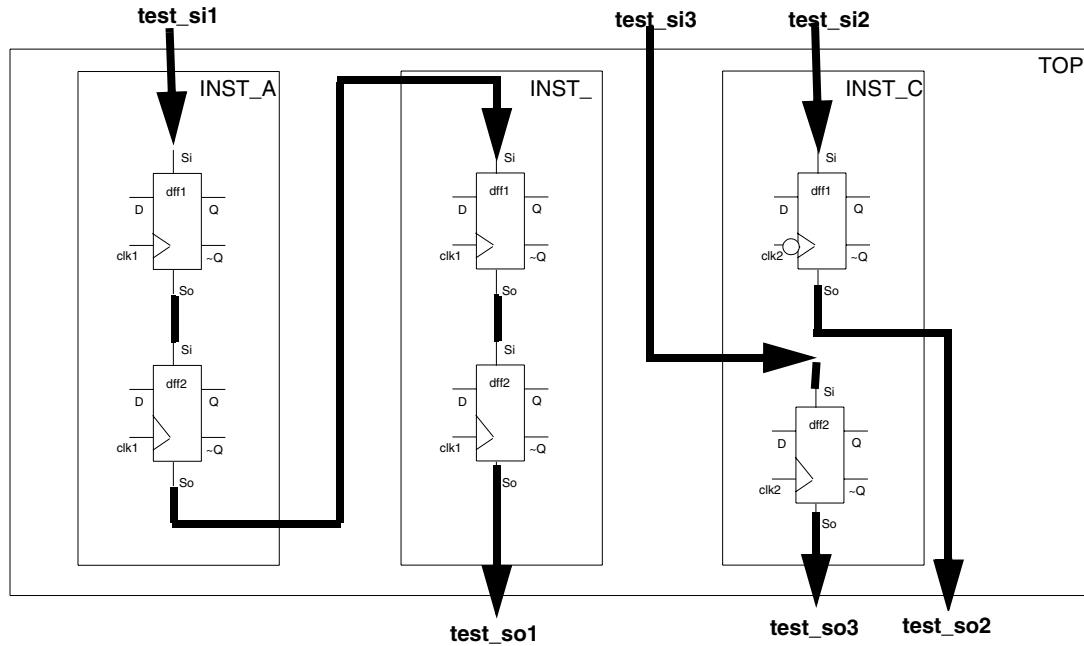
For example, assume that you have a design with three clock domains and your desired scan architecture contains two balanced scan chains.

```
dc_shell> set_dft_signal -view existing_dft \
                  -type ScanClock -timing [list 45 55] \
                  -port {clk1, clk2}

dc_shell> set_scan_configuration -chain_count 2
```

In the default case shown in [Figure 6-3](#), DFT Compiler overrides your request for two chains and generates three scan chains, one for each clock domain (clk1, positive-edge clk2, negative-edge clk2). Because the clock domains contain unequal numbers of cells, DFT Compiler generates unbalanced scan chains.

Figure 6-3 Unbalanced Scan Chains Due to Multiple Clock Domains



You can reduce the number of scan chains and achieve slightly better balancing by mixing clock edges within a single chain.

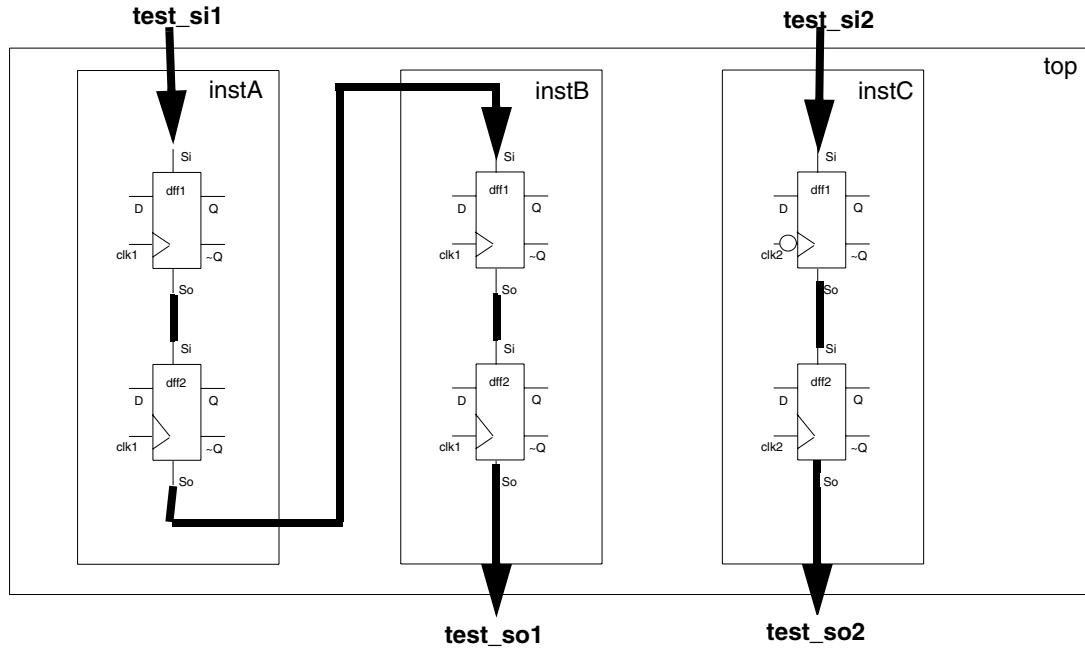
```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port {clk1, clk2}

dc_shell> set_scan_configuration -chain_count 2

dc_shell> set_scan_configuration -clock_mixing \
    mix_edges
```

Mixing clock edges in a single scan chain produces a small timing risk. DFT Compiler automatically orders the cells within the scan chain so the cells clocked later in the cycle appear earlier in the scan chain, resulting in a functional scan chain. [Figure 6-4](#) shows the scan architecture when you allow edge mixing.

Figure 6-4 Better Balancing With Mixed Clock Edges



You can balance the scan chains by mixing clocks:

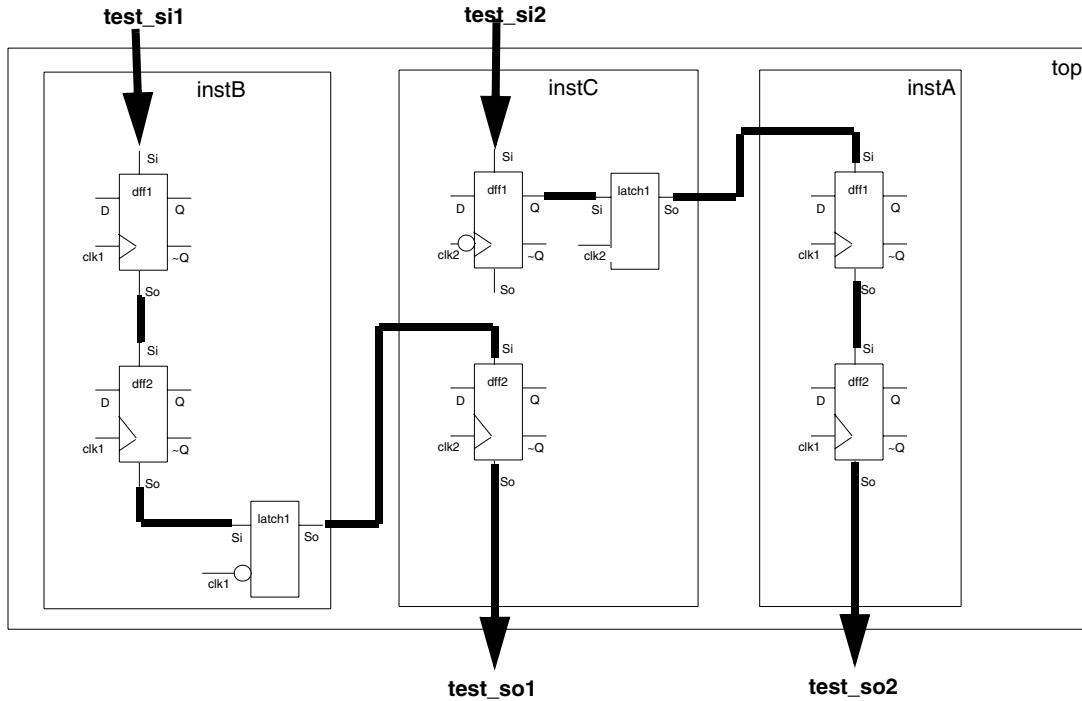
```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port {clk1, clk2}

dc_shell> set_scan_configuration -chain_count 2

dc_shell> set_scan_configuration -clock_mixing \
    mix_clocks
```

Mixing clocks in a single scan chain can produce a large timing risk. To reduce this risk, DFT Compiler adds lock-up latches to the scan path wherever clock changes occur. DFT Compiler guarantees a functional scan chain under ideal clock timing, that is, one with no clock skew. [Figure 6-5](#) shows the resulting scan architecture.

Figure 6-5 Balanced Scan Chains With Mixed Clocks



See the section “Handling Multiple Clock Designs” for clock-mixing considerations. See “[Using Scan Lock-Up Elements](#)” on page 6-43 for details about scan lock-up latches.

## Multibit Components and Scan Chains

You can specify whether multibit components are implicitly treated as synthesizable segments by using the `set_scan_configuration -preserve_multibit_segment` command. This command applies globally to the current design and overrides any specification on subdesigns. By default, multibit components are implicitly treated as synthesizable segments. Note, for example, the command

```
dc_shell> set_scan_configuration \
           -preserve_multibit_segment false
```

This command ensures that each bit of all sequential multibit components is treated individually when the `insert_dft` command builds scan chains. Scan replacement is still performed homogeneously for each multibit component.

Implicit synthesizable specifications cannot be removed; they can only be overridden.

You can use the `remove_scan_specification -segment` command to remove explicit specifications of synthesizable segments, and you can use the `set_scan_path` command to specify a scan chain that has multibit and synthesizable segments.

The `preview_dft` command reports on synthesizable segments when you specify the `-show segments` option.

**Example 6-2** shows a sample script that creates a multibit component. This script makes two assumptions:

- Design B has sequential cells `r[0]` through `r[3]`.
- Design B is instantiated twice in design A as `B1` and `B2`.

#### *Example 6-2 Multibit Cell Sample Script*

```
current_design = B

/* create multibit reg in design B */
create_multibit -name reg {r*}
current_design = A

/* Specify scan chain */
set_scan_path chain1 -view spec -ordered_elements {B1/r[0]
B1/r[1] B2/reg B1/r[2] B1/r[3]}
```

**Example 6-3** shows the resulting scan chain.

#### *Example 6-3 Resulting Scan Chain for Sample Script*

```
test_si ->
b1/r[0] ->
b1/r[1] ->
b2/r[3] ->
b2/r[2] ->
b2/r[1] ->
b2/r[0] ->
b1/r[2] ->
b1/r[3] ->
test_so
```

---

## Controlling the Routing Order

Use the `set_scan_path` command to control the routing order explicitly. You can specify the routing order of nonscan as well as scanned sequential cells. Each `set_scan_path` command generates a scan chain; DFT Compiler uses the first command argument as the scan chain name. If you enter multiple `set_scan_path` commands with the same scan chain name, DFT Compiler uses only the last command entered.

You can provide partial or complete scan ordering specifications. Use the `-complete true` option to indicate that you have completely specified a scan chain. DFT Compiler does not add cells to a completely specified scan chain. If you provide a partial scan-ordering specification, DFT Compiler might add cells to the scan chain. DFT Compiler places the cells specified in a partial ordering at the end of the scan chain.

DFT Compiler validates the specified scan ordering. The checks performed by DFT Compiler include

- Cell assignment

DFT Compiler verifies that you have not assigned a cell to more than one scan chain. A violation triggers the following message during execution of the `set_scan_path` command:

```
Error: Scan chains '%s' and '%s' have common elements.  
(TESTDB-256)  
Common elements are:  
    %s
```

DFT Compiler discards the second scan path specification, keeping the first scan path specification which contains the common element).

- Clock ordering

DFT Compiler verifies that the active clock edge of the next scan cell occurs concurrently or before the active clock edge of the current scan cell or that the active edge can be synchronized with a scan lock-up latch.

If your multiplexed flip-flop design violates this requirement, DFT Compiler reorders the invalid mixed-clock scan chains and triggers the following message during execution of the `preview_dft` command:

```
Warning: User specification of chain '%s' has been  
reordered. (TEST-342)
```

- Clock mixing

DFT Compiler verifies that all cells on a scan path have the same clock unless you have specifically requested clock mixing. A violation triggers the following message during execution of the `preview_dft` command:

```
Warning: Chain '%s' has elements clocked by different  
clocks. (TEST-353)
```

DFT Compiler creates the requested scan chain. Unless you have disabled scan lock-up latch insertion, DFT Compiler inserts a scan lock-up latch between clock domains.

- Black box cells

DFT Compiler verifies that the specified cells are valid scan cells. If a sequential cell has a test design rule violation or has a `scan_element false` attribute, DFT Compiler considers it a black box cell. A violation triggers the following message during execution of the `preview_dft` command:

```
Warning: Cannot add '%s' to chain '%s'. The element is  
not being scanned. (TEST-376)
```

DFT Compiler creates the requested scan chain without the violating cells.

---

## Routing Scan Chains and Global Signals

Most scan cells have both a scan output pin (`test_scan_out`) and an inverted scan output pin (`test_scan_out_inverted`) defined in the technology library. If the functional path through a sequential cell has timing constraints, DFT Compiler automatically selects the scan output pin with the most timing slack for use as the scan output. To disable this behavior, set the `test_disable_find_best_scan_out` variable to `true`.

Scan chain allocation and ordering might differ between a top-down implementation and a bottom-up implementation because

- DFT Compiler does not modify subdesign scan chains unless explicitly specified in your scan configuration
- DFT Compiler overrides alphanumeric ordering to provide a shared scan output connection on the current design but not on subdesigns

---

## Rerouting Scan Chains

The scan specification process previously discussed enables both initial routing and rerouting of your design. However, the specify-preview loop runs faster than the specify-synthesize loop. Try to avoid rerouting by iterating through the specify-preview loop until the scan architecture meets your requirements.

To optimize the design during scan assembly, DFT Compiler

- Performs scan-specific optimizations to reduce the timing impact of scan routing.

In many cases, the scan path uses the functional output as the scan output. The scan path routing increases the output load on the functional output. If you used test-ready compile for scan replacement, this additional loading is compensated for during optimization. If you used constraint-optimized scan insertion, DFT Compiler uses focused optimization techniques during scan assembly to minimize the impact of the additional load on the overall design performance.

- Replaces unrouted scan cells with their nonscan equivalents.

If you used test-ready compile for scan replacement, your design might contain unrouted scan cells. These unrouted scan cells occur because the cell has a test design rule violation.

DFT Compiler replaces these unrouted scan cells with their nonscan equivalents during execution of the `insert_dft` command.

Your design might contain sequential cells that are defined in the technology library as scan cells but can also implement functional logic in your design. These cells have functional connections to both the data and scan inputs, and DFT Compiler does not modify these cells during scan assembly.

- Fixes hold-time violations on the scan path if the clock net has the `fix_hold` attribute.

## **Stitching Scan Chains Without Optimization**

In some circumstances, you might want to stitch your design's scan chains together but avoid the optimization step. This process is referred to as "rapid scan synthesis." Such circumstances might include

- Stitching completed subdesigns together
- Performing synthesis and scan insertion in the logical domain and optimizations in the physical domain
- Performing analysis on the design

## **Specifying a Stitch-Only Design**

When DFT Compiler performs scan stitching without optimization, it still performs comprehensive logical DFT design rule checks, but it eliminates the runtime-intensive synthesis mapping, timing violation fixing, and design rule fixing steps.

Consequently, the design is only stitched and no further optimizations are performed on the design.

To enable scan stitching without optimization, use the following command:

```
dc_shell> set_scan_replacement
```

## **Mapping the Replacement of Nonscan Cells to Scan Cells**

You might want to stitch a design that has not been scan-replaced. The `set_dft_insertion_configuration -synthesis_optimization none` command can perform scan replacement on designs of this sort.

If a simple one-to-one mapping of a nonscan to a scan cell is not available in the library, DFT Compiler performs a cell decomposition followed by a sequential mapping algorithm. You can avoid this step by using the command

```
dc_shell> set_scan_replacement \
           -nonscan nonscan_cell_list \
           -multiplexed_flip_flop scan_cell
```

The options in this command should always be specified as a pair. If they are not, an error results. Many cells can be listed in the `-nonscan` option, but only one cell can be listed in the `-multiplexed_flip_flop` option. You can use the `-lssd` option in place of the `-multiplexed_flip_flop` option.

If you use this command and a scan cell definition exists in the ASIC library, the mapping you specified with the `set_scan_replacement` command overrides the library definition. This command is global in nature; it affects the entire design.

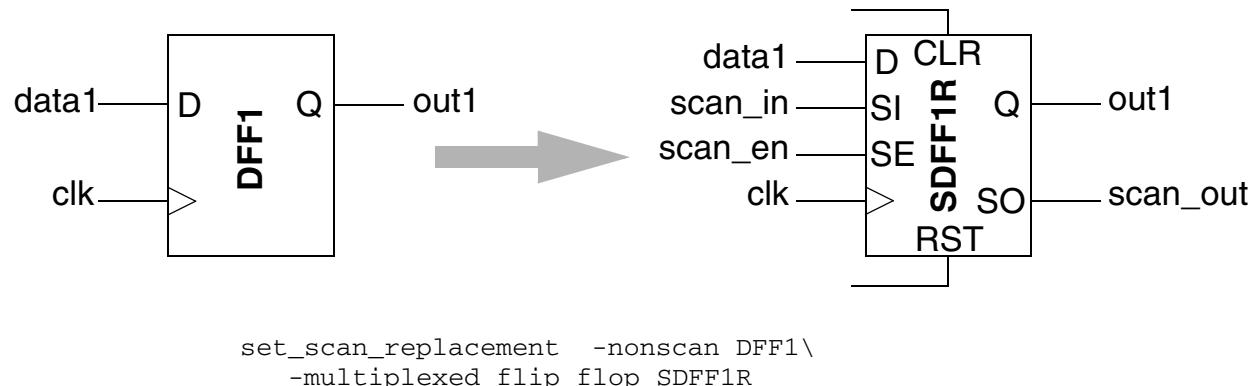
For example, the scan cell DFFS1 is a direct mapping of the nonscan cell DFFD1, but with scan pins. To specify the mapping of the DFFD1 nonscan cell to the DFFS1 scan cell, use the following command:

```
dc_shell> set_scan_replacement -nonscan DFFD1 \
           -multiplexed_flip_flop DFFS1
```

### Few-Pins-to-Many-Pins Scan Cell Replacement Scenario

If you select a scan cell that has more pins than the nonscan cell it replaces, the extra pins are tied to the inactive state and a warning is issued. You can fix this problem by respecifying a more appropriate cell with the `set_scan_replacement` command.

*Figure 6-6 Few-to-Many Scenario (Accepted)*



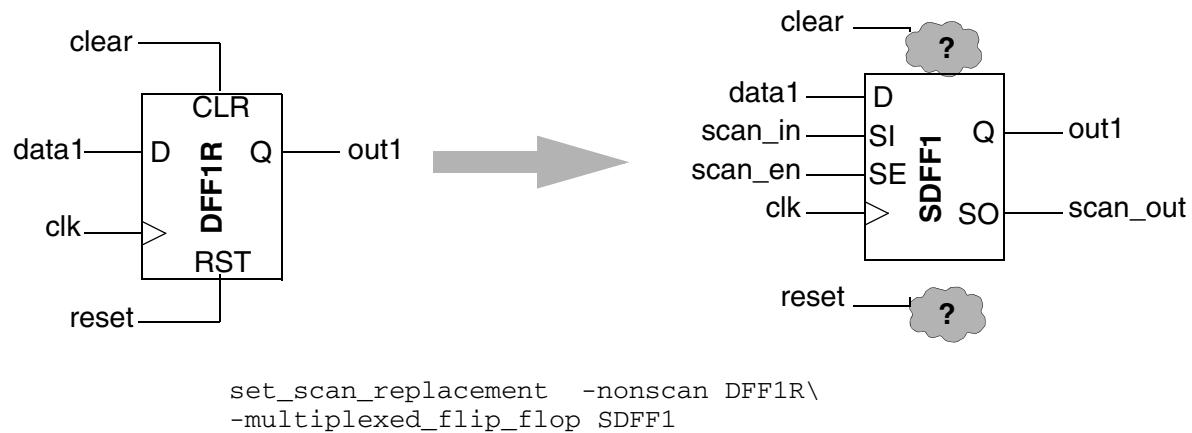
In [Figure 6-6](#), the replacement cell has more inputs and outputs than required by the nonscan cell. The unused pins of the scan cell are left unconnected.

### Many-Pins-to-Few-Pins Scan Cell Replacement Scenario

Alternatively, if you select a scan cell that has fewer pins than the nonscan cell it replaces, the extra pins are left unconnected. To avoid problems with bad logic, an error message is issued and the replacement does not occur. You can fix this problem by respecifying a more appropriate cell with the `set_scan_replacement` command.

In [Figure 6-7](#), for example, the clear and reset pins do not exist on the scan cell. They are left unconnected, causing bad logic.

*Figure 6-7 Many-to-Few Scenario (Rejected)*



### Conditions Under Which Scan Cells Are Excluded or Nonscan Cells Become Scan Cells

This section describes the conditions under which

- A sequential cell is excluded from the DRC violations
- A sequential cell is excluded from the scan chains
- A nonscan cell becomes a scan cell
- A scan cell is unscanned

#### DRC Violation Report (`dft_drc`)

A cell XYZ should be reported as a valid nonscan cell by DRC if the following command is used:

```
dc_shell> set_scan_element false XYZ
```

### Scan Architect (insert\_dft)

A cell XYZ will not be part of the scan chains if any of the following conditions are met:

- The following command is used:

```
dc_shell> set_scan_element false XYZ
```

- The cell XYZ is DRC violated

- The following command is used:

```
dc_shell> set_scan_configuration -exclude_elements XYZ
```

Note:

You use `set_scan_configuration -exclude` to prevent flip-flops from being stitched into the scan chains. The difference between using `set_scan_configuration -exclude` and `set_scan_element false` is that the former command does not unscan the specified flip-flops during `insert_dft` whereas the latter command does unscan the flip-flops.

### Scan Replacement (insert\_dft)

A nonscan flip-flop cell, FF, will become a scan cell in either of the two following cases:

- Both of the following conditions are met:

- The nonscan flip-flop cell is not DRC violated
- The following command is used:

```
dc_shell> set_scan_element true FF
```

- Both of the following conditions are met:

- The following command is used:

```
dc_shell> set_scan_element true FF
```

- The following command is used:

```
dc_shell> set_scan_configuration -exclude_elements FF
```

A scan cell, SFF, will be converted to a nonscan cell in either of the two following cases:

- The following command is used:

```
dc_shell> set_scan_element false SFF
```

- Both of the following conditions are met:

- The scan cell, SFF, is DRC violated
- The following command is used:

```
dc_shell> set_dft_insertion_configuration \
          -unscan true
```

---

## Using Existing Subdesign Scan Chains

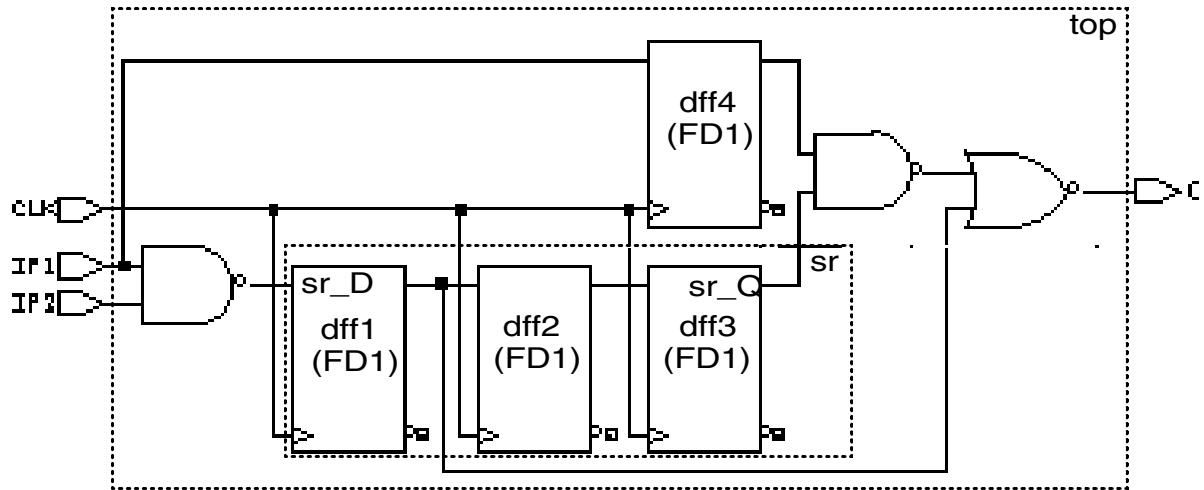
A subdesign scan chain uses subdesign ports for all test signals. DFT Compiler can infer subdesign scan chains during test design rule checking.

To reuse existing subdesign scan chains, follow these steps:

- Set the current design to the subdesign containing the existing scan chain.
- Use the `set_dft_signal` command to identify the existing scan ports.
- Create a test protocol by using the `create_test_protocol` command.
- Set the current design to the design where you are assembling the scan structures.
- Use the `set_scan_path` command to control the scan chain connections, if desired.

For example, subdesign sr in [Figure 6-8](#) contains a shift register. The shift register performs a serial shift function, so DFT Compiler can use this existing structure in a scan chain. The scan input signal connects to subdesign port `sr_D`. The scan output signal connects to subdesign port `sr_Q`. The shift register always performs the serial shift function, so the shift register does not need a scan enable signal.

Figure 6-8 Subdesign Scan Chain Example Before Scan Insertion



Use the following command sequence to infer the subdesign scan chain in module sr:

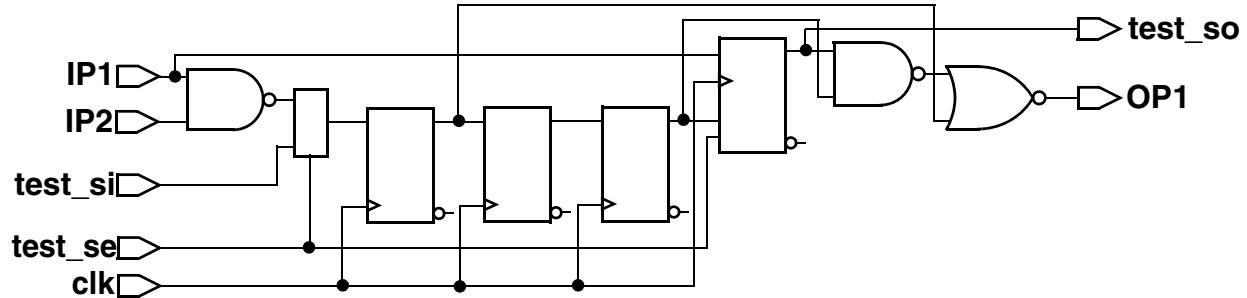
```
dc_shell> current_design sr
dc_shell> set_dft_signal -view spec -port sr_D \
-type ScanDataIn
dc_shell> set_dft_signal -view spec -port sr_Q \
-type ScanDataOut
dc_shell> create_test_protocol
dc_shell> dft_drc
```

Use the following command sequence to include this scan chain in a top-level scan chain:

```
dc_shell> current_design top
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> insert_dft
dc_shell> dft_drc
```

[Figure 6-9](#) shows the top-level scan chain, which includes the subdesign scan chain. DFT Compiler added a multiplexer, controlled by the scan enable signal, to select between the functional data input and the scan input. The hierarchical cell name determines the location of the subdesign scan chain in the top-level scan chain.

Figure 6-9 Subdesign Scan Chain Example After Scan Insertion



## Uniquifying Your Design

When you run the `insert_dft` command, DFT Compiler automatically assigns a unique name to any subdesigns that changed during the scan insertion process. The default naming convention saves subdesign A as A\_test\_1. If two instances of subdesign A are different, they are saved as A\_test\_1 and A\_test\_2. The following scenarios illustrate examples in which unique names are assigned to instances of a subdesign:

- You specify a different scan ordering in each instance of the same reference design.  
For example, if you route and rebalance a design so that two instances of the subdesign have different scan chain ordering, the `insert_dft` command uniquifies the design.
- The `insert_dft` command identifies different solutions during constraint optimization and design rule fixing.

Constraint optimization and design rule fixing are features of the `insert_dft` command. To eliminate unnecessary uniquification, turn off these features by entering the following commands:

```
dc_shell> set_dft_insertion_configuration \
           -synthesis_optimization none
```

- There are scan violations in one instance but not in another instance, and `insert_dft` repairs one but not the other.

You can choose the suffix that gets appended to the design name to create the unique name. The naming convention for the suffix appended to the design name is controlled by the following command:

```
set insert_test_design_naming_style name
```

In the previous example, the default name is `design_name_test_counter`.

Note:

To prevent unification of your design, enter the command

```
dc_shell> set_dft_insertion_configuration \
           -preserve_design_name true
```

---

## Reporting Scan Path Information on the Current Design

Use the `report_scan_path` command to display scan path information for the current design.

Note:

To show changes caused by running the `insert_dft` command, you must run the `dft_drc` command before the `report_scan_path` command. Running an incremental compile or any other command that changes the database causes the `dft_drc` results to be discarded. In such a case, you need to run `dft_drc` again before you use `report_scan_path`.

[Example 6-4](#) shows the type of information displayed by the `report_scan_path -chain all` command.

### *Example 6-4 Scan Path Information Displayed by the report\_scan\_path Command*

```
Report from report_scan_path
*****
Report: scan_path
Design: P
*****
Complete scan chain (test_si --> test_so) contains 4 cells:
test_si  ->
          instA/dff1->
          instA/dff2->
          instB/dff1->
          instB/dff2->
          test_so
```

For more information on the options of the `report_scan_path` command, see the man pages.

---

## Architecting Scan Signals

For test design rule checking to recognize test ports in your design, your scan-inserted design must have appropriate `signal_type` attributes on test ports. When you build scan chains by using the `insert_dft` command, all of these attributes are automatically set.

The following topics discuss the process for architecting scan signals:

- Specifying Scan Signals for the Current Design
- Selecting Test Ports
- Suppressing Replacement of Sequential Cells
- Changing the Scan State of a Design
- Removing Scan Specifications
- Keeping Specifications Consistent
- Synthesizing Three-State Disabling Logic
- Configuring Three-State Buses
- Handling Bidirectional Ports
- Using Scan Lock-Up Elements
- Assigning Test Port Attributes

---

## Specifying Scan Signals for the Current Design

Use the `set_dft_signal` command to specify one or more scan signals for the current design.

Table 6-1 provides a list of attribute value assignments.

*Table 6-1 Attribute Value Assignments for Test Signals*

Test I/O port signal	Attribute value	Valid on input	Valid on output	Valid on three-state output	Valid on bidirectional
Scan-in	ScanDataIn	Yes	No	No	Yes
Scan-out	ScanDataOut	No	Yes	Yes	Yes
Scan-enable	ScanEnable	Yes	No	No	Yes
Bidirectional enables	InOutControl	Yes	No	No	*
Asynchronous control ports	Reset	Yes	No	No	Yes

\*Not recommended: complex methodologies required.

The following is an example of the `set_dft_signal` command specifying a scan-in port. If you enter

```
dc_shell> set_dft_signal -view spec -port scan_in \
           -type ScanDataIn
```

DFT Compiler responds with the following:

```
Performing set_dft_signal on port 'scan_in'.
```

In the preceding example, `-view spec` option indicates that the specified ports are to be used during DFT scan insertion and that DFT Compiler is to perform the connections. In this example, `scan_in` is the name of the scan-in port that the `insert_dft` command uses. (The other value of the `-view` argument is `-existing_dft`, which directs the tool to use the specified ports as is because they are already connected.)

When the `insert_dft` command creates additional ports for scan test signals, it assigns a name to each new port. You can control the naming convention by using the port naming style variables shown in [Table 6-2](#).

*Table 6-2 Port Naming Style Variables*

Name	Default value
<code>test_scan_in_port_naming_style</code>	<code>test_si%s%s</code>
<code>test_scan_out_port_naming_style</code>	<code>test_so%s%s</code>
<code>test_scan_enable_port_naming_style</code>	<code>test_se%s</code>
<code>test_scan_enable_inverted_port_naming_style</code>	<code>test_sei%s</code>
<code>test_clock_port_naming_style</code>	<code>test_c%s</code>
<code>test_scan_clock_port_naming_style</code>	<code>test_sc%s</code>
<code>test_scan_clock_a_port_naming_style</code>	<code>test_sca%s</code>
<code>test_scan_clock_b_port_naming_style</code>	<code>test_scb%s</code>
<code>test_mode</code>	<code>test_mode%</code> s
<code>test_point_clock</code>	<code>none</code>

Follow these guidelines when using the `set_dft_signal` command:

- Use the `set_dft_signal` command for scan insertion and for design rule checking. The `set_dft_signal` command indicates I/O ports that are to be used as scan ports. After the `insert_dft` command connects these ports, it places the necessary `signal_type` attributes on the ports for post-insertion design rule checking.
- Use the `set_dft_signal -view existing_dft` command if you read in an ASCII netlist and you need to perform design rule checking. Before you use the `set_dft_signal` command, the ASCII netlist does not contain the `signal_type` attributes annotated by scan insertion. Without these attributes, `dft_drc` does not know which ports are scan ports and therefore reports that the design is untestable.
- Use the `set_dft_signal -view existing_dft` command if the ports in your design are already connected and no connection is to be made by DFT Compiler.
- Use the `set_dft_signal -view spec` command if the connections do not exist in your design and you expect DFT Compiler to make the connections for you.

## Using the -view specification and existing Arguments

Unlike other tools used in the implementation flow, DFT Compiler changes the functionality of your design such that the design can operate in either functional (“mission”) mode or test mode.

In order to construct this dual modality, DFT Compiler needs to know what already exists in the design. You use the `-view existing` option in the `set_dft_signal` command to provide such information. The tool then uses this information to perform pre-insertion design rule checking (DRC) in order to determine the elements that can be incorporated into scan chains.

Typical examples that use the `-view existing` option include

- Clock signals:

```
set_dft_signal -view existing -type ScanClock -port \
    clk -timing {45 55}
```

- Asynchronous set and reset signals:

```
set_dft_signal -view existing -type Reset -port rst \
    -active_state 0
```

By default, DFT Compiler creates new ports to the design. You have the capability of specifying which existing ports the tool uses to build the DFT structures by using the `-view spec` option in the `set_dft_signal` command.

Typically, the `-view spec` option can be used to specify ports that are to function as scan-in and scan-out ports (either dedicated scan-in and scan-out ports or shared functional ports used also as scan-in and scan-out ports), such as

```
set_dft_signal -view spec -type ScanDataIn -port scan_in_1
set_dft_signal -view spec -type ScanDataOut -port scan_out_1
```

and

```
set_dft_signal -view spec -type ScanDataIn -port data_in_bus_2
set_dft_signal -view spec -type ScanDataIn -port data_out_bus_2
```

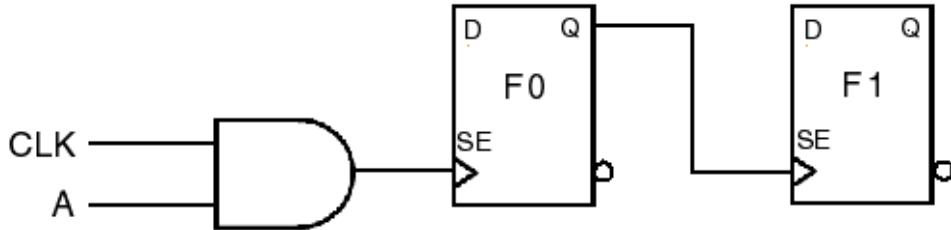
As a general rule,

- If the information is needed for pre-insertion DRC, then it should be specified by using the `-view existing` option.
- If the information is needed to build DFT structures, then it should be specified by using the `-view spec` option.

### Using -view existing -type Constant versus Using -view spec -type TestMode

Consider the design example shown in [Figure 6-10](#).

*Figure 6-10 Design With an Uncontrolled Clock Signal*



In order for the clock signal CLK to reach flip-flop F0 during test, you would specify

```
set_dft_signal -view existing -type Constant -active_state 1 -port A
```

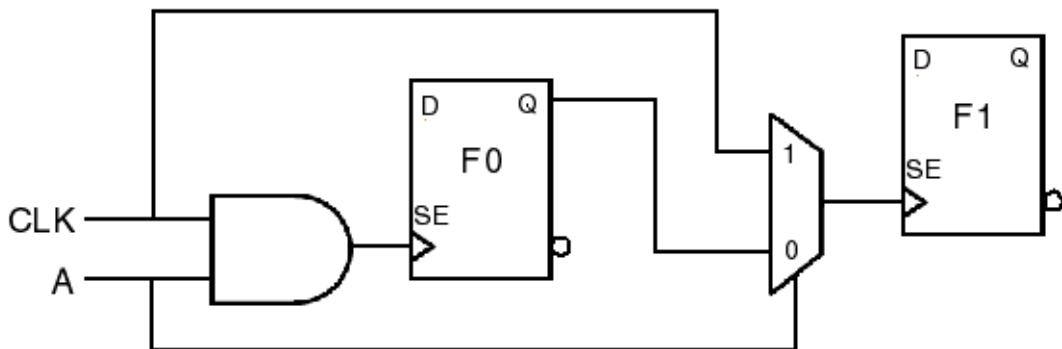
But then the clock signal at flip-flop F1 would remain uncontrolled.

You can enable the AutoFix capability to make the clock signal at F1 controllable. By default, AutoFix creates a new port, called “test\_mode” that is set to 1 during test, which makes flip-flop F1 controllable. However, you can use port A as the controlling signal by specifying

```
set_dft_signal -view spec -type TestMode -port A
```

After the `insert_dft` command is run, the design is changed as shown in [Figure 6-11](#), which avoids creating an extra test mode port.

*Figure 6-11 Design With a Controlled Clock Signal*



---

## Selecting Test Ports

Sharing ports between test and normal operation minimizes the number of dedicated test ports required for internal scan. By default, DFT Compiler minimizes the number of dedicated test ports by sharing scan outputs with functional ports when the design contains scannable cells that directly drive functional ports. If your semiconductor vendor does not support this configuration, you can request dedicated scan output ports. Always use dedicated ports for scan-enable and test clock signals.

To enable simulation by using the same test bench for the RTL and gate-level implementations of your design, you can define unconnected ports in your RTL description for use as scan ports. These unconnected ports are called dummy scan ports. If you defined dummy scan ports for your design, use the `set_dft_signal` command to instruct DFT Compiler to use these ports:

```
dc_shell> set_dft_signal -view spec -port \
              dummy_scan_in -type ScanDataIn

dc_shell> set_dft_signal -view spec -port \
              dummy_scan_en -type ScanEnable \
              -active_state 1

dc_shell> set_dft_signal -view spec -port \
              dummy_scan_out -type ScanDataOut
```

## Sharing Scan-In Pins With Multiple Scan Chains

The following methods can be used for sharing scan-in pins:

- Explicitly specifying scan-in pin sharing
- Scan-in pin sharing by default

### Explicitly Specifying Scan-In Pin Sharing

In a simple flow, scan-in pin sharing is specified by use of the `set_dft_signal` command. The chains assigned to the scan-in pin are specified as a set. The sharing functionality is implemented during scan architecture. The chains are all connected to the same scan-in pin.

For example, to connect three chains—chain1, chain2, and chain3—to the same scan-in pin, use the following commands:

```
dc_shell> set_dft_signal -view spec \
              -port test_si1 -type ScanDataIn

dc_shell> set_scan_path [chain1 chain2 chain3] \
              -view spec -scan_data_in test_si1
```

## Sharing Scan-In Pins

If you want to specify the number of chains shared per scan-in port, use the `-shared_scan_in` option of the `set_scan_configuration` command.

```
dc_shell> set_scan_configuration -shared_scan_in 7
```

## Sharing a Scan Input With a Functional Port

By default, DFT Compiler always generates a dedicated scan input port. To share a scan input port with an existing functional port, use the `set_dft_signal` command.

```
dc_shell> set_dft_signal -view spec -port fnc_in \
    -type ScanDataIn
```

If you select a bidirectional port as the scan input port, DFT Compiler automatically inserts the necessary bidirectional control logic.

## Sharing a Scan Output With a Functional Port

If a scan chain contains a scannable sequential cell whose output directly drives a port in the current design, DFT Compiler automatically uses this cell as the last cell in the scan chain. If multiple scannable sequential cells directly drive output ports, DFT Compiler uses the cell that is last in alphanumeric order. DFT Compiler disrupts alphanumeric ordering to share the scan output port only on the current design. If you use the top-down design flow on a hierarchical design, DFT Compiler places each subdesign scan chain in alphanumeric order. Use the `preview_dft` command to see if a cell has been moved to the end of the scan chain to prevent a dedicated scan output port.

To select the functional port to be used as a scan output port, use the `set_dft_signal` command.

```
dc_shell> set_dft_signal -view spec -port fnc_out \
    -type ScanDataOut
```

If you select a bidirectional or three-state port as the scan output port, DFT Compiler automatically inserts the necessary control logic.

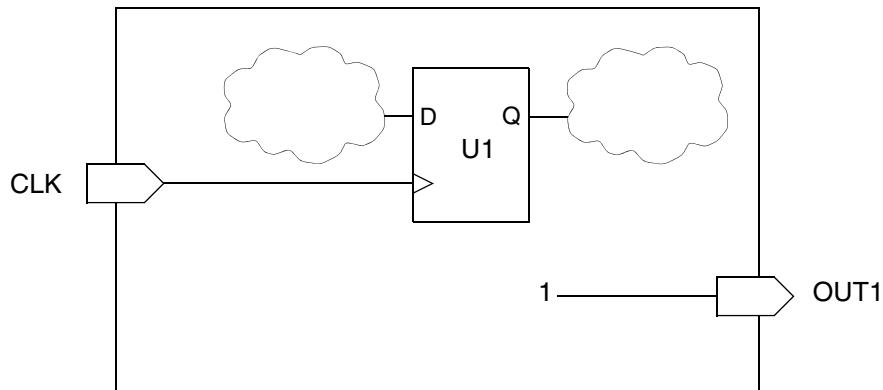
If a scannable sequential cell drives the specified output port, DFT Compiler places that cell last in the scan chain. Otherwise, DFT Compiler automatically adds the control or multiplexing logic required to share the scan output port with the functional output port.

If the port you specify as a scan-out port is tied high or to ground in functional mode, you must set the `test_mux_constant_so` variable to true. If you set this variable, DFT Compiler multiplexes the scan-out signal, using the scan-enable signal to control the multiplexer. [Figure 6-12](#) and [Figure 6-13](#) illustrate the insertion of a multiplexer during scan

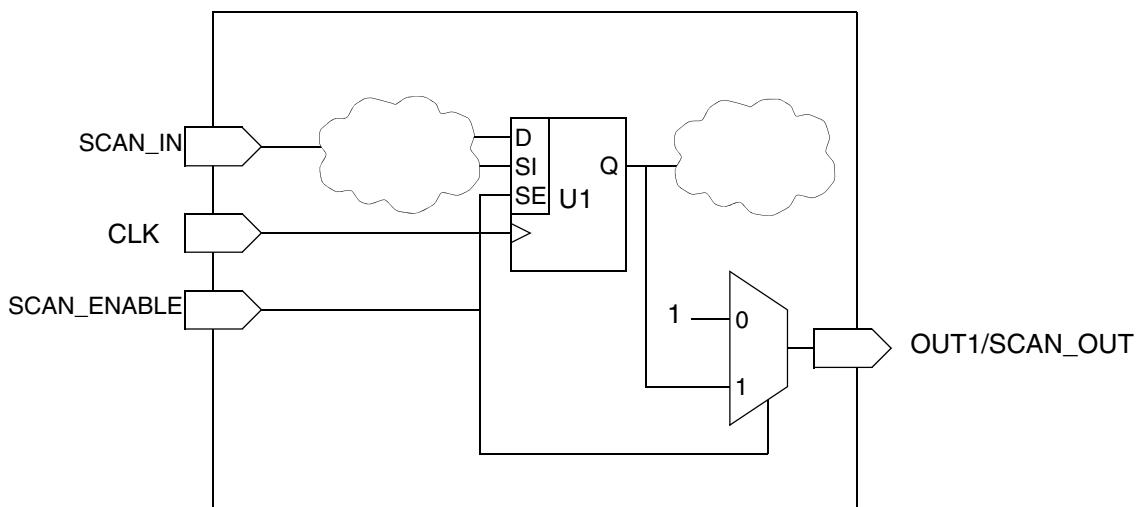
insertion. The OUT1 port is connected to a logic 1 signal before scan insertion. After scan insertion, the logic 1 signal is multiplexed with the output of the last element of the scan chain (U1).

If you do not set `test_mux_constant_so`, DFT Compiler ignores the constant value during scan insertion and uses the port as a scan output. This might change the output of the design during functional mode.

*Figure 6-12 Constant Output Value Example Before Scan Insertion*



*Figure 6-13 Constant Output Value Example After Scan Insertion*



## Associating Scan Enable Ports With Multiple Scan Chains

To associate a specific port with specific scan chains, use the `set_dft_signal` and `set_scan_path` commands.

```
dc_shell> set_dft_signal -view spec -port port_name \
           -type ScanEnable -active_state 1

dc_shell> set_scan_path {chain_names} -view spec \
           -scan_enable port_name
```

If the condition set with this command cannot be met, a warning is issued during scan preview and scan insertion.

## Using Dedicated Scan Output Ports

If your semiconductor vendor requires dedicated scan output ports or you prefer them, use the `set_scan_configuration` command to specify dedicated scan outputs:

```
dc_shell> set_scan_configuration \
           -create_dedicated_scan_out_ports true
```

The `test_dedicated_subdesign_scan_outs` variable controls the creation of dedicated scan output ports on subdesigns during hierarchical scan assembly. Leaving this variable at its default setting (`true`) enables the scan signals to be buffered separately from the functional signals at the module level, thereby reducing the impact of scan loading on the functional signal path. It also enables better placement of scan-out lockup latches by `insert_dft` during bottom-up scan insertion.

However, the addition of scan-out buffers between hierarchical levels can cause an increase in the gate count, and the creation of dedicated scan-out ports on subdesigns can cause more unqualified cells to be instantiated in the subdesigns. When these effects outweigh the benefits of having dedicated scan-out ports, set the `test_dedicated_subdesign_scan_outs` variable to `false`:

```
dc_shell> set test_dedicated_subdesign_scan_outs false
```

You cannot specify both dedicated subdesign scan-out ports and hierarchical scan chain isolation. If you specify both, DFT Compiler generates the following message during scan assembly:

Warning: Ignoring the request for dedicated subdesign scan-out ports. Hierarchical isolation was also requested.  
(UIT-47)

---

## Suppressing Replacement of Sequential Cells

Use the `set_scan_element` command to determine whether specific sequential cells are to be replaced by scan cells that become part of the scan path during `insert_dft`.

For full-scan designs, `insert_dft` replaces all nonviolated sequential cells with equivalent scan cells by default. Therefore, you do not need to set the `scan_element` attribute unless you want to suppress replacement of sequential cells with scan cells. To prevent such replacement for certain cells, set the `scan_element` attribute to `false` for those cells.

Note:

If you want to specify which scan cells are to be used for scan replacement, use the `set_scan_register_type` command.

You should not use the `set_scan_element true` command if you use the `compile -scan` command to replace elements.

## In Logical Scan Synthesis

In logical scan synthesis, the `set_scan_element false` command unscans the cell on a design in which scan replacement has already occurred.

---

## Changing the Scan State of a Design

In certain circumstances, you might find it necessary to manually set the scan state of a design. Use the `set_scan_state` command to do so. The `set_scan_state` command has three options: `unknown`, `test_ready`, and `scan_existing`.

If there are nonscan elements in the design, use `set_scan_element false` to properly identify them.

You can check the test state of the design by using the `report_scan_state` command.

One situation in which you would set the scan state is if you needed to write a netlist of a test-ready design and read it into a third-party tool. After making modifications, you can bring the design back into DFT Compiler as shown in [Example 6-5](#).

### *Example 6-5 Changing the Scan State of a Design*

```
dc_shell> read_file -format verilog my_design.v
dc_shell> report_scan_state
*****
Report : test
-state
Design : MY_DESIGN
Version: 2002.05
Date   : Wed Jul 25 18:12:39 2001
*****
Scan state          : unknown scan state
1
dc_shell> set_scan_state test_ready
Accepted scan state.
1
dc_shell> report_scan_state
*****
Report : test
-state
Design : MY_DESIGN
Version: 2002.05
Date   : Wed Jul 25 18:14:47 2001
*****
Scan state          : scan cells replaced with loops
```

#### **Important:**

You do not need to set the scan state if you are following the recommended design flow.

---

## **Removing Scan Specifications**

The `remove_scan_specification` command removes scan specifications from the current design. Note that this command deletes only those specifications you defined with the `set_scan_configuration` command.

Specifications defined using other commands are removed by issuing the corresponding remove command. For example, you use `remove_scan_path` to remove the path specifications you defined with the `set_scan_path` command.

Note that `remove_scan_specification` does not change your design. It merely deletes specifications you have made.

You can use the `remove_scan_specification` command to remove explicit specifications of synthesizable segments. When you remove an explicit specification, the multibit component inherits the current implicit specification.

Note:

The `remove_scan_specification` command does not affect the settings made with the `set_scan_register_type` command. These settings can be removed by use of the `remove_scan_register_type` command.

---

## Keeping Specifications Consistent

The set of user specifications contributing to the definition of the scan design must be consistent. User-supplied specification commands forming part of a consistent specification have the following characteristics:

- Each specification command is self-consistent. It cannot contain mutually exclusive requirements. For example, a command specifying the routing order of a scan chain cannot specify the same element in more than one place in the chain.
- All specification commands are mutually consistent. Two specification commands must not impose mutually exclusive conditions on the scan design. For example, two specification commands that place the same element in two different scan chains are mutually incompatible.
- All specification commands yield a functional scan design. You cannot impose a specification that leads to a nonfunctional scan design. For example, a specification that mandates fewer scan chains than the number of incompatible clock domains is not permitted.

The number of clock domains in your design, together with your clock-mixing specification, determines the minimum number of scan chains in your design. If you specify an exact number of scan chains smaller than this minimum, the `insert_dft` command issues a warning message and implements the minimum number of scan chains.

---

## Synthesizing Three-State Disabling Logic

DFT Compiler can, by default, handle three-state nets. It does so with the following functionality:

- By default, it distinguishes between internal and external three-state nets.
- By default, it prevents bus contention by causing only one three-state driver to be active at one time.

- By default, it modifies internal three-state nets in bottom-up design methodology to make exactly one three-state driver active.

To prevent bus contention or bus float, internal three-state nets in your design must have a single active driver during scan shift. DFT Compiler automatically performs this task.

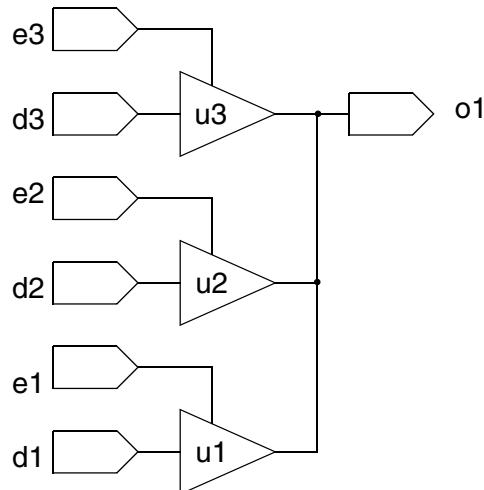
DFT Compiler determines if the internal three-state nets in your design meet this requirement.

By default, DFT Compiler adds disabling logic to internal three-state nets that do not meet this requirement. The scan-enable signal controls the disabling logic and forces a single driver to be active on the net throughout scan shift.

In some cases, DFT Compiler adds redundant disabling logic because the disabling logic checks for internal three-state nets are limited.

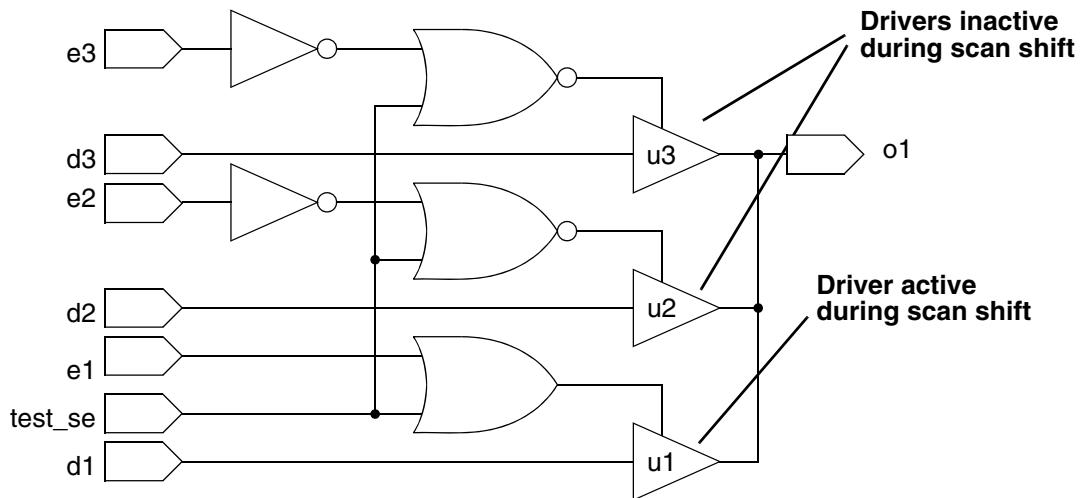
[Figure 6-14](#) shows the simple internal three-state net used as an example throughout this section.

*Figure 6-14 Internal Three-State Net Example*



[Figure 6-15](#) shows the disabling logic added by DFT Compiler during the `insert_dft` process.

Figure 6-15 Three-State Output With Disabling Logic



If the design already contains logic that prevents or can be configured to prevent the occurrence of bus contention and bus float during scan shift, you can use the `set_dft_configuration` command to elect not to have the disabling logic added by DFT Compiler.

```
dc_shell> set_dft_configuration -fix_bus disable
```

During scan shift, DFT Compiler does not check for bus contention or bus float conditions. If you do not add the three-state disabling logic, verify that no invalid conditions occur during scan shift.

If you want to perform bottom-up scan insertion, you must choose a strategy for handling the insertion of three-state enabling and disabling logic. If you use the `set_dft_configuration -fix_bus disable` command, your design will be free of bus float or bus contention during scan shift. However, during bottom-up scan insertion, the `insert_dft` command might be forced to modify modules that it has already processed.

This strategy is easy to implement in scripts but can result in repeated modifications to subblocks. Note that DFT Compiler does recognize three-state enabling and disabling logic that it has previously inserted in a submodule and so does not insert unnecessary or redundant enabling and disabling logic.

For example, consider a top-level design with two instances of a module of type `sub_type_1`. Both of these instances drive a three-state bus that, in turn, drives inputs on another module. If you perform scan insertion with default settings on the design `sub_type_1`, then in the top design, the three-state ports that drive this common bus will be turned off in scan shift, thus

creating a float condition. In other words, when you run `insert_dft` at the top level with default options selected, the `insert_dft` command modifies one of the two instances of `sub_type_1`. As a result, each net within the bus has a single enabled driver during scan shift.

You can consider two other, nondefault, strategies when you want to use bottom-up scan insertion.

You can synthesize three-state disabling logic at the top level only. Synthesis of disabling logic at the top level guarantees a consistent implementation across all subdesigns. Use the `set_dft_configuration -fix_bus disable` command to disable synthesis of three-state disabling logic in subdesigns.

```
/* subdesign command sequence */
dc_shell> current_design subdesign
dc_shell> set_dft_configuration -fix_bus disable
...
dc_shell> insert_dft

/* top-level command sequence */
dc_shell> current_design top
dc_shell> set_dft_configuration -fix_bus enable
...
dc_shell> insert_dft
```

A third option is to use the `preview_dft -show tristates` command before you run `set_dft_configuration` on each submodule to determine what enabling and disabling logic should be inserted on the external three-state nets for each module. This strategy is the most complex to use, and your scripts need to be specific to each design. However, if you implement this method correctly, you can assemble submodules into a complete testable design without further modification of a submodule by `set_dft_configuration`.

---

## Configuring Three-State Buses

The `set_dft_configuration` command can configure three-state buses according to settings applied by the `set_ autofix_configuration` command.

If the `-fix_bus` option of the `set_scan_configuration` command is set to `disable`, no changes to the three-state driver logic are made, regardless of any other three-state settings.

## Configuring External Three-State Buses

On external three-state nets, the `-type external_bus` option of the `set_ autofix_configuration` command controls three-state disabling behavior. If you want to make no changes to the external three-state nets, use the

`-method no_disabling` option. If you want to allow exactly one three-state driver to be enabled on each external three-state net, you can use the `enable_one` option. If you want to ensure that all external three-state nets are disabled, use the `disable_all` option, which is the default behavior for the `external_tristates` setting.

You might have multiple modules that are stitched together at the top level, and you might want to be sure that one of those modules contains the active three-state drivers while the other modules are all off. You can do that by using a bottom-up scan insertion methodology and by setting the `set_scan_configuration -external_tristates` command appropriately for each module before you run `insert_dft` on that module.

## Configuring Internal Three-State Buses

The same rules apply for internal three-state nets as for external three-state nets. If you allow all your subdesigns to be set to the default behavior, `insert_dft` can choose a three-state driver on the net to make active and can disable all others.

## Overriding Global Three-State Bus Configuration Settings

You can override these internal and external three-state net settings by using the `set_ autofix_element` command, which can be applied to individual nets in your design.

This command applies only to the nets and not to individual three-state drivers.

You might have a situation in which multiple instances of the same design must have separate three-state configuration settings. You can achieve this by unifying the particular instances and then using the `set_ autofix_element` command to define the type of enabling or disabling logic you want to see applied on that instance.

## Disabling Three-State Buses and Bidirectional Ports

There are several different methods you can use to disable logic to ensure that three-state buses and bidirectional ports are properly configured during scan shift:

- To set the default behavior for top-level three-state specifications, use the following command:

```
set_dft_configuration -fix_bus enable | disable
```

- To set the default behavior for top-level bidirectional port specifications, use the following command:

```
set_dft_configuration \
    -fix_bidirectional enable | disable
```

- To set global three-state specifications, use the following command:

```
set_ autofix_configuration \
```

```
-type internal_bus | external_bus \
-method disable_all | enable_one |no_disabling
```

- To set global bidirectional port specifications, use the following command:

```
set_ autofix_configuration -type bidirectional \
-method input | output | no_disabling
```

- To set local three-state specifications with `-include_element` and `-exclude_element`:

```
set_ autofix_element -type internal_bus | external_bus \
-method input | output | no_disabling
```

- To set local bidirectional port specifications with `-include_element` and `-exclude_element`:

```
set_ autofix_element -type bidirectional \
-method input | output | no_disabling
```

---

## Handling Bidirectional Ports

Every semiconductor vendor has specific requirements regarding the treatment of bidirectional ports during scan shift. Some vendors require that bidirectional ports be held in input mode during scan shift, some require that bidirectional ports be held in output mode during scan shift, and some have no preference. DFT Compiler provides the ability to set the bidirectional mode both globally and individually.

Before you insert control logic for bidirectional ports, understand your vendor's requirements for these cells during scan shift.

If the `-fix_bidirectional disable` option of the `set_dft_configuration` command is set, no disabling logic is added to any bidirectional ports, regardless of any other bidirectional port settings.

## Setting Individual Bidirectional Port Behavior

To specify bidirectional behavior on individual ports, use the `set_ autofix_element` command.

Use the `reset_ autofix_element` command to remove all `set_ autofix_element` specifications for the current design.

Use `preview_dft` to see the bidirectional port conditioning that will be implemented for each bidirectional port in a design.

## **Fixed Direction Bidirectional Ports**

Bidirectional ports that have enables connected to constant values and that are therefore always configured in either input mode or output mode are referred to as degenerated bidirectional ports. DFT Compiler does not add control logic for degenerated bidirectional ports.

DFT Compiler recognizes constant values on the enable pins of bidirectional ports for the following cases:

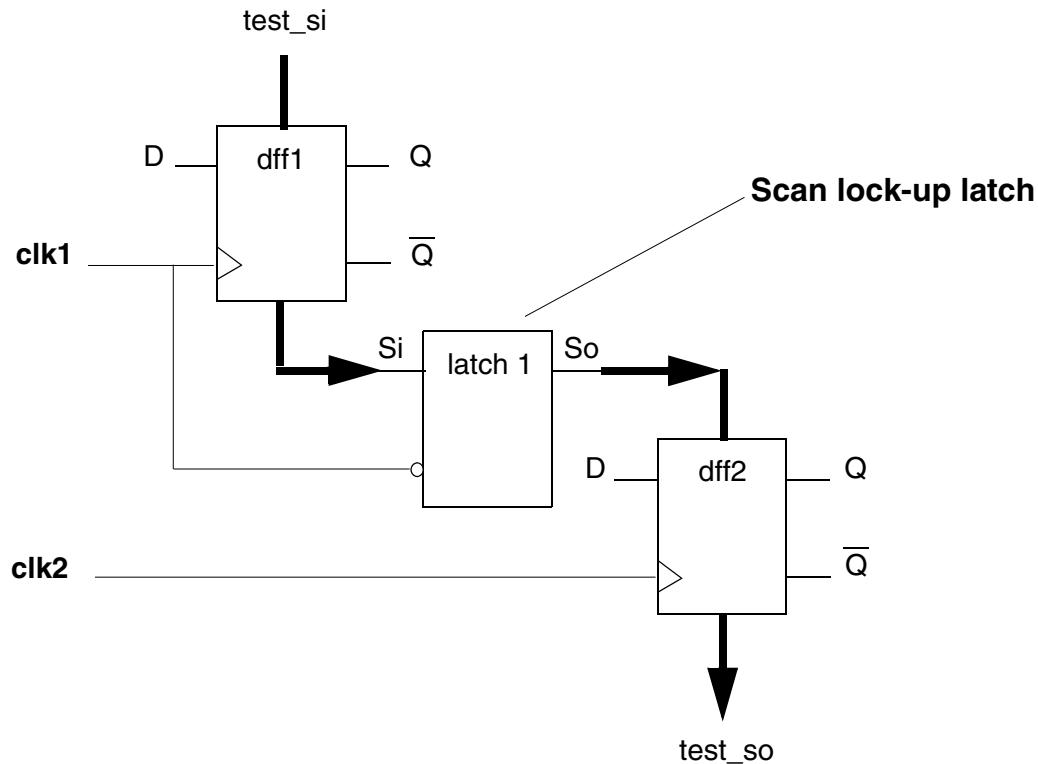
- Enable forced to a constant value by a tie-off cell in the circuit
- Enable forced to a constant value by a `set_dft_signal` command

---

## **Using Scan Lock-Up Elements**

A scan lock-up element is a retiming sequential cell on a scan path that is clocked by the inversion of the previous scan cell's clock. [Figure 6-16](#) shows a lock-up latch.

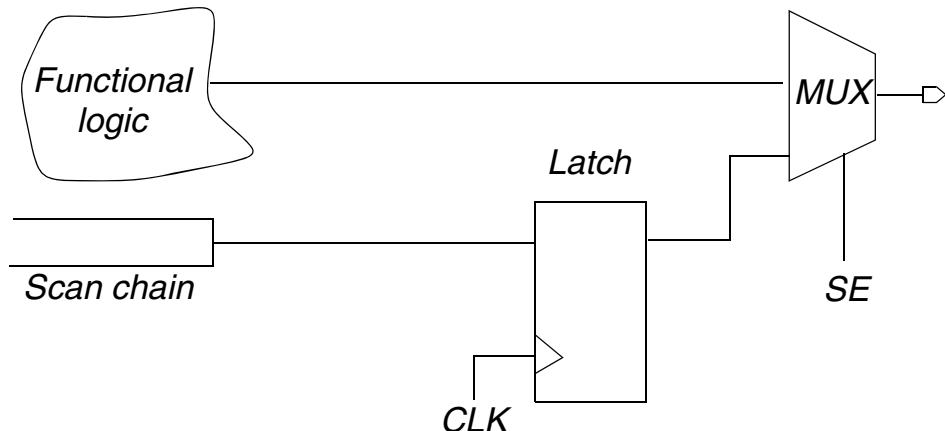
Figure 6-16 Scan Lock-Up Latch



Use scan lock-up elements in multiplexed flip-flop designs whenever adjacent scan cells have different clocks or different branches of a gated clock. For successful synchronization, the falling edge of the current scan cell must occur after or concurrent with the rising edge of the next scan cell.

By default, DFT Compiler adds scan lock-up elements to your multiplexed flip-flop scan chain whenever adjacent scan cells are clocked by different clocks. If `set_scan_configuration -insert_terminal_lockup true` is specified, it also inserts end-of-chain lock-up elements, as shown in [Figure 6-17](#). End-of-chain lock-up elements are shown in `preview_dft` reports. The `-insert_terminal_lockup` option is set to false by default.

Figure 6-17 End-of-Chain Lock-Up Latch



You can select the type of synchronization element used in the scan chain with the following option of the `set_scan_configuration` command:

```
-lockup_type latch|flip_flop
```

The default lock-up type is a level-sensitive latch. If you choose `flip_flop` as the lock-up type, an edge-triggered flip-flop is used as the synchronization element. In either case, the cell is not constrained from the target library used for mapping the synchronization element. You may use the `-lockup_type` option in conjunction with either of the following commands:

```
dc_shell> set_scan_configuration -add_lockup true \
           -lockup_type latch|flip_flop

dc_shell> set_scan_configuration \
           -insert_terminal_lockup true \
           -lockup_type latch|flip_flop
```

You can disable insertion of scan lock-up elements in multiplexed flip-flop designs by using the `set_scan_configuration -add_lockup false` command. If you execute this command after specifying a scan style other than multiplexed flip-flop, DFT Compiler generates the following message:

```
Warning: Scan style '%s' does not add lockup latches to
the scan chain. Ignoring the set_scan_configuration
-add_lockup command. (UIT-227)
```

**Note:**

DFT Compiler does not add scan lock-up elements for clock-edge changes or for scan styles other than multiplexed flip-flop.

When adding scan lock-up elements, DFT Compiler adds the scan lock-up element to the module containing the last scan cell before the clock change.

Regardless of your selected scan style, you can explicitly add scan lock-up elements to your scan chain by using the `set_scan_path` command.

Use the `preview_dft -show cells` command to see where `set_dft_configuration` plans to insert scan lock-up elements in your scan chain.

---

## Assigning Test Port Attributes

If you always save and read mapped designs in the .ddc format, you usually do not need to explicitly set `signal_type` attributes. If you do not save your design in .ddc format, you must use the `set_dft_signal` command.

Note:

Use the `set_dft_signal` command for scan-inserted, existing-scan, and test-ready designs.

When `insert_dft` sets attributes on test ports, for all scan styles, it creates the following values:

- It places either a `test_scan_enable` or a `test_scan_enable_inverted` attribute on scan-enable ports. The `test_scan_enable` attribute causes a logic 1 to be applied to the port for scan shift. The `test_scan_enable_inverted` attribute causes a logic 0 to be applied to the port for scan shift.
- Scan input ports are identified with the `test_scan_in` attribute.
- Scan output ports are identified with the `test_scan_out` or `test_scan_out_inverted` attribute.

Note that some scan styles require test clock ports on the scan cell.

---

## Architecting Test Clocks

When DFT Compiler creates a test protocol, it uses default values for the clock timing, based on the clock type, unless you explicitly specify clock timing.

This section shows you how to set test clocks and handle multiple clock designs. It include the following subsections:

- [Setting Test Clocks](#)
- [Handling Multiple Clock Designs](#)

---

## Setting Test Clocks

The default timing used by the `create_test_protocol` command is shown in the following table. This timing is used unless you change it by using the `set_dft_signal` command.

Clock type	First edge (ns)	Second edge (ns)
Edge-triggered	45.0	55.0
Master clock	30.0	40.0
Slave clock	60.0	70.0

## Specifying Clock Timing Attributes

You can explicitly specify the clock timing by using the `set_dft_signal` command. This command sets the following timing attributes on the clock ports you specify:

- `MasterClock`
- `ScanMasterClock`

To verify the values of these timing attributes for all clock ports, use the `report_dft_signal` command.

Note:

The `set_dft_signal` command has a time period associated with it. That period has to be identical to the `test_default_period` value. If you change the value of one, you must be sure to check the value of the other.

## The Waveform Section

The arguments to `set_dft_signal` are the same as the values specified in the statements that make up the STIL waveform section. The `rise` argument becomes the value of the `rise` argument in the waveform statement in the test protocol clock group. The `fall` argument becomes the value of the `fall` argument in the waveform statement in the test protocol clock group.

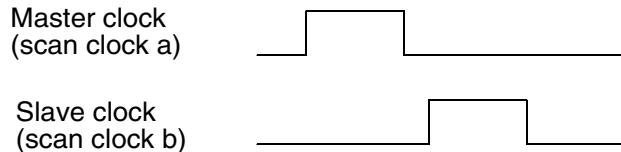
## Specifying the Clock Period

If you use the `set_dft_signal` command, the default value is the period value defined by the `test_default_period` variable.

## Multiplexed Flip-Flop Design Example

The clock timing for the multiplexed flip-flop design example shown in [Figure 6-18](#) is a positive pulse clock with a period of 100.0 ns. The rising edge occurs at 45.0 ns and the falling edge occurs at 55.0 ns. This clock waveform is shown in [Figure 6-18](#).

*Figure 6-18 Default Clock Timing for Multiplexed Flip-Flop Example*



You can explicitly specify this clock waveform by using the following `set_dft_signal` command:

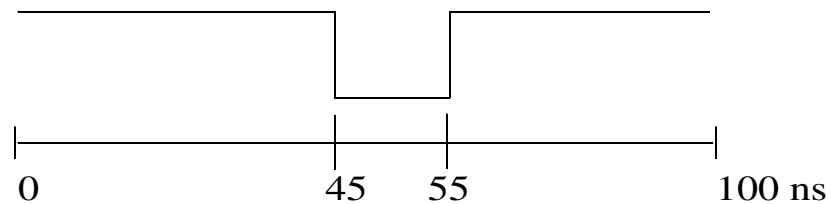
```
dc_shell> set_dft_signal -view existing_dft \
                     -type ScanClock -timing [list 45.0 55.0] \
                     -port CLK
```

If a return-to-1 clock is required instead of the default clock, you can use the following `set_dft_signal` command:

```
dc_shell> set_dft_signal -view existing_dft \
                     -type ScanClock -timing [list 55.0 45.0] \
                     -port CLK
```

[Figure 6-19](#) shows the waveform diagram.

*Figure 6-19 Return-to-1 Waveform Diagram*



---

## Handling Multiple Clock Designs

Having multiple system clocks necessitates giving special attention to the scan architecture in multiplexed flip-flop designs. Because multiplexed flip-flop designs use the system clock for scan shift, the ordering of cells within a scan chain can affect the scan shift operation. The easiest way to prevent clock skew problems during scan shift is to allocate cells to scan chains by clock domain.

By default, DFT Compiler creates a scan chain for each clock domain in the design. You can mix different clock edges or different clocks (or both) within a scan chain by using the `set_scan_configuration -clock_mixing` command. This command applies only to the multiplexed flip-flop scan style. If you execute this command after specifying a scan style other than multiplexed flip-flop, DFT Compiler generates the following message:

```
Warning: Scan style '%s' does not need scan clock domain  
constraints. Ignoring the set_scan_configuration  
-clock_mixing command. (UIT-229)
```

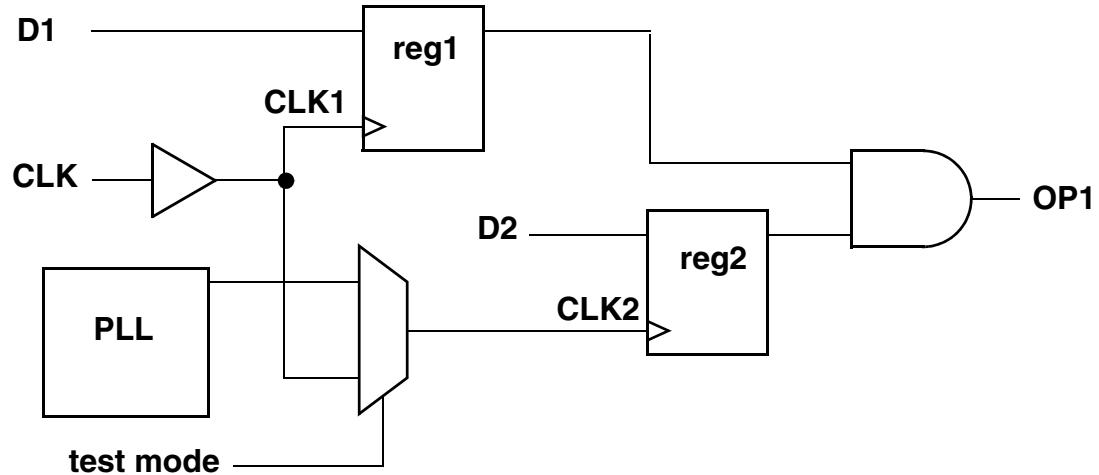
## Internal Clocks

The `preview_dft` command does not see internal clocks created by test points, but `insert_dft` does.

For the purpose of building scan chains, the `insert_dft` command, by default, treats all internal clock signals driven by the same top-level port as the same clock signal.

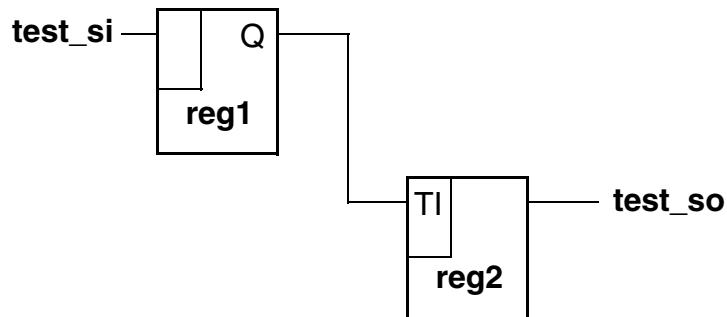
For example, the `insert_dft` command treats the internal clocks CLK1 and CLK2 in [Figure 6-20](#) as the same top-level clock, CLK.

Figure 6-20 Circuit With Same Top-Level Clock Driving Internal Clock Signals



By default, the `insert_dft` command creates a single scan chain, as shown in Figure 6-21.

Figure 6-21 Default Scan Chain



**Note: The rest of the circuit is unchanged.**

Note that the multiplexer on CLK2 delays the arrival of the clock at reg2. This can cause a hold-time violation unless you enable hold-time violation fixing by using the `set_fix_hold` command. If you enable hold-time violation fixing (`set_fix_hold CLK`) before building scan chains, the `insert_dft` command adds buffers to the scan connection between reg1/Q and reg2/TI.

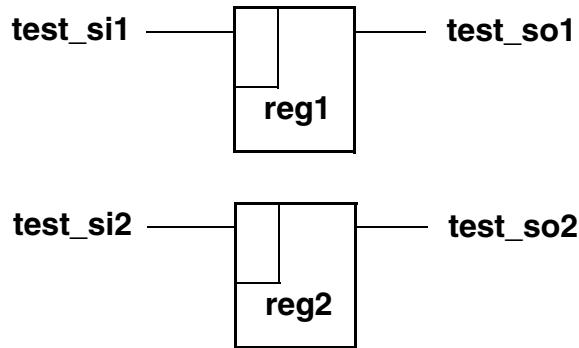
You can avoid creating the hold-time violation by assigning the two flip-flops to separate scan chains. To do this, use the following command:

```
dc_shell> set_scan_configuration \
           -internal_clocks multi
```

This command instructs the `insert_dft` command to treat internal clocks driven by multiplexers or other multiple input gates as separate clocks.

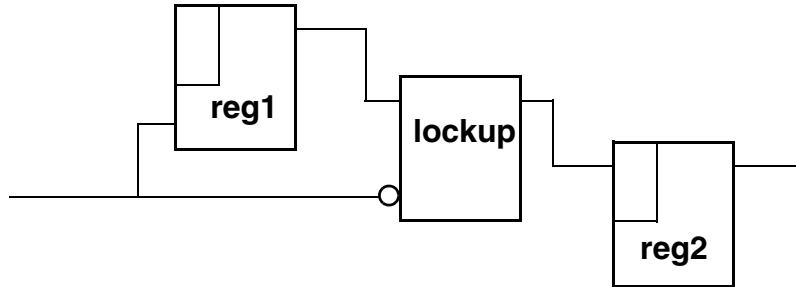
If you use the default setting of no mixed clocks (`set_scan_configuration -clock_mixing no_mix`) or use a setting of mixed edges (`set_scan_configuration -clock_mixing mix_edges`), the `insert_dft` command places `reg1` and `reg2` on separate scan chains as shown in [Figure 6-22](#).

*Figure 6-22 Default Scan Chain With `set_scan_configuration -internal_clocks multi`*



If you use mixed clocks (`set_scan_configuration -clock_mixing mix_clocks` or `mix_clocks_not_edges`), the `insert_dft` command creates a single scan chain with a lock-up latch between `reg1` and `reg2`, as shown in [Figure 6-23](#).

Figure 6-23 Scan Chain With `set_scan_configuration -clock_mixing mix_clocks`



**Note: The rest of the circuit is unchanged.**

You can enable the same behavior on just one clock line by using the `-internal_clocks` option of the `set_dft_signal` command. Using the circuit shown in [Figure 6-20](#), the following command tells the `insert_dft` command to treat only internal clocks driven by the CLK clock line as separate clocks:

```
dc_shell> set_dft_signal -view existing_dft \
           -type ScanClock -timing [list 45 55] \
           -internal_clocks multi -port CLK
```

If you set opposing values by using the `set_scan_configuration` command and the `set_dft_signal` command, then the value from the `set_dft_signal` command takes precedence. For example, assume that you set the following opposing values on the circuit in [Figure 6-20](#) by using these two commands:

```
dc_shell> set_scan_configuration -internal_clocks none
dc_shell> set_dft_signal -view existing_dft \
           -type ScanClock -timing [list 45 55] \
           -internal_clocks multi -port CLK
```

Because the value set by the `set_dft_signal` command takes precedence, signals driven by CLK via multiplexers or other multiple-input gates are treated as separate clocks. All other clocks in the design are treated according to the default configuration. Therefore, CLK and CLK2, shown in [Figure 6-20](#), are treated as different clocks.

**Note:**

The `-internal_clocks` option affects only scan chain building for the multiplexed flip-flop scan style.

## Assigning Scan Chains to Specific Clocks

You might want to allocate specific clocks to given scan chains. By doing this you gain additional control over scan chain specifications. To associate a scan chain to a clock domain, use the `set_scan_path` command.

```
dc_shell> set_scan_path chain_name \
           -view spec -scan_master_clock clock_name
```

If you also use the `-exact_length` option to define the number of scan cells to be included in that scan chain, DFT Compiler will build the scan chains if it can meet clock mixing requirements and ensure that the scan chains are valid.

If the specifications in the `set_scan_path` command cannot be met, they are not applied.

## Requirements for Valid Scan Chain Ordering

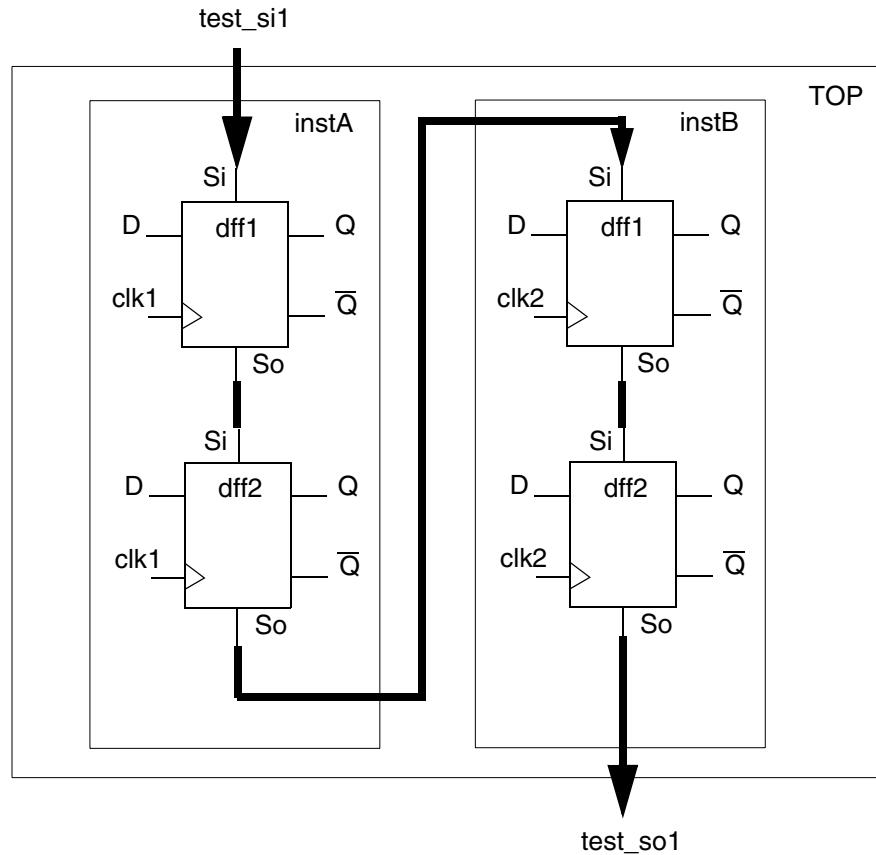
DFT Compiler generates valid mixed-clock scan chains based on the specified or inferred test clock timing. The scan cells in a valid scan chain are ordered so the cells clocked later in the cycle appear earlier in the scan chain. This guarantees that all cells in the scan chain get the expected data during scan shift.

DFT Compiler uses ideal clock timing to determine valid scan chain ordering. Because DFT Compiler does not consider clock skew when ordering scan chains, mixed-clock scan chains might not shift properly. However, if you set the `fix_hold` attribute on the clock nets, DFT Compiler fixes hold-time violations during scan assembly. Always perform full-timing logic simulation on mixed-clock scan chains.

Although DFT Compiler guarantees correct shift function under ideal clock timing, it cannot guarantee that capture problems will not occur. Capture problems are caused by your functional logic; modify your design to correct capture problems. For more information, see [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

[Figure 6-24](#) shows a design with a mixed-clock scan chain. DFT Compiler created this scan chain with default timing (rising edge at 45 ns, falling edge at 55 ns) for both clk1 and clk2. The validity of the scan chain depends on the clock timing. You can make the scan chain invalid if you change the clock timing.

Figure 6-24 Mixed-Clock Scan Chain



For example, the scan chain in Figure 6-24 is

- A valid scan chain (the instA cells are clocked after the instB cells) if you define the test clock timing as

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 55 65] \
    -port clk1

dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port clk2
```

- An invalid scan chain (the instA cells are clocked before the instB cells) if you define the test clock timing as

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port clk1
```

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 55 65] \
    -port clk2
```

To maintain the validity of your scan chains, do not change the test clock timing after assembling the scan structures.

By default, when you request mixing of clocks within a multiplexed flip-flop scan chain, DFT Compiler inserts lock-up latches to prevent timing problems.

## Using Multiple Master Clocks in LSSD Designs

In level-sensitive scan designs (LSSD), you need not allocate scan chains by clock for timing purposes; however, you might want to do so. Assume that you have a latch-based design with two system enables, en1 and en2, and you want a scan chain allocated for each enable.

The command sequence given in [Example 6-6](#) accomplishes this:

*Example 6-6 Command Sequence for Multiple Master Clocks in LSSD*

```
dc_shell> set_scan_configuration -style lssd
dc_shell> dft_drc

/* create test A clock ports and assign to scan chains */

dc_shell> create_port -direction in {A_CLK1 A_CLK2}
dc_shell> set_dft_signal -view spec -port A_CLK1 \
    -type ScanMasterClock
dc_shell> set_scan_path 1 -view spec \
    -scan_master_clock A_CLK1
dc_shell> set_dft_signal -view spec -port A_CLK2 \
    -type ScanMasterClock
dc_shell> set_scan_path 2 -view spec \
    -scan_master_clock A_CLK2

/* explicitly allocate cells to scan chains by system enable */

dc_shell> create_clock en1 -name cclk1 -period 100
dc_shell> set_cclk1_cells [all_registers -clock cclk1]
dc_shell> set_scan_path 1 -include [get_objectname $cclk1_cells]
dc_shell> create_clock en2 -name cclk2 -period 100
dc_shell> set_cclk2_cells [all_registers -clock cclk2]
dc_shell> set_scan_path 2 -include [get_objectname $cclk2_cells]

/* preview scan configuration and implement */

dc_shell> preview_dft -show all
dc_shell> insert_dft
```

## Dedicated Test Clocks for Each Clock Domain

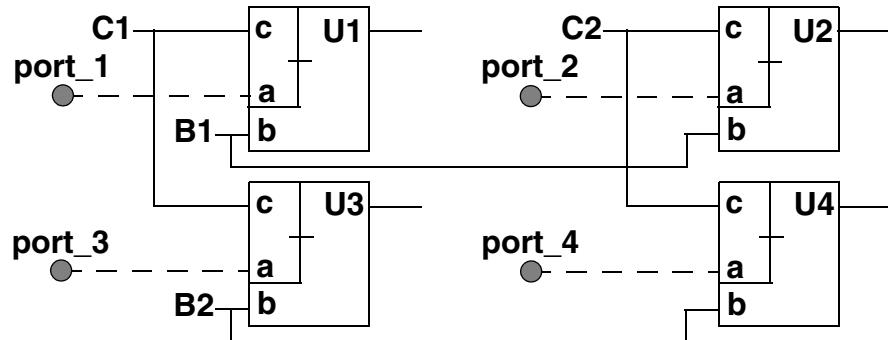
The `insert_dft` command creates clocks that are used only for test purposes when it routes scan chains by using the following scan styles:

- LSSD
- Clocked LSSD

The test clocks are dedicated for each system clock domain. This makes clock trees and clock signal routing easier. The `insert_dft` command uses the following guidelines to determine how test clocks are added:

- For sequential cells with multiple test clocks, the `insert_dft` command adds a test clock for each unique set of system clocks. For example, in [Figure 6-25](#), cell U1 is clocked by C1 (master) and B1 (slave), cell U2 is clocked by C2 and B1, cell U3 is clocked by C1 and B2, and cell U4 is clocked by C2 and B2. The `insert_dft` command adds four test clocks, one for each unique set.

*Figure 6-25 Adding Test Clocks for Sequential Cells With Multiple Test Clocks*



For cells that are clocked by the same system clock, the `insert_dft` command adds the same test clock to these cells, even though they are clocked by different clock senses (rising edge, falling edge, active low, and active high). When a clock is distributed to pins with mixed clock senses, the `insert_dft` command inserts inverters to ensure design functionality.

## Controlling LSSD Slave Clock Routing

For designs using either LSSD scan style or clocked LSSD scan style, all single-latch and flip-flop elements have an unconnected slave clock pin after scan replacement.

If possible, the `insert_dft` command uses the slave clocks distributed to double-latch elements and does either of the following:

- Creates, at most, one new port per design when you want to use only the slave clocks distributed to the double-latch elements

- Creates one or more ports when you want test clocks created according to different system clocks

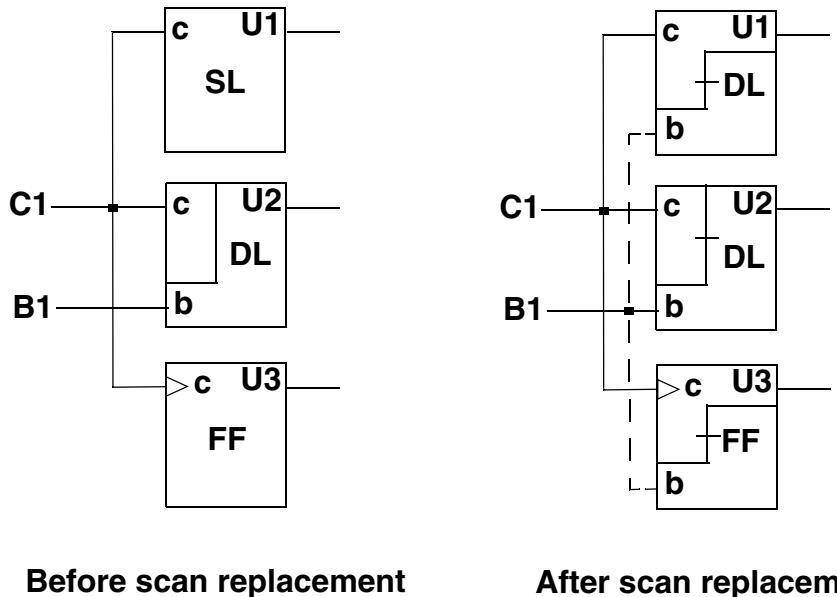
The `insert_dft` command uses the following guidelines when connecting slave clock pins of single-latch and flip-flop elements after scan replacement:

- Connect the unconnected slave clock pin of LSSD scan style single-latch or flip-flop elements to the slave clock pin of the double-latch that is clocked by the same system clock. See [Figure 6-26](#).

Note:

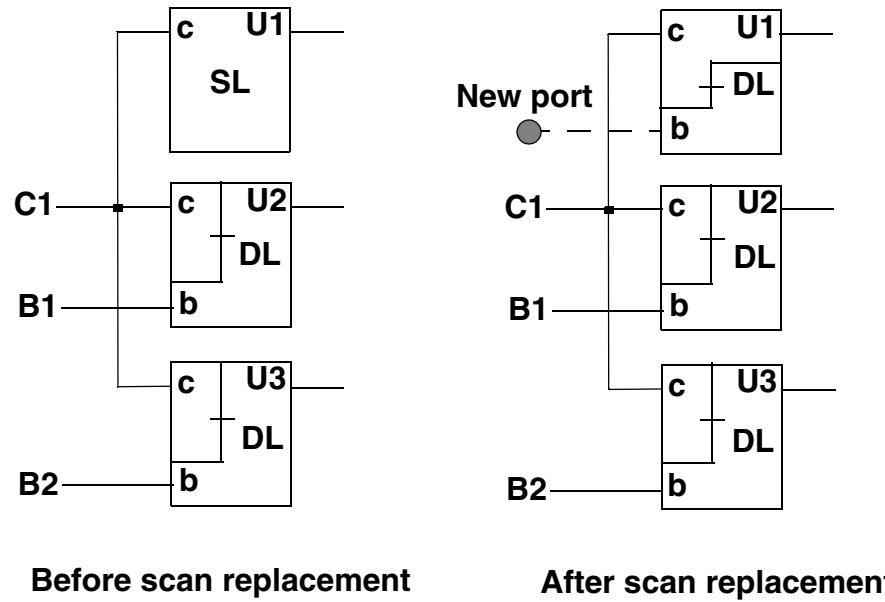
For clarity, the A clock is omitted in [Figure 6-26](#) through [Figure 6-29](#) after scan replacement.

*Figure 6-26 Single-Latch and Double-Latch Cells With the Same System Clock*



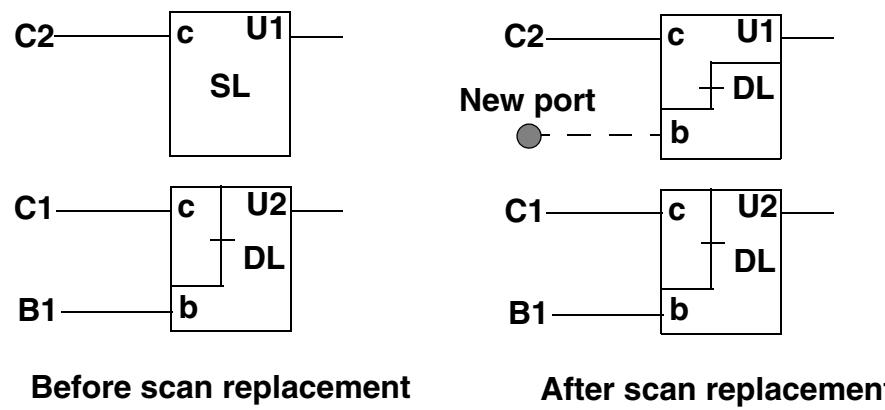
- Connect to a new slave clock, creating a new one if necessary, if a system clock drives multiple cells with different slave clocks. See [Figure 6-27](#).

Figure 6-27 Single-Latch and Double-Latch Cells Clocked by the Same System Clock



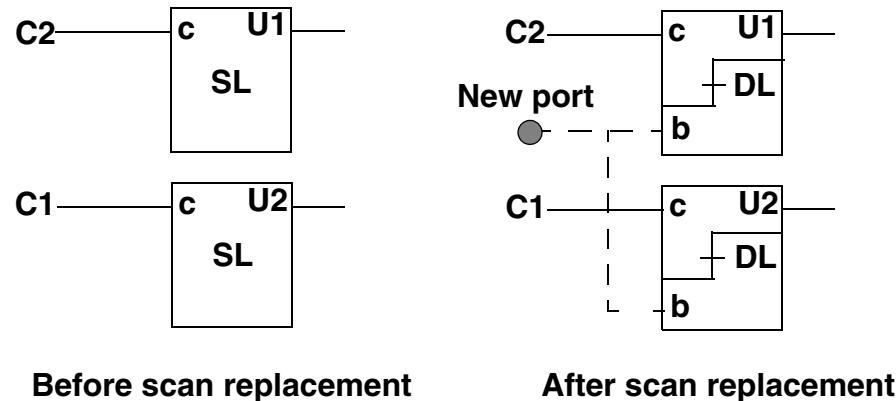
- Connect to a new slave clock port, creating one if necessary, if double-latch cells are driven by different clocks. See [Figure 6-28](#).

Figure 6-28 Single-Latch and Double-Latch Cells Clocked by Separate System Clocks



- Connect to a new slave clock, creating a new port if necessary, if there are no double-latch cells. See [Figure 6-29](#).

*Figure 6-29 Connecting Slave Clock Pin: No Double-Latch Cells*



## Modifying Your Scan Architecture

Unless conflicts occur, the `set_scan_configuration` commands are additive. You can enter multiple `set_scan_configuration` commands to define your scan configuration. If a conflict occurs, the latest `set_scan_configuration` command overrides the previous configuration.

To modify your scan configuration, you can rely on the override capability or remove the complete scan configuration and start over. Use the `reset_scan_configuration` command to remove the complete scan configuration. Do not use the `reset_design` command to remove the scan configuration. Configuring the scan chain does not place attributes on the design, so `reset_design` has no effect on the scan configuration and removes all other attributes from your design, including constraints necessary for optimization.

To make minor adjustments to the scan architecture, modify the scan specification script generated by the `preview_dft -script` command.

```

dc_shell> preview_dft -script > scan_arch.tcl
/* manually modify scan_arch.tcl to reflect desired
architecture */
dc_shell> source scan_arch.tcl
dc_shell> preview_dft
dc_shell> insert_dft

```

---

## Post-Scan Test Design Rule Checking

After you perform scan insertion, you need to perform design rule checking again to ensure that no violations have been introduced into your design by the scan insertion process.

This section has the following subsections related to post-scan test DRC:

- [Preparing for Test Design Rule Checking After Scan Insertion](#)
  - [Checking for Topological Violations](#)
  - [Checking for Scan Connectivity Violations](#)
  - [Causes of Common Violations](#)
  - [Ability to Load Data Into Scan Cells](#)
  - [Ability to Capture Data Into Scan Cells](#)
- 

### Preparing for Test Design Rule Checking After Scan Insertion

Before performing test design rule checking, you need to ensure that your timing parameters are set properly. See [Chapter 5, “Pre-Scan Test Design Rule Checking,”](#) for information on timing parameters.

Check the following settings:

- Ensure that the following timing parameters are correctly set in your .synopsys\_dc.setup file:
  - test\_default\_delay
  - test\_default\_strobe
  - test\_default\_bidir\_delay
  - test\_default\_period

Note:

Consult your ASIC vendor for timing value requirements.

- If you bring your design in as a netlist and not in .ddc format, you need to reset all your set\_scan\_configuration, set\_dft\_signal attributes.
- Ensure that signal\_type attributes exist on test ports.

If you used the `insert_dft` command to build scan chains and you saved the design in .ddc format, DFT Compiler automatically establishes the signal type attributes.

If you did not use the `insert_dft` command (for example, you read in an ASCII netlist) or if you have saved the `insert_dft` design as a netlist instead of as a `.ddc` file, you must reestablish the signal type attributes by using the `set_dft_signal` command.

You can list test ports by using the `report_dft_signal` command.

- Use the `report_scan_state` command to verify that the `scan_existing` constraint is set. If the constraint is set, the report reads

Existing Scan: True

If the constraint is not set, you cannot do a scan test design rule checking run and the report reads

Existing Scan: False

---

## Checking for Topological Violations

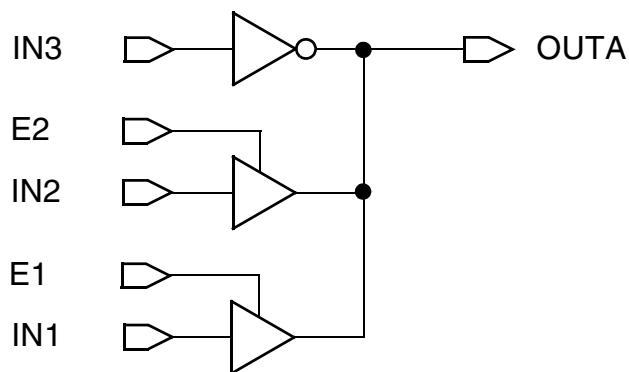
Topological checks are global connectivity checks that `dft_drc` performs in a structural manner.

If the `dft_drc` command cannot determine the logic function associated with a wired net, it issues the following message:

Warning: Type of wired net %s is unknown. (TEST-114)

The presence of a non-three-state driver on a three-state net (see [Figure 6-30](#)) results in contention on that net.

*Figure 6-30 A Non-Three-State Driver*



If the `dft_drc` command detects such a condition, it flags the violation with:

Warning: Three-state net %s is not properly driven. (TEST-115).

If the `dft_drc` command detects the presence of a pull-up driver or a pull-down driver on a non-three-state net, it flags the problem with

```
Error: Pullup/pulldown net %s has illegal driver(s).  
(TEST-331)
```

Any violation on a net forces the net to the value X for the entire protocol simulation.

---

## Checking for Scan Connectivity Violations

After the `dft_drc` command completes test protocol simulation, it analyzes the simulation results to determine the following:

- The architecture of the scan chains
- Whether the capture state and the state of the cell that is scanned are the same

The `report_scan_path` command reports the scan chain architecture determined by the `dft_drc` command.

Running an incremental compile or other command that affects the database can cause the information gathered by `dft_drc` to be invalidated. If you run a `report_scan_path` and get an error message saying that no scan path is defined, try running `dft_drc` again, immediately followed by a `report_scan_path` command.

## Scan Chain Extraction

A scan chain is a group of sequential elements through which a uniquely identifiable bit of scan data travels. The `dft_drc` command extracts scan chains from a design by tracing scan data bits through the multiple time frames of the protocol simulation. Scan chains are protocol dependent: For a given design, specifying a different test protocol can result in different scan chains. As a corollary, scan-chain-related problems can be caused by an incorrect protocol, by incorrect `set_dft_signal` specifications, or even by incorrectly specified timing data.

---

## Causes of Common Violations

During test design rule checking on scan designs, DFT Compiler simulates the test protocol to verify that the scan operation functions correctly. Protocol simulation verifies that scan cells predictably perform the following tasks:

- Receive data during scan input
- Capture data during parallel capture

- Shift data during scan output

The following sections describe the scan operation checks for each of these tasks and provide guidance in correcting the problems.

---

## Ability to Load Data Into Scan Cells

To ensure that the scan shift process can successfully load data into the scan cells, DFT Compiler verifies that

- Data arrives at the scan input pin of each scan cell
- The test clock pulse arrives at the test clock pin of each scan cell
- Scan data is not corrupted during scan shift

If a scan cell does not meet these conditions, DFT Compiler cannot control the scan cell. Typical causes for uncontrollable scan cells include

- Incorrect or incomplete test configuration
- Invalid clock logic
- Incorrect timing relationships between clocks for two-phase clocking
- Nonscan sequential cells clocked by the test clock
- Invalid scan path

DFT Compiler generates this message when it detects that it cannot shift through a scan chain:

```
Begin Scan chain violations...
```

```
Error: Chain c1 blocked at DFF gate FF_A after tracing 2
cells. (S1-1)
```

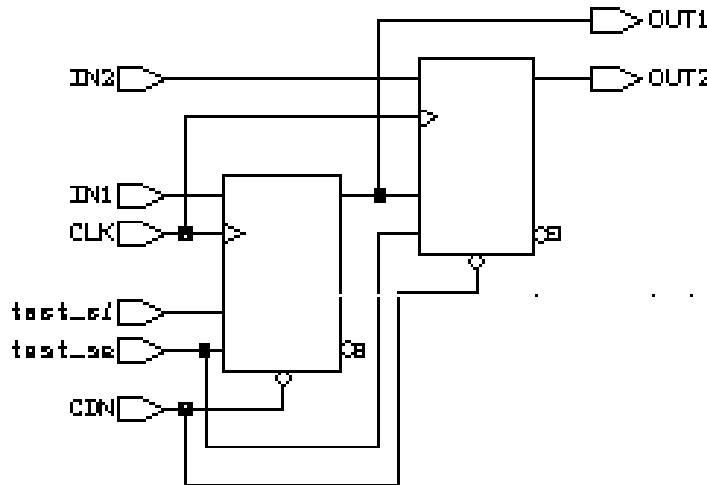
```
Scan chain violations completed...
```

The following sections provide examples of the typical causes of uncontrollable scan cells.

## Incomplete Test Configuration

[Figure 6-31](#) shows a simple scan design.

*Figure 6-31 Simple Scan Design*



When reading this design from an ASCII netlist, specify the test ports. If you neglect to identify the scan input port, DFT Compiler does not flag any violations, but it will not be able to extract scan chains. If you issue the `report_scan_path -chain all` command, you will get the report shown in [Example 6-7](#).

### *Example 6-7 Incomplete Scan Test Report*

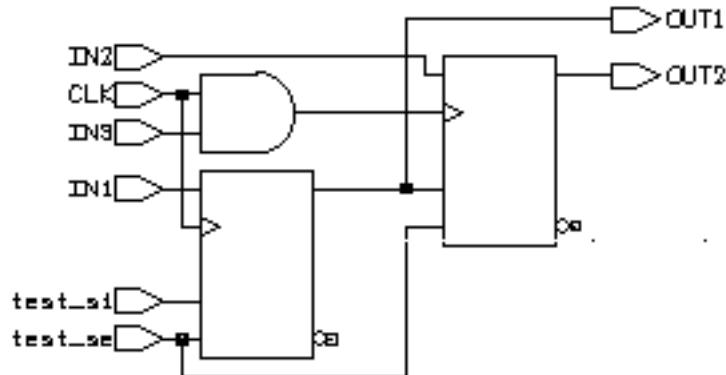
```
dc_shell> report_scan_path -chain all
*****
Report : test
      -report_scan_path
*****
Attributes indicate that the design 'fig5_2' has scan
circuitry.
But no scan-path information was found for design 'fig5_2'.
Please ensure that the scan ports for design 'fig5_2' are
defined and run dft_drc to reconstruct the scan path
information.
1
```

To resolve this, identify the scan input ports, scan output ports, test clocks, and asynchronous sets and resets and rerun `dft_drc`.

## Invalid Clock Logic

Figure 6-32 shows a design with a combinational gated clock.

Figure 6-32 Combinationally Gated Clock



If you do not hold port IN3 at logic 1 during scan shift, pulses applied at clock port CLK might not reach the clock pin of cell FF\_B; therefore, the clock input of cell FF\_B violates the test clock requirements. DFT Compiler generates messages such as these:

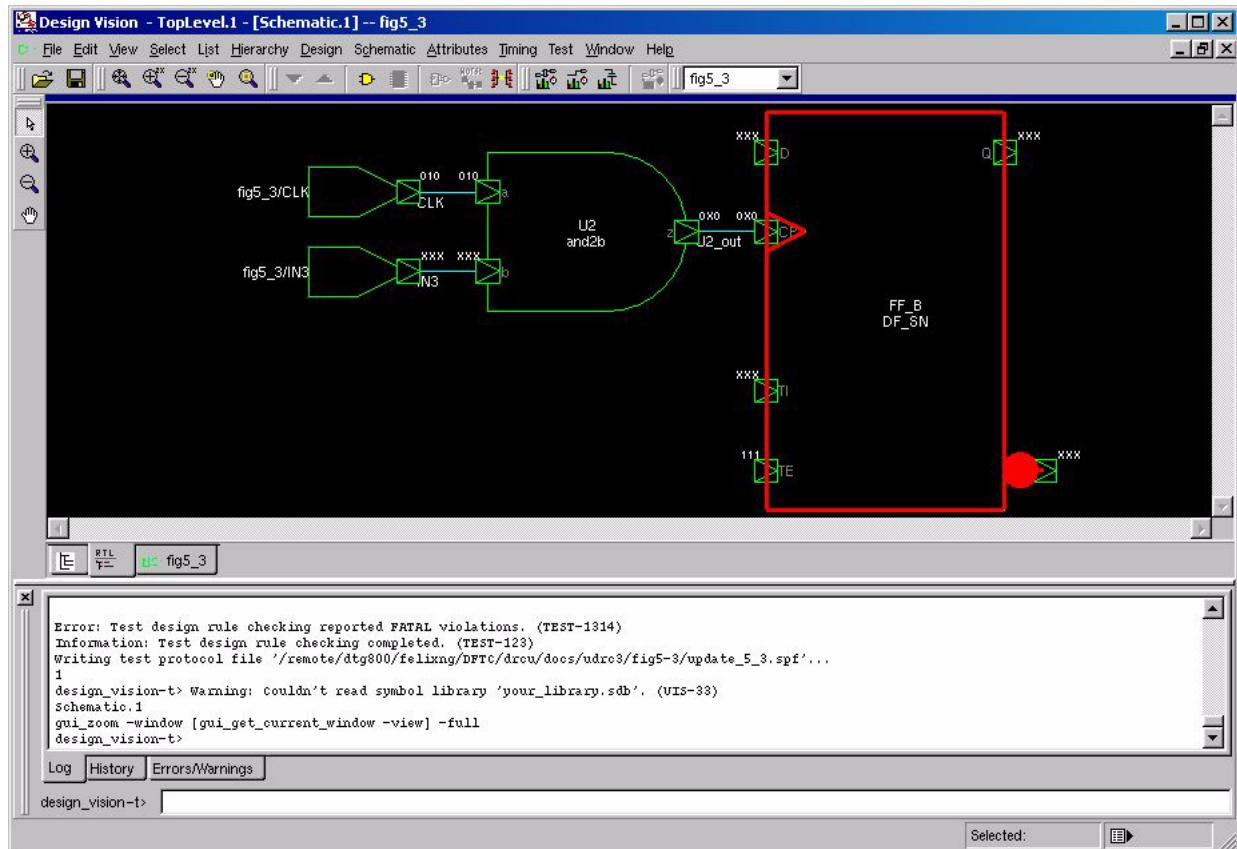
-----  
Begin Scan chain violations...

Error: Chain c1 blocked at DFF gate U1 after tracing 0  
cells. (S1-1)

Scan chain violations completed...  
-----

Bring up the Design Vision Graphical Schematic Debugger, as shown in [Figure 6-33](#).

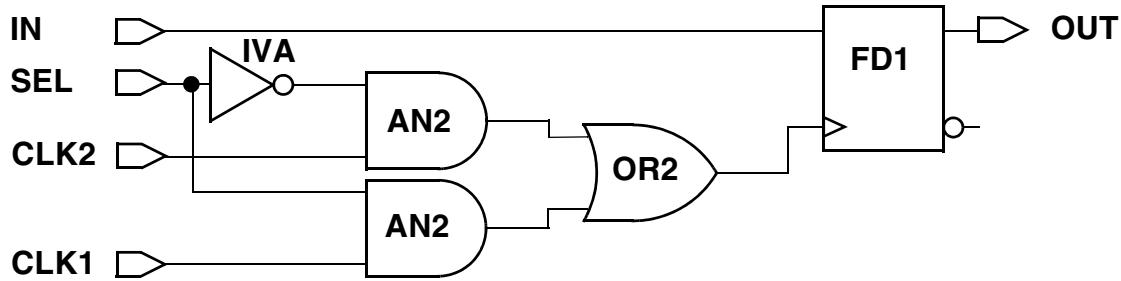
Figure 6-33 The Design Vision Graphical Schematic Debugger



The debugger shows that the clock input of the cell FF\_B contains an X. This indicates that the clock was completely controllable.

In Figure 6-34, if SEL = 1, the path from CLK1 is active, although the path from CLK2 is not. In general, you use the `set_dft_signal` command to specify constant logic values on ports, as explained later in this chapter.

Figure 6-34 A Clock Selector Network



In this example, if you specify `set_dft_signal-view existing_dft -type Constant -active_state 1` on the SEL port, you will see this violation:

```
-----
Begin Pre-DFT violations...

Warning: Clock CLK2 cannot capture data with other clocks
off. (D8-1)

Pre-DFT violations completed...
---
```

A D8 violation indicates that a clock cannot capture data while others are off. Each clock must be capable of capturing data. This does not prevent scan insertion, but you might want to investigate the cause of the violation.

You can correct invalid clock gating violations by inserting logic.

If a clock pin is driven by constant logic, the `dft_drc` command issues a warning:

```
Warning: Clock input CP of DFF FF_A couldn't capture data.
(D17-1)
```

The waveforms of the inferred clocks are taken either from a previous invocation of the `set_dft_signal` command or from the scan style-dependent default values.

## Incorrect Clock Timing Relationship

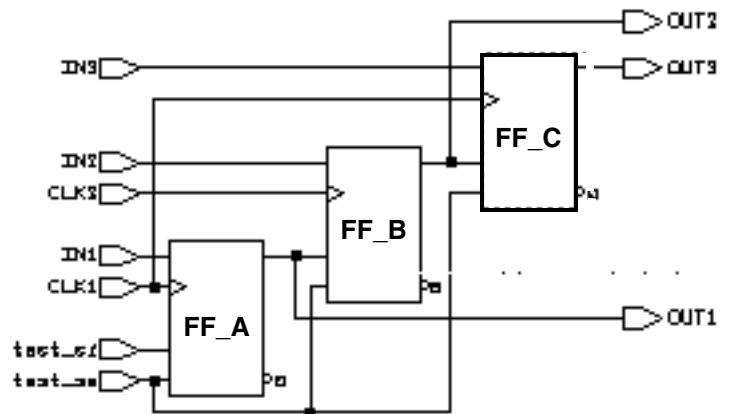
A structurally valid scan chain becomes invalid due to the clock timing definitions in the following cases:

- The cell ordering of the scan chain in a scan design with multiple clock domains has later cells triggered by later clocks (data flow-through).

- The active levels of the master clock and the slave clock overlap in designs with two-phase clocking.

[Figure 6-35](#) shows a scan design with multiple clocks. Structurally this design meets the scan design rules. However, the ability to shift data through the scan chain depends on the relationship between the multiple clocks.

*Figure 6-35 Existing Scan Design With Multiple Clocks*



Unless CLK1 and CLK2 have identical timing, this design always results in an invalid scan path due to the clock timing relationship. CLK2 triggers cell FF\_B, and CLK1 triggers both the cell driving it (FF\_A) and the cell driven by it (FF\_C).

If the clock timings are identical, design rule checker will report messages such as

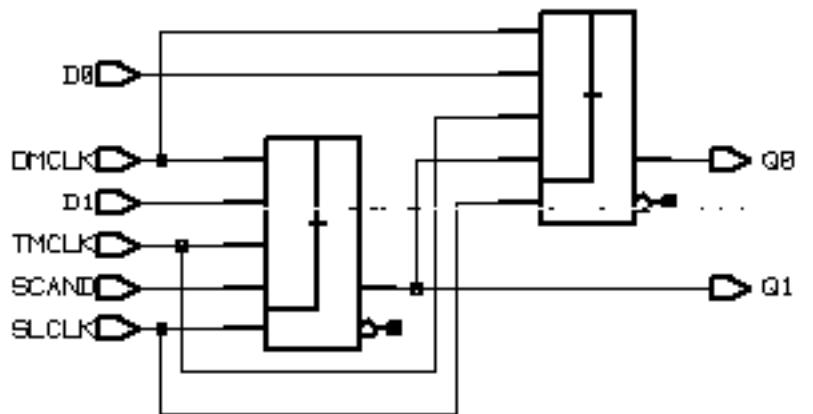
Warning: Multiple clocks (CLK1 CLK2) were used to shift scan chain c1. (S22-1)

If the clock timings are different, design rule checker will report messages such as

Warning: Dependent slave FF\_B may not hold same value as master FF\_A. (S29-1)

[Figure 6-36](#) shows an LSSD design. Structurally, this design meets the scan design rules. However, the ability to shift data through the scan chain depends on the relationship between the master clock (TMCLK) and the slave clock (SLCLK).

Figure 6-36 Simple LSSD Design



DFT Compiler uses zero-delay timing, so you cannot depend on delays in the clock nets to prevent overlapping master and slave clocks. Because DFT Compiler considers both the master and slave clocks active at 55 ns after the start of the vector, this command sequence defines an invalid timing relationship for the design in [Figure 6-36](#):

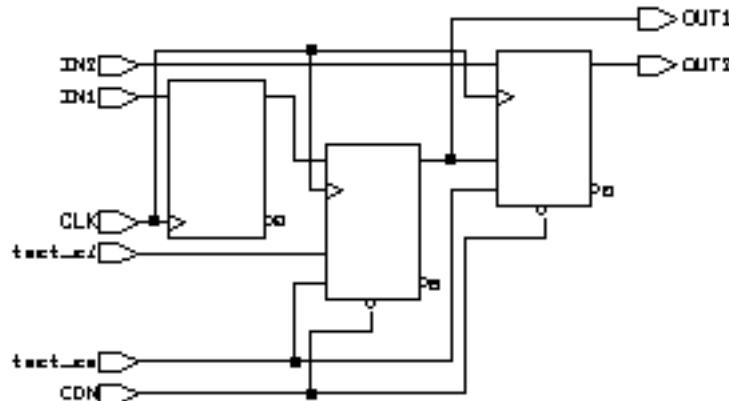
```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port TMCLK

dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 55 65] \
    -port SLCLK
```

## Nonscan Sequential Cells

[Figure 6-37](#) shows a scan design with a nonscan sequential cell.

*Figure 6-37 Scan Design With Nonscan Sequential Cell*



DFT Compiler supports this configuration but generates uncontrollable-scan-cell messages to indicate exclusion of the nonscan cell from the scan chain.

If the nonscan cell has a `scan_element false` attribute, DFT Compiler generates messages such as this:

```
Warning: Nonscan DFF U1 disturbed during time 45 of shift procedure. (S19-1)
```

---

## Ability to Capture Data Into Scan Cells

To ensure that the parallel capture cycle results in data that is successfully captured into the scan cells, DFT Compiler verifies that

- The capture data is valid

Valid capture data depends only on the scanned-in state and primary input values. Modification of capture data by other capture data or the capture clock invalidates the capture data.

- The system clock pulse arrives at the system clock pin of each scan cell

If a scan cell does not meet these conditions, DFT Compiler cannot capture data into the scan cell. Typical causes of failed data capture include the following:

- A clock signal drives the data input to a scan cell.
- A functional path in the design has sequential endpoints clocked by different clock domains (untestable functional path).
- A bidirectional port drives the data input to a scan cell, and the data is released before the capture clock.

- A master-slave cell with an inferred behavior for the B clock pulse causes the cell capture state to be different from the cell scan-out state.
- A sequential element drives an asynchronous input to a scan cell.
- The test protocol does not include a capture clock.

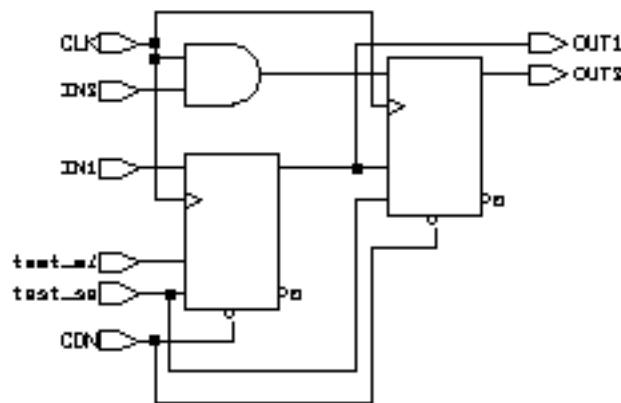
DFT Compiler generates diagnostic messages indicating the source of the violation.

The following sections provide examples of the typical causes of failed data capture.

## Clock Driving Data

In the design shown in [Figure 6-38](#), the clock signal CLK drives the data input to cell FF\_B. Pulsing the clock signal during capture can cause the data input to cell FF\_B to change.

*Figure 6-38 Design With Clock Driving Data*



DFT Compiler generates this message:

```
Warning: Clock CLK connects to LE clock/data inputs CP/D of
DFF FF_B. (C12-1)
```

Although the `dft_drc` output and the scan path report indicate that the affected cell is scannable, the cell is actually scan controllable only.

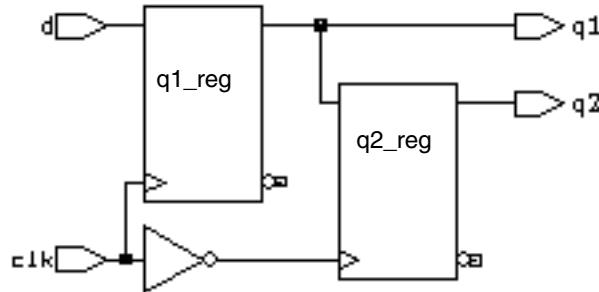
This violation usually has a minor impact on fault coverage, so make it one of the last violations you correct, if at all. Correcting this violation requires the addition of test-mode logic, which also has a minor fault coverage impact. Fixing the violation means trading one set of untested faults for another, possibly smaller, set of untested faults.

Use the Design Vision Graphical Schematic Debugger to locate and analyze the clock-driving data problem.

## Untestable Functional Path

Figure 6-39 shows a design with an untestable functional path. A functional path exists between cells q1\_reg and q2\_reg. Using the default clock waveform of rising edge at 45 ns and falling edge at 55 ns, q2\_reg receives the data captured in cell q1\_reg.

Figure 6-39 Untestable Functional Path



Because the capture data in cell q2\_reg depends on data other than the scanned-in state and the primary input values, DFT Compiler generates messages such as these:

```
Warning: Clock clk can capture new data on TE input CP of
DFF q2_reg. (D14-1)
Source of violation: input CP of DFF q1_reg.
```

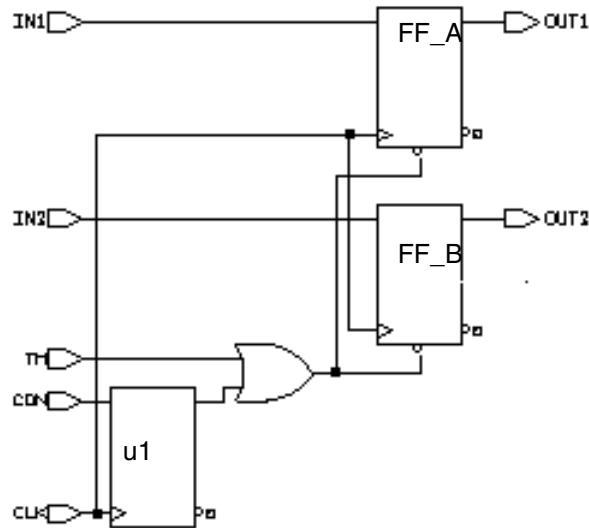
Use the Design Vision Graphical Schematic Debugger to locate and analyze the untestable functional path problem. Contact Synopsys support personnel for access to a script that loads the debugger.

In most cases, you must change the design to correct the problem.

## Uncontrollable Asynchronous Pins

The asynchronous pins shown in Figure 6-40 are uncontrollable, because they are driven by sequential logic. If you hold the TM signal at logic 1 only during scan shift, the asynchronous resets on cells FF\_A and FF\_B can change as a result of the capture clock.

Figure 6-40 Uncontrollable Asynchronous Pins

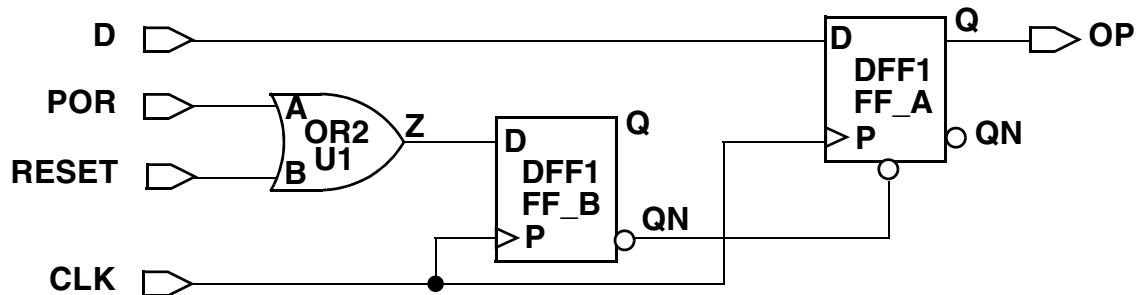


DFT Compiler will report the following:

Warning: Clock CDN cannot capture data with other clocks off. (D8-1)

Uncontrollable pins usually occur when the asynchronous signal is generated from the state of other sequential devices, as shown in Figure 6-41. You can correct this violation by inserting test mode logic.

Figure 6-41 Circuit With Uncontrollable Asynchronous Clear





# 7

## Advanced DFT Architecture Methodologies

---

The chapter describes advanced DFT architecture-related methodologies and processes in the following sections:

- Performing Scan Extraction
- Inserting Observe Test Points
- Using AutoFix
- Implementing User-Defined Test Points
- Pipelining Scan Enable Architecture
- Using Multimode Scan Architecture
- Multivoltage Support
- Power-Aware Functional Output Gating
- Internal Pins Flow
- Support for Implicit Scan Chain Elements
- Creating Scan Groups
- Identification of Shift Registers

---

## Performing Scan Extraction

The scan chain extraction process extracts scan chains from a design by tracing scan data bits through the multiple time frames of the protocol simulation. For a given design, specifying a different test protocol can result in different scan chains. As a corollary, scan chain-related problems can be caused by an incorrect protocol, by incorrect `set_dft_signal` specifications, or even by incorrectly specified timing data.

When performing scan extraction, you always use the descriptive view (`-view existing_dft`), because you are defining test structures that already exist in your design.

To perform scan extraction

1. Define the scan input and scan output for each scan chain. To define these relationships, first use the `set_scan_configuration` command to specify the scan style and then use the `-view existing_dft` option with the `set_scan_path` and `set_dft_signal` commands, as shown in the following examples:

```
dc_shell> set_scan_configuration \
           -style multiplexed_flip_flop

dc_shell> set_dft_signal -view existing_dft \
           -type ScanDataIn -port TEST_SI

dc_shell> set_dft_signal -view existing_dft \
           -type ScanDataOut -port TEST_SO

dc_shell> set_dft_signal -view existing_dft \
           -type ScanEnable -port TEST_SE

dc_shell> set_scan_path chain1 \
           -view existing_dft \
           -scan_data_in TEST_SI \
           -scan_data_out TEST_SO
```

2. Define the test clocks, reset, and test mode signals by using the `set_dft_signal` command.

```
dc_shell> set_dft_signal -view existing_dft \
           -type ScanClock -port CLK \
           -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
           -type Reset -port RESETN \
           -active_state 0
```

3. Create the test protocol by using the `create_test_protocol` command.

```
dc_shell> create_test_protocol
```

4. Extract the scan chains by using the `dft_drc` and `report_scan_path` commands.

```
dc_shell> dft_drc  
dc_shell> report_scan_path -view existing_dft \  
      -chain all  
dc_shell> report_scan_path -view existing_dft \  
      -cell all
```

---

## Inserting Observe Test Points

Observe test point insertion is a technique that reduces test pattern count while maintaining test coverage in your design. It can be enabled and implemented along with regular scan insertion. You can use your regular scan insertion scripts with some add-on commands to enable test point insertion.

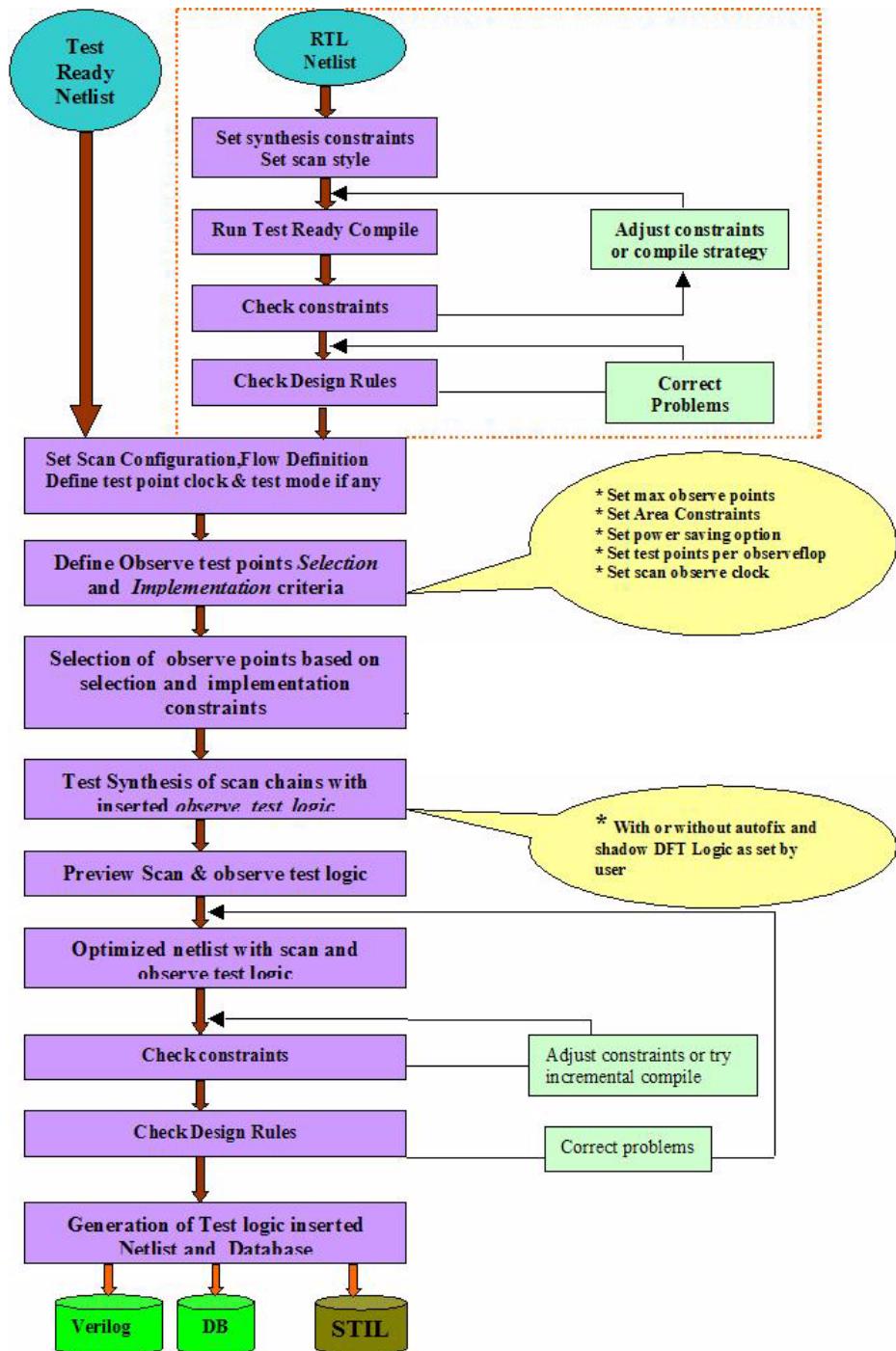
For details on adaptive scan technology, see the *DFT MAX User Guide: Adaptive Scan*.

The following sections explain the observe test point insertion process:

- [Understanding Observe Points](#)
- [Reading In Your Netlist and Configuring for Scan](#)
- [Configuring for Observe Points](#)
- [Previewing Scan and Observe Test Point Logic](#)
- [Inserting Observe Test Logic With Scan Chains](#)

These steps are illustrated in [Figure 7-1](#).

Figure 7-1 Bottom-Up Observe Point Analysis and Bottom-Up Test Point Insertion Flow



---

## Understanding Observe Points

A node in a design that must be observed to improve the testability of a design is called an *observe point*. The node can be made observable either by connecting it to a primary output or by connecting it to a functional input of a scan flip-flop and stitching the flip-flop into a scan chain. The new scan flip-flop inserted to observe the node is called an *observe flip-flop*. The observe flip-flop can detect numerous hard-to-detect faults and can significantly reduce the scan pattern count.

The signal that triggers the observe flip-flop is called the *observe clock*. There are two types of observe clocks:

- A dominant observe clock.

This clock is the existing scan test clock that triggers the most scan flip-flops in the design.

- A dedicated observe clock.

An existing data port or a newly created dedicated port could be used as an observe clock.

DFT Compiler supports two methods for doing observe point analysis: bottom-up observe analysis and top-down observe analysis. The preferred method for SoC designs is bottom-up observe analysis, which is discussed here.

In the bottom-up observe test point analysis flow, you use DFT Compiler to specify the observe test point selection and implementation constraints for each scan-replaced module of a top-level design. DFT Compiler performs observe point analysis and synthesizes selected observe points for each module. This is done independently, with specified clocking mechanisms for observe flip-flops. All modules are stitched together to create the top-level design. Finally, ATPG analysis is performed to obtain the test coverage for the entire chip-level design.

---

## Reading In Your Netlist and Configuring for Scan

Your first step is to read in your test-ready design netlist. For more information on performing this step, see [Chapter 2, “Running RTL Test Design Rule Checking.”](#)

After reading in your design netlist, specify all your scan configuration requirements as you normally do in DFT Compiler and then enable test volume data reduction functionality.

Note:

If you intend to use AutoFix on your design, you must do so before inserting observe points. For more information on AutoFix, see [“Using AutoFix” on page 7-14.](#)

---

## Configuring for Observe Points

This section contains the following subsections about tasks associated with observe point configuration:

- [Enabling Observe Point Analysis](#)
- [Defining an Observe Clock](#)
- [Defining a Test Mode](#)
- [Selecting and Implementing Observe Test Logic](#)

### Enabling Observe Point Analysis

You enable observe point analysis on your design when you use the following commands:

```
dc_shell> set_dft_configuration \
           -observe_points enable|disable \
           -control_points enable|disable
dc_shell> set_testability_configuration -type observe
```

The `-type` option is a mandatory switch for the `set_testability_configuration` command. All other command switches are optional. If you do not set the options on both of these commands, observe point analysis is not performed on your design.

#### Caution!

If you set both the `lbist` and `testability` options on the `set_testability_configuration` command, neither LBIST nor observe point insertion functionality is invoked. An error message is displayed, warning you that these two options cannot be performed together.

### Defining an Observe Clock

You can choose to use an existing port as your observe clock by using the following command:

```
set_testability_configuration -type observe \
                             -clock_signal MY_CLK
set_testability_configuration -type control \
                             -clock_signal MY_CLK -view spec
```

If you do not choose to use an existing port, the dominant or dedicated test clock is used instead. The clock type chosen in this case is dependent on the type of configuration you have chosen.

Note:

If you do specify an observe clock, its precedence is higher than the dominant or default dedicated observe clock for observe flip-flops defined by use of the `set_testability_configuration` command.

## Defining a Test Mode

You can specify a module-level test mode port by using the following command:

```
set_dft_signal -port testmode1 -view spec -type TestMode \
               -active_state 1
```

## Selecting and Implementing Observe Test Logic

The term *observe test logic* describes all the logic associated with the implementation of observe test points.

You select and implement observe test point logic by using the `set_testability_configuration` command. With this command, you can define

- The maximum number of observe points to be selected
- The maximum area overhead you want to allow
- The implementation of power-saving logic
- The number of test points per observe flip-flop
- The observe clock type

The following definitions and corresponding options provide selection criteria for choosing the number of observe test points.

### Setting the Maximum Observe Points

You can set the maximum number of observe points to insert into a module by using the command

```
set_testability_configuration -type max_test_points N
```

This command controls how many observe points will be inserted into the design.

If you do not set a value, DFT Compiler selects an optimal set of test points. This optimal value depends on the area and other constraints defined in the `set_testability_configuration` command.

The number of observe points defined with the `max_test_points` option can be affected if

- The `max_additional_logic` option is also set

- The number of observe points found by DFT Compiler is less than the number set by the `max_additional_logic` option

For example, you might define the `max_test_points` value to be 500, but DFT Compiler can find only 400 test points. In such a case, the following will occur:

- Before implementing the 400 test points, DFT Compiler will check the `max_additional_logic` constraint.
- If the area constraint is not violated, all 400 test points are selected.
- If the area constraint is violated with a count of 400 test points, DFT Compiler will identify a suitable number of test points that do not violate the area overhead constraint and will implement that number.

### **Setting the Maximum Extra Area Overhead**

You can define the maximum amount of extra area overhead to allocate for observe test logic by using the following command:

```
set_testability_configuration -type observe \
    max_additional_logic_area N
```

The area overhead option has a higher priority than the `max_additional_logic_area` and `test_points_per_scan_cell` options in determining observe point selection.

`N` is the percentage of the area occupied by test point logic in a given current design.

The area overhead defined with the `max_additional_logic_area` option can

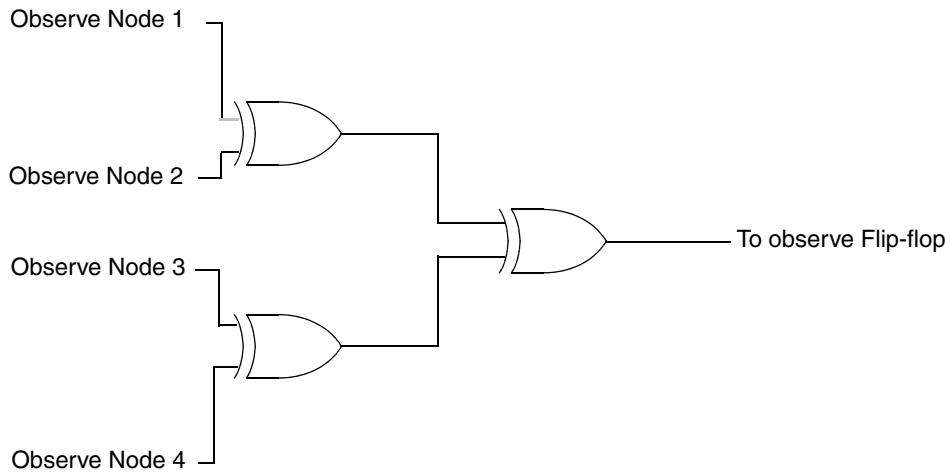
- Cause the number of observe points implemented to be fewer than the number defined by the `max_test_points` or by DFT Compiler if those test points cause the area overhead allocation to be exceeded
- Cause the number of observe points per observe flip-flop implemented to be less than the number defined by the `test_points_per_scan_cell` option

If no value is set for this option, area overhead is not restricted.

### **Setting the power\_saving\_on Option**

Typically, observe points are implemented as a network of balanced XOR gates routed to an observe flip-flop. See [Figure 7-2](#).

Figure 7-2 XOR Tree Network Without Power Save Logic



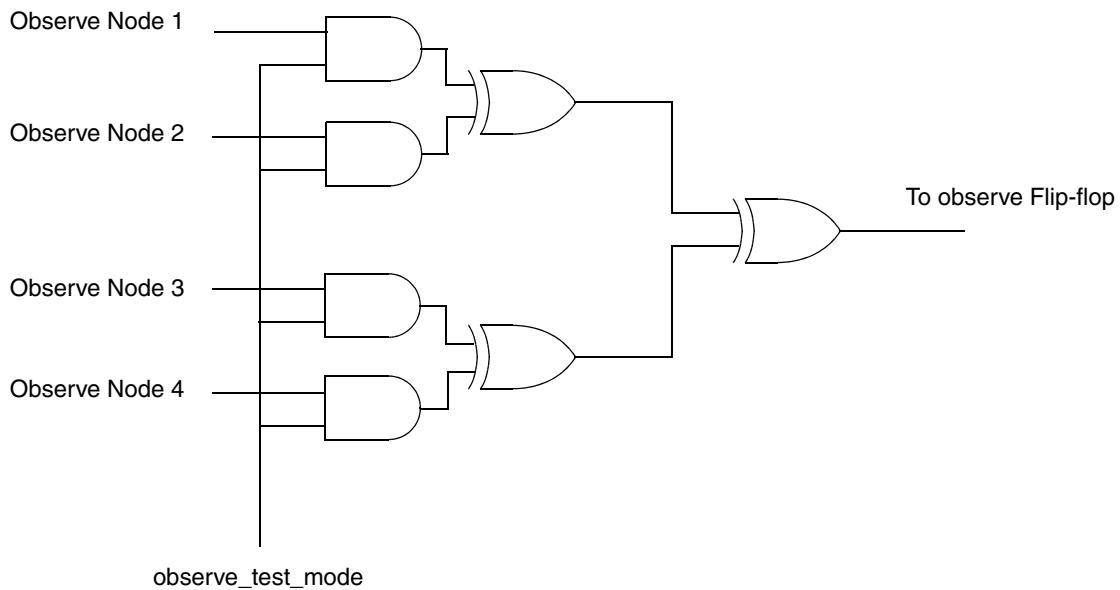
When a device is in functional mode, every time the logic value on an observe node changes, either from 1 to 0 or from 0 to 1, the entire XOR gate network will toggle. This toggling can result in significant power losses. To avoid such losses in power, use the following command:

```
dc_shell> set_testability_configuration -type observe\  
          -power_saving enable
```

The default value for this option is `disable`.

The power saving logic that is added to a design when `-power_saving` is set to `enable` is illustrated in Figure 7-3.

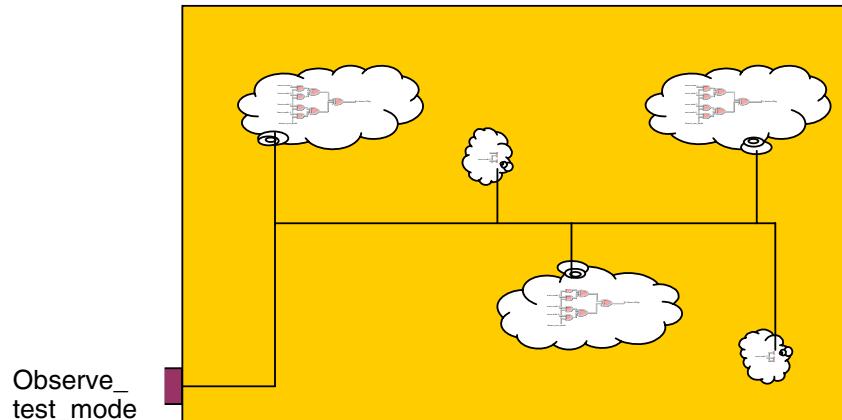
Figure 7-3 XOR Tree Network With Power Save Logic



The results of the `power_saving` implementation are as follows:

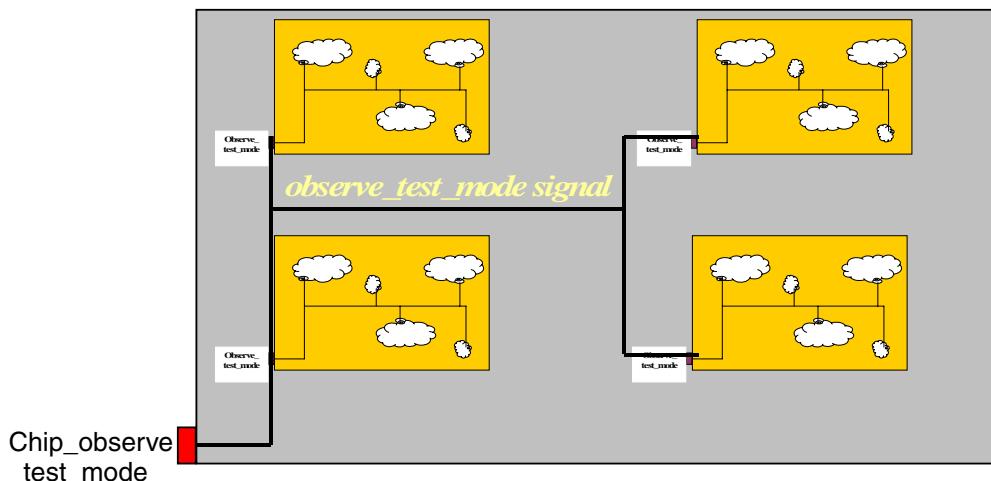
- Setting the `power_saving` option to enable causes the insertion of two input AND gates into every observe node.
- One of the inputs of each AND gate is connected to a new global signal called `observe_test_mode`. This signal is set to logic 1 in test mode. At all other times, it remains at logic 0.
- If a `test_mode` signal is already defined in a module, the `observe_test_mode` signal is tied to the module's `test_mode` port.
- If a `test_mode` signal is not defined in a module, a new port, called `observe_test_mode`, is created. All of these modules are connected together at the top level, as shown in [Figure 7-4](#).

*Figure 7-4 Inserting a New test\_mode Port at the Module Level*



- If no definition of a `test_mode` port exists at the chip level, a new port named `chip_observe_test_mode` is created. All module-level `observe_test_mode` ports are tied to this port, as shown in [Figure 7-5](#).

*Figure 7-5 Chip Level Connection for observe\_test\_mode*



If the addition of power save logic exceeds the existing `observe_logic` area limit, the following might be affected:

- The number of test points inserted might be less than the count defined with the `max_test_points` option.

- The number of observe test points per observe flip-flop inserted might be less than the number defined by the `test_points_per_scan_cell` option.

Note:

If the value defined in the `test_points_per_scan_cell` option is 1, the power saving option is disregarded and no gating logic is implemented.

### Defining the Observe Clock Type

When you set the `clock_type` option, you can choose from the two following clock types:

- The dominant clock of the module in which observe points are to be inserted.  
Although a module can have many clocks, the dominant clock is the module-level clock that triggers the maximum number of flip-flops. The dominant clock type is the default value of the `clock_type` option.
- A new dedicated clock for observe flip-flops.  
A dedicated clock is an observe clock that triggers only observe flip-flops at the chip level.

To select the dedicated clock type, issue the following command:

```
% set_testability_configuration -type observe \
    -clock_type dedicated
```

If you define a dedicated clock, DFT Compiler creates the clock name, such as `tpclk`.

The dedicated clock implementation has the following characteristics:

- Each module with an observe point is assigned a new port.
- All modules with dedicated clock ports are tied together at the chip level and are connected to a user-specified port or to a new chip-level port created by DFT Compiler.
- The dedicated observe clock triggers only the observe flip-flops. As a result, the clock grouping feature can be used effectively in an ATPG run with TetraMAX for better pattern count reduction.

Note:

If a user-defined `test_point` clock is already specified, this option is ignored irrespective of clock type.

### Defining the Number of Test Points for Each Observe Flip-Flop

You can choose the structure of an XOR tree that is associated with observe flip-flops by defining the number of test points to assign to each of them.

You do this by using the following command:

```
% set_testability_configuration -type observe \
    -test_points_per_scan_cell N
```

In the previous example, N is a positive integer. The default value is 8.

The `-test_points_per_scan_cell` option can be affected in two ways:

- If the value you choose for this option causes the maximum area overhead constraint, the implementation of this command might be changed.
- If you select a single observe point per observe flip-flop, the power saving option is disregarded.

---

## Previewing Scan and Observe Test Point Logic

To preview the observe test logic you have chosen to implement, use the following command:

```
dc_shell> preview_dft -test_points all
```

When you use this command, the following information is reported:

- The number of observe test points selected
- The number of observe flip-flops

The command also reports on the functionality that is to be added to the design:

- Observe clock type and name
- Power-save-on status

All other preview options are available, without changes to their functionality.

---

## Inserting Observe Test Logic With Scan Chains

After you define the observe test logic configuration, the `insert_dft` command inserts the following additional logic, depending on the configuration constraints you have defined:

- A new port for a dedicated `observe_clock`, if required, for requested modules
- A new `observe_test_mode` test control signal for the `-power_saving` option if a module does not contain a `test_mode` definition

---

## Using AutoFix

The AutoFix feature automatically fixes scan rule violations associated with uncontrollable clocks, uncontrollable asynchronous set signals, uncontrollable asynchronous reset signals, three-state signals, and bidirectional signals.

By default, AutoFix is disabled. It will not work unless you specifically enable it. AutoFix is supported in both the multiplexed flip-flop and LSSD scan styles.

To use AutoFix, you enable the utility and specify the scope of the design to which it will apply. You can use AutoFix to fix uncontrollable clocks only, asynchronous preset/clear inputs only, or both, and you can specify one leaf-level cell, one hierarchical cell, or a list of cells on which to operate. Within the defined scope of the design, AutoFix automatically fixes all violations of the specified type(s) found by `dft_drc`. If there are no violations, AutoFix does not make any changes to the design.

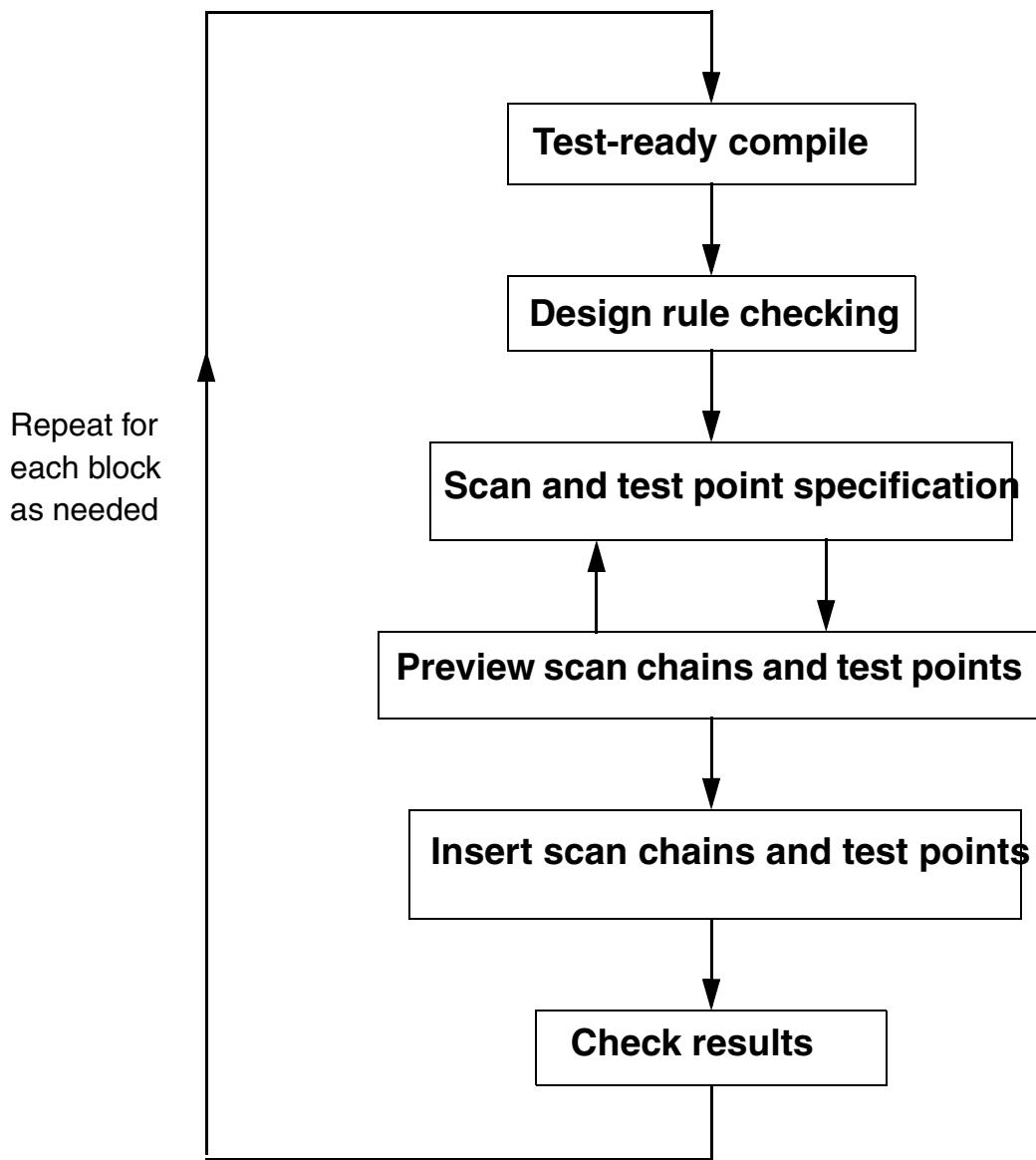
This section contains the following subsections:

- [Understanding the Flow](#)
  - [When to Use AutoFix](#)
  - [Configuring AutoFix](#)
  - [Previewing and Inserting Scan Chains and Test Points](#)
  - [AutoFix Script Example](#)
  - [Top-Down and Bottom-Up Design Flows](#)
- 

### Understanding the Flow

The AutoFix design flow with test point insertion is very similar to the ordinary scan synthesis design flow. The general steps in the design flow are illustrated in [Figure 7-6](#).

*Figure 7-6 Scan Synthesis Design Flow With Test Point Insertion*



You start with the `compile -scan` and `dft_drc` commands. Then you specify the parameters for scan insertion and test point insertion. After you set these parameters, you run `preview_dft` to get a preview of the scan chains and test points. If necessary, you repeat the setup steps to obtain the desired configuration of scan chains and test points.

When this configuration is satisfactory, you perform scan chain routing and test point insertion with the `insert_dft` command. Finally, you check the results with the `dft_drc -coverage_estimate` and `report_scan_path` commands.

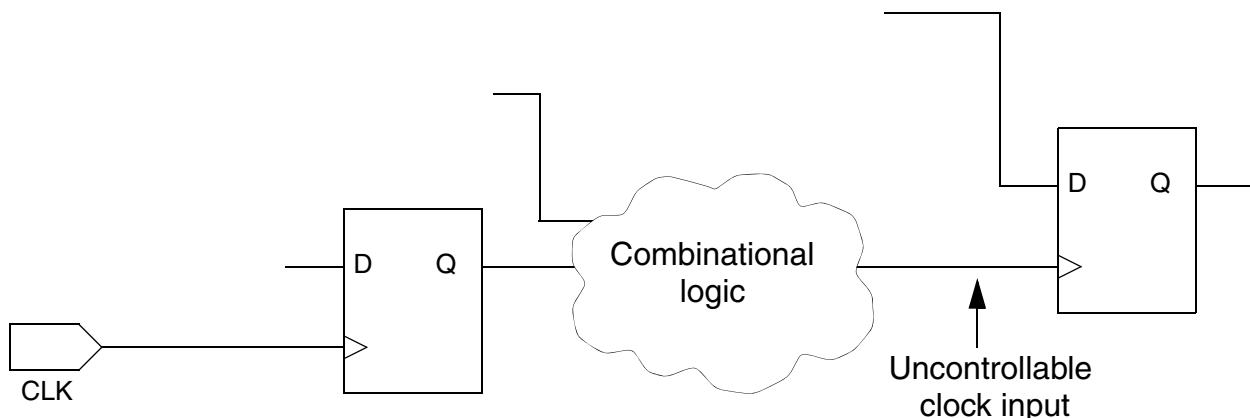
---

## When to Use AutoFix

Each scan flip-flop in a design must be clocked by a signal that can be controlled by a primary input port. Otherwise, clocking of data into the flip-flop cannot be controlled during test. In addition, the asynchronous preset and clear inputs of each flip-flop must be inactive during test. Otherwise, the data in the flip-flop can be set or cleared at any time, leaving unknown data in the flip-flop.

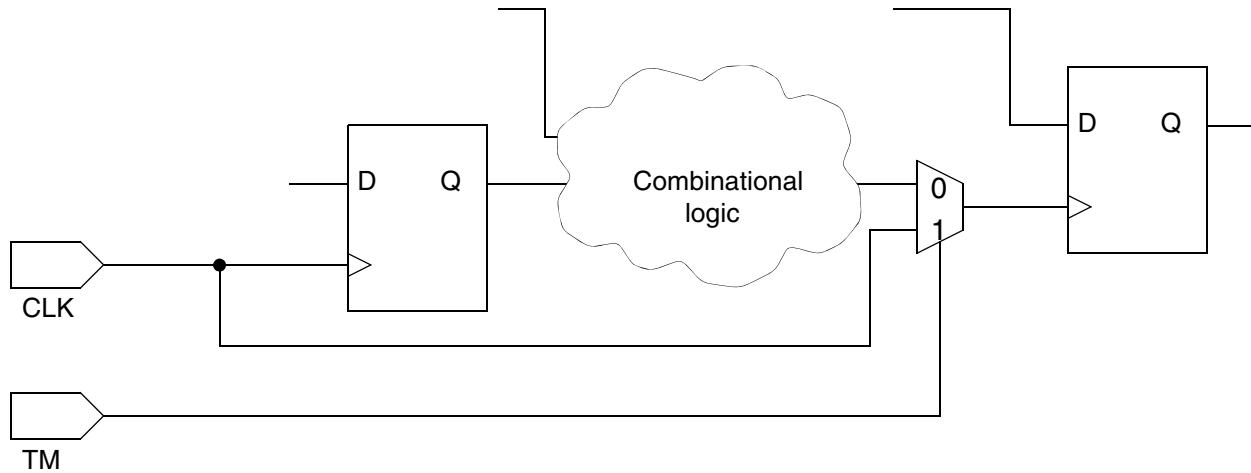
If a flip-flop is clocked by an uncontrollable signal, the command flags the condition as a design rule violation. For example, [Figure 7-7](#) shows a portion of a schematic containing a flip-flop whose clock input cannot be controlled from a primary input. The `dft_drc` command reports this as a design rule violation. If you do not fix this violation, the flip-flop will not be included in a scan chain and faults downstream from the flip-flop output might not be detectable.

*Figure 7-7 Uncontrollable Clock Input*



AutoFix can automatically fix each uncontrollable clock input as shown in [Figure 7-8](#). It inserts a multiplexer at the clock input. The multiplexer is controlled by a “test mode” primary input signal, called TM in this example. For normal, mission-mode operation, the TM signal is inactive and the circuit operates the same as before. During test, the TM signal is asserted, causing the flip-flop to be clocked by a primary input signal, CLK in this example. This configuration allows the flip-flop to be controlled during testing.

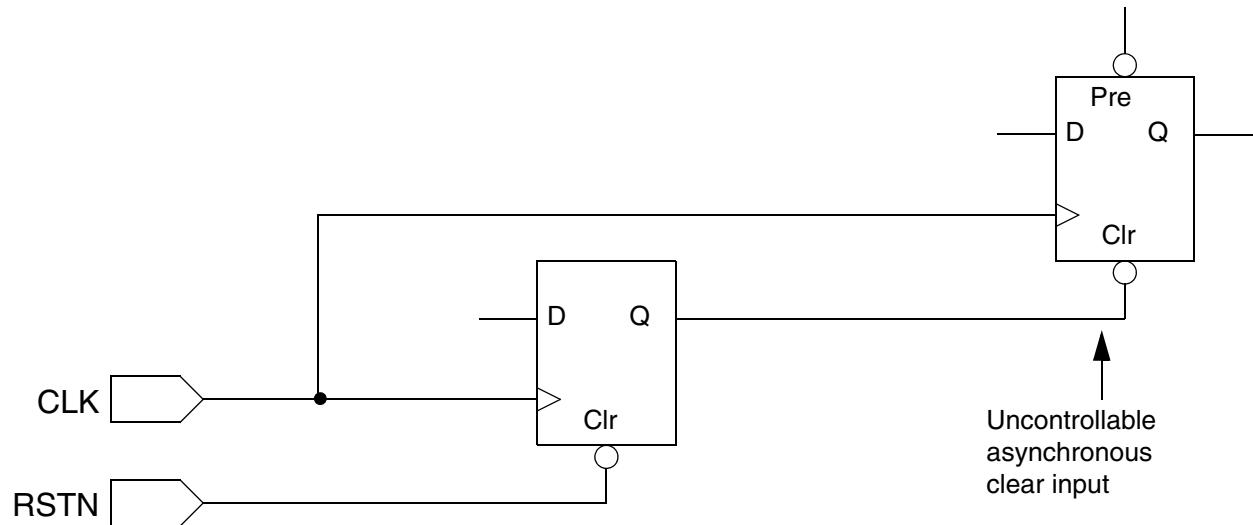
Figure 7-8 Uncontrollable Clock Input Fixed by AutoFix



The same TM signal can be used to enable and disable all multiplexers inserted by AutoFix. Note that the TM signal is different from the scan-enable signal used to control shift and capture with scan chains. The test mode signal is always active during test, even during capture time.

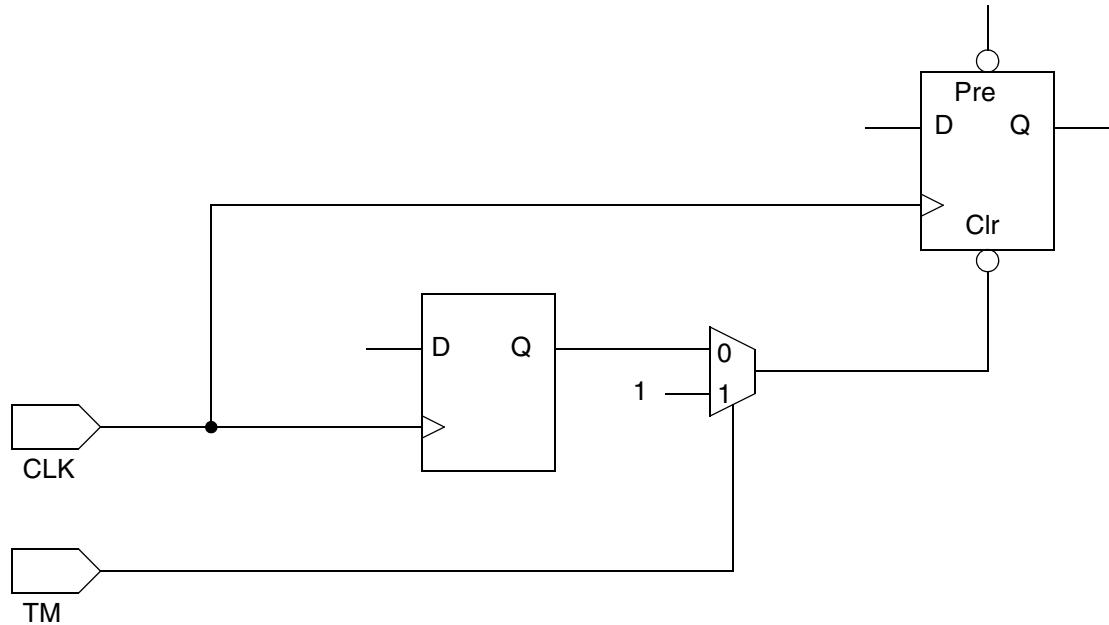
AutoFix can automatically fix uncontrollable asynchronous preset and clear inputs in a similar manner. For example, consider the subcircuit in Figure 7-9. The asynchronous clear input of the upper-right flip-flop is connected directly to the Q output of the lower-left flip-flop, causing the upper-right flip-flop to be cleared at unpredictable times.

*Figure 7-9 Uncontrollable Asynchronous Clear Input*



When asked to fix this type of violation, AutoFix inserts a multiplexer with one input connected to logic 1, as shown in [Figure 7-10](#). This is functionally the same as an OR gate. For mission-mode operation, the TM signal is inactive and the circuit operates the same as without the inserted logic. During test, the TM signal is asserted, which holds the clear input inactive.

*Figure 7-10 Uncontrollable Asynchronous Clear Input Fixed by AutoFix*



---

## Configuring AutoFix

The following sections explain the AutoFix configuration process:

- [Enabling Test Point Utilities](#)
- [Specifying Test Point Signals](#)
- [Specifying AutoFix Behavior](#)

### Enabling Test Point Utilities

The command for enabling test point utilities is `set_dft_configuration`. This command works like the `set_scan_configuration` command, except that it sets up test point parameters rather than scan chain parameters. Using the `set_dft_configuration` command, you specify the test point utilities that are to be applied.

To show the current `set_dft_configuration` settings, use the `report_dft_configuration` command. To remove the current settings, use the `reset_dft_configuration` command.

### Specifying Test Point Signals

Each test point utility can use a new or existing primary input to control the test mode and can use another new or existing clock as a test point clock. The `set_dft_signal` command lets you specify which existing primary input signals are allowed to be used for specific test purposes. It also enables you to specify which existing ports can be used for certain scan control purposes.

Using the `set_dft_signal -type` command, you specify the test point signal type (the test mode pin for all test point utilities) and the existing primary input port that can be used for that purpose. In the absence of such a specification, the test point utility does not use an existing primary input port. Instead, it creates a new dedicated input port.

AutoFix uses the following signal types for the `set_dft_signal -type` command:

- **TestData**  
Defines the external test data pin.
- **TestMode**  
Use this signal type to specify the test mode pin.

### Specifying AutoFix Behavior

You can use the following commands for specifying the behavior of AutoFix:

- `set_autofix_configuration`
- `set_autofix_element`

The `set_autofix_configuration` command specifies the types of design rule violations to fix: uncontrollable clock inputs, uncontrollable asynchronous preset/clear inputs, or both. All violations of the specified type(s) are fixed, unless you specify otherwise with the `set_autofix_element` command. You can also specify an asynchronous set/reset port that is to be used to fix uncontrollable asynchronous violations automatically. These fixes can be performed on sequential cells and on hierarchical cells that contain sequential cells.

The `-exclude_elements` option of the `set_autofix_element` command lets you exclude a specified list of leaf-level cells or hierarchical cells from being fixed by AutoFix. The command works like the `set_scan_element` command, except that it affects AutoFix rather than scan replacement. The command syntax lets you separately allow or not allow fixing of clock violations and preset/clear violations. You can specify the clock that is connected to the test points on the designated parts of the design. If AutoFix needs to perform a fix and if no existing clock has been assigned, AutoFix looks for a clock that is controllable further back in the logic. If AutoFix does not find a clock in the logic, it creates one.

The following sample script uses `clk` and `resetn` to globally fix all gated clocks and asynchronous set and reset signals in your design:

```
set_dft_signal -type TestData -port clk
set_dft_signal -type TestData -port resetn
```

The following sample script fixes all specified gated clocks, sets and resets, and uses `clk`, `setn`, and `resetn` to control them:

```
set_dft_signal -type TestData -port clk -hookup_pin U6/Z
set_dft_signal -type TestData -port resetn
set_dft_signal -type TestData -port setn
set_dft_signal -type TestMode -port my_test_mode
set_autofix_configuration -type set -test_data setn \
    -control_signal my_test_mode
set_autofix_configuration -type reset -test_data resetn \
    -control_signal my_test_mode
set_autofix_configuration -type clock -test_data clk \
    -control_signal mytest_mode
```

## Previewing and Inserting Scan Chains and Test Points

After you set up the scan chain parameters and test point parameters, you can get a preview of the scan chains and test points before you synthesize them. To do this, use the `preview_dft` command.

When you are satisfied with the scan chain configuration and test point configuration reported by the `preview_dft` command, you synthesize the scan chains and test points by using the `insert_dft` command.

After you run `insert_dft`, you should always run `dft_drc` to check for any remaining design rule violations. To obtain specific information on the inserted scan chains and test points, use the `report_scan_path` command.

---

## AutoFix Script Example

The script in [Example 7-1](#) is a scan synthesis session that includes test point insertion using AutoFix.

### *Example 7-1 Scan Synthesis With Test Point Insertion*

```
current_design MY_DESIGN
compile -scan
create_test_protocol
dft_drc

/* Scan Specification */
set_scan_configuration -clock_mixing mix_edges ...

/* Test Point Architect Specification */
set_dft_configuration -fix_clock enable -fix_reset enable \
set_dft_signal -type TestMode -port TEST_MODE
set_dft signal -type TestData -port TCLK

/* Autofix Specification */
set_ autofix_configuration -type clock -exclude_element U2
set_ autofix_configuration -type reset -exclude_element U2
set_ autofix_element U3/myreg -type clock -test_data TCLK \
    -control_signal TEST_MODE

preview_dft
insert_dft
dft_drc
report_scan_path -chain all
```

---

## Top-Down and Bottom-Up Design Flows

DFT Compiler and AutoFix support both top-down and bottom-up hierarchical design flows. In the top-down approach, you specify the scan architecture and test point parameters at the top level of the design and allow the tools to work on the design as a whole. In the bottom-up approach, you perform scan chain routing and test point insertion in stages—starting with lower-level blocks first and then at higher levels of hierarchy—until you reach the top level.

Using a bottom-up design flow requires stitching together the scan chains and test point control signals as you work your way up the hierarchy.

This section provides two examples of test point insertion using AutoFix. Each example shows a simple schematic of the design, the test point insertion goals, and a set of commands that can be used to achieve those goals.

The examples are discussed in the following sections:

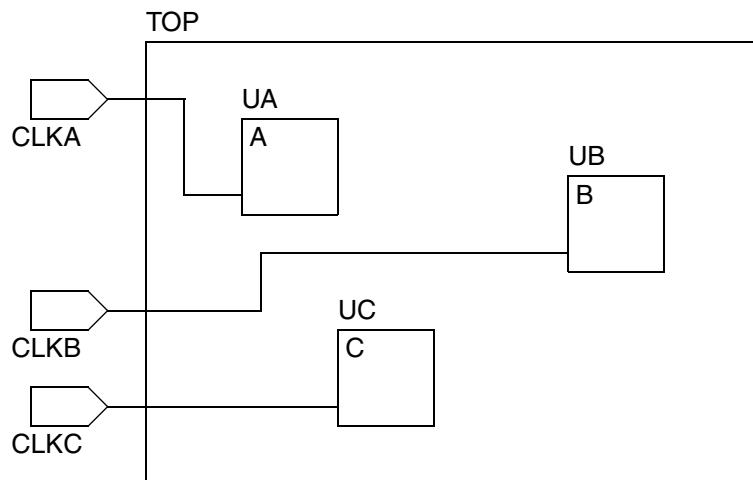
- [Top-Down Example](#)
- [Bottom-Up Example](#)

---

## Top-Down Example

To demonstrate a top-down approach to scan and test point insertion, [Figure 7-11](#) shows a simplified schematic of a design at the top level.

*Figure 7-11 Top-Down AutoFix Example*



This top-level design, called TOP, contains three hierarchical blocks: A, B, and C. There are also three clock ports—CLKA, CLKB, and CLKC—and each is used to clock the respective hierarchical block seen at the top level. The other primary input ports and primary output ports are not shown.

For this example, suppose that you have the following design goals:

- There should be two scan chains: one for elements clocked on positive edges and another for elements clocked on negative edges. Different clocks can be mixed in the same scan chain.

- AutoFix should fix all uncontrollable clock violations and uncontrollable asynchronous preset/clear violations throughout the design, except for clock violations in block C, as noted next.
- AutoFix should use CLKA to fix clock violations in block A.
- AutoFix should use CLKB to fix clock violations in block B.
- AutoFix should not fix uncontrollable clock violations in block C.

Note:

Because sequential elements in block C that have clock violations are not included in any scan chain, there might be a reduction in fault coverage.

[Example 7-2](#) shows a sequence of commands that can be used to achieve these design goals.

#### *Example 7-2 Commands for Top-Down Design*

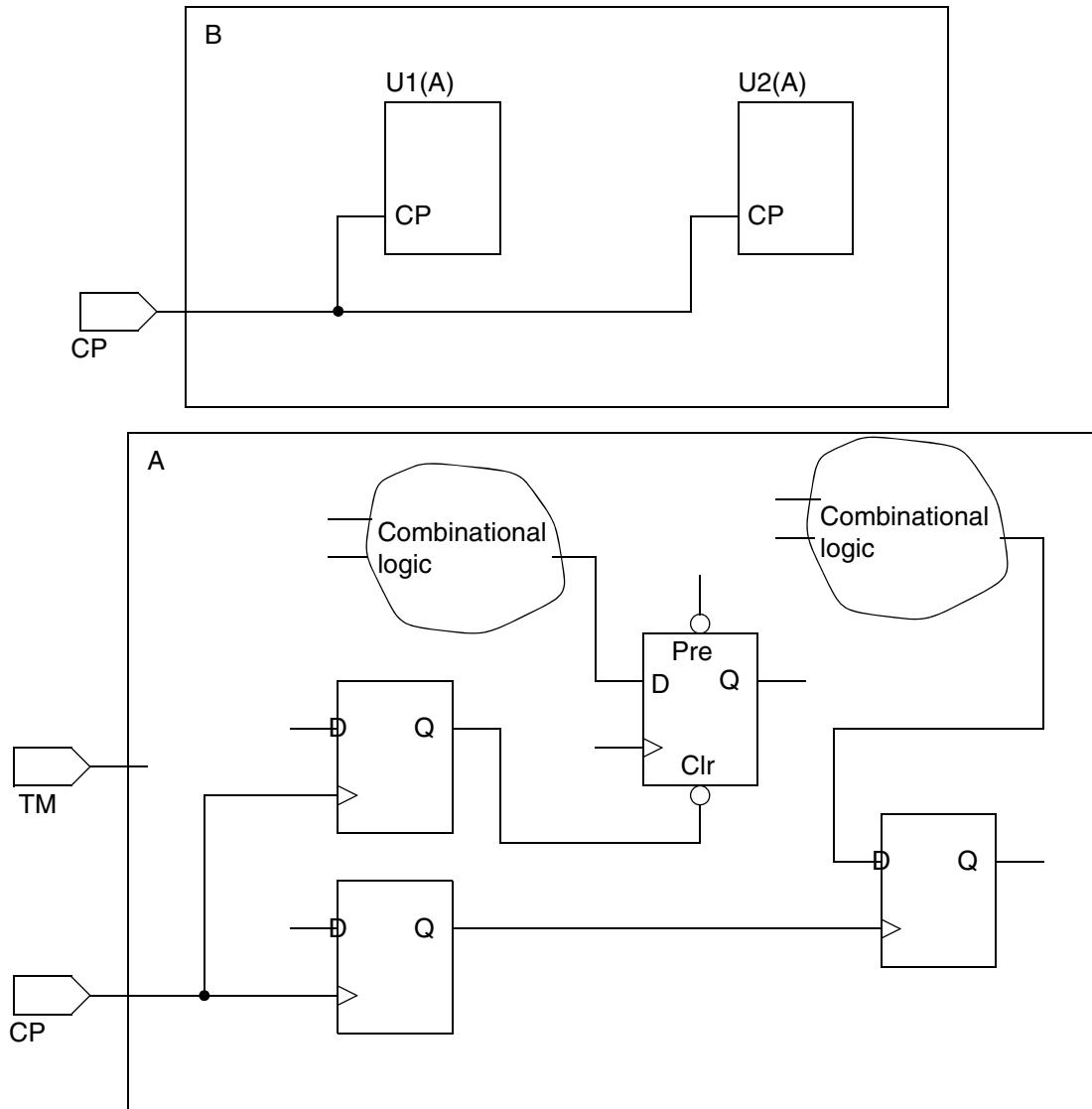
```
dc_shell> current_design TOP
dc_shell> set_scan_configuration -clock_mixing \
           mix_clocks_not_edges
dc_shell> set_dft_configuration -fix_clock enable -fix_reset enable \
           -fix_set enable
dc_shell> set_automix_element UA -type clock -test_data CLKA
dc_shell> set_automix_element UB -type clock -type_data CLKB
dc_shell> set_automix_configuration -type clock -exclude_elements UC
dc_shell> create_test_protocol
dc_shell> preview_dft
dc_shell> insert_dft
```

---

## Bottom-Up Example

To demonstrate a bottom-up approach to scan and test point insertion, [Figure 7-12](#) shows a simplified schematic view of a hierarchy at two levels, B and A.

Figure 7-12 Bottom-Up AutoFix Example



Design B (at the top of the figure) contains two instances of cell A, both of which are clocked by clock CP. Cell A contains two flip-flops clocked by CP. The output of one flip-flop is used to clock some other flip-flops, and the output of the other is used to control the asynchronous clear inputs of other flip-flops. The result is uncontrollable clock violations and uncontrollable asynchronous input violations. Logic that is not relevant to this example is not shown.

Using a bottom-up approach to this design, you perform scan insertion and test point insertion on design A (at the bottom of the figure) first. To ensure that the functional clock CP is used to fix the clock violations, you use the `set_autofix_configuration` command:

```
dc_shell> current_design A
dc_shell> set_autofix_configuration -type clock -test_data CP
```

Note:

Use the `set_autofix_configuration` command if the clock signals causing the clock violations are internally generated. An “internally generated” clock signal is derived sequentially from a clock input alone, such as a clock divider, not in combination with any other inputs.

If there is an existing input port for block A that you want to use as the test mode control input, use the `set_dft_signal` command to specify that signal:

```
dc_shell> set_dft_signal -type TestMode -port MY_TM \
           -hookup_pin my_tm_pin
```

Otherwise, DFT Compiler creates a new, separate input called `test_mode`.

By using the `set_dft_signal` command, you can also set an existing input port for block A that you might want to use as an inverted test mode control input, as shown in the following example:

```
dc_shell> set_dft_signal -type TestMode \
           -port MY_inverted_TM \
           -hookup_pin my_tm_pin -active_state 0
```

Be sure to pay attention to the sense of the inverted test mode.

Use the `preview_dft` command to preview the scan architecture and test point plan, and use the `insert_dft` command to route the scan chains and synthesize the test points:

```
dc_shell> preview_dft
dc_shell> insert_dft
```

To verify that there are no scan violations and to estimate test coverage, constrain the added test mode port before you run the `dft_drc` command:

```
dc_shell> dft_drc -coverage_estimate
```

To stitch the scan chains together at the next-higher level of hierarchy, make the higher-level design the current design and repeat the `preview_dft` and `insert_dft` commands:

```
dc_shell> current_design B  
dc_shell> preview_dft  
dc_shell> insert_dft
```

In this example, DFT Compiler creates a new primary input for test mode control, called `test_mode` by default.

To verify that the scan architecture has no violations and to estimate test coverage, use

```
dc_shell> dft_drc -coverage_estimate
```

---

## Implementing User-Defined Test Points

User-defined test points provide you with the flexibility to insert control and observe test points at user-specified locations in the design. User-defined test points can be used for a variety of purposes, including the ability to fix uncontrollable clocks and asynchronous signals, increase the coverage of the design, and reduce the pattern count.

The following subsections describes the following processes for implementing user-defined test points:

- [Types of User-Defined Test Points](#)
- [Test Point Options](#)
- [User-Defined Test Points Example](#)

---

### Types of User-Defined Test Points

Various types of user-defined test points available for insertion in DFT Compiler are described in the following subsections:

- [Force Test Points](#)
- [Control Test Points](#)
- [Observe Test Points](#)

### Force Test Points

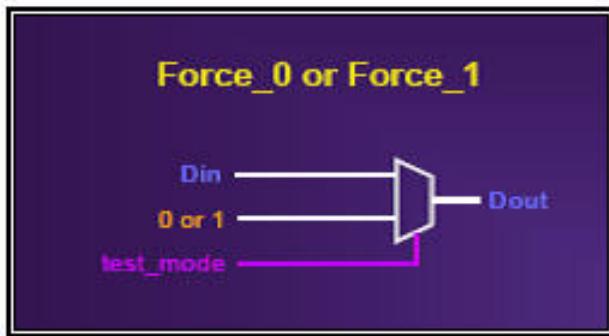
Force test points are used when a value needs to be forced during the entire test session. The following force test points are available:

- `Force_0`
- `Force_1`

- Force\_01
- Force\_z0
- Force\_z1
- Force\_z01

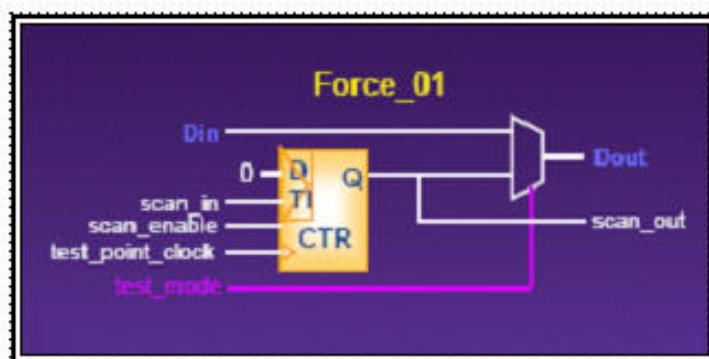
A multiplexer is used with a Force\_0 or Force\_1 test point in cases in which a 0-input is connected to the location where the test point is to be inserted, a 1-input is connected to 0 or 1, and the select line is driven by the TestMode signal. See [Figure 7-13](#).

*Figure 7-13 Example of a Force\_0 or Force\_1 Test Point*



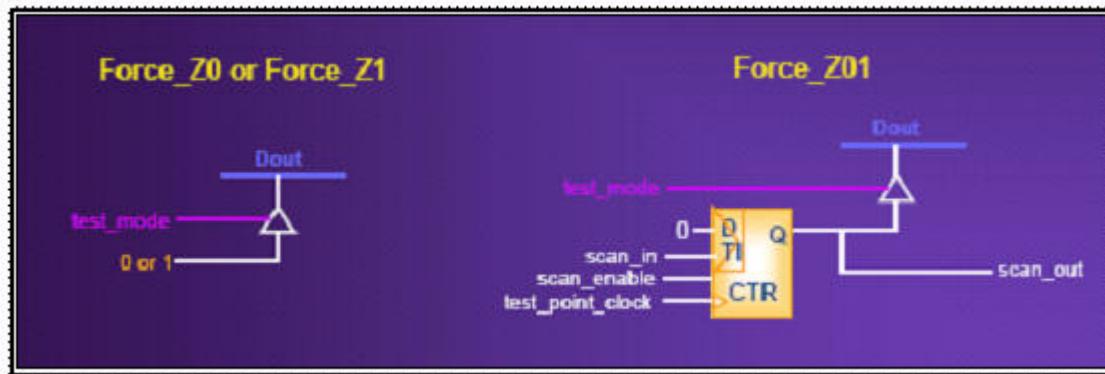
When a Force\_01 test point is being used, one input of the multiplexer is connected to a scan register. See [Figure 7-14](#).

*Figure 7-14 Example of a Force\_01 Test Point*



The Force\_z0, Force\_z1, and Force\_z01 test points are used for three-state signals in a design. See [Figure 7-15](#).

Figure 7-15 Example of Using Force\_z Test Points



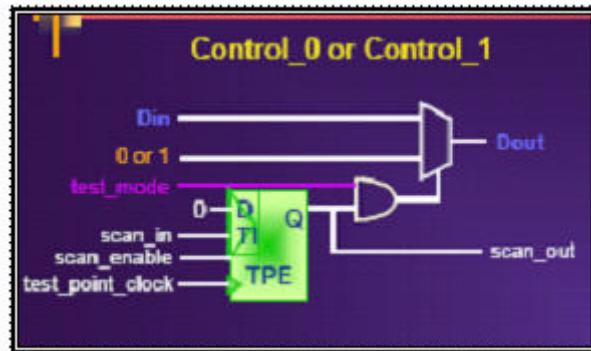
Note that the AutoFix feature of DFT Compiler uses Force\_0 and Force\_1 test points for asynchronous signal fixing and Force\_01 test points for clock fixing and for fixing clock-as-data and X propagation.

## Control Test Points

Control test points are typically inserted within a design to increase the fault coverage of the design.

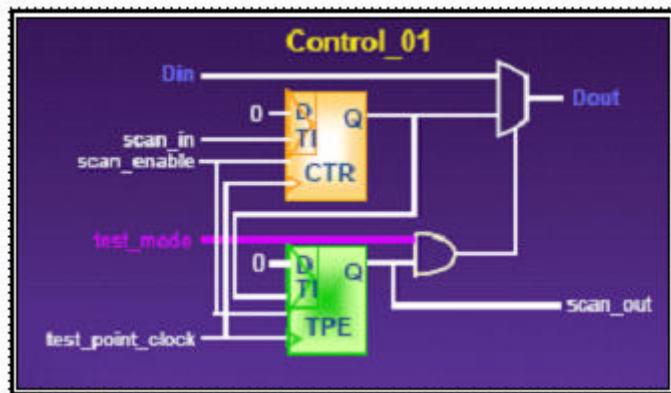
A Control\_0 or Control\_1 test point is built with a multiplexer, an AND gate, and a test point enable (TPE) scan register. The 0-input is the user-specified location where the test point is inserted. The 1-input is usually tied to 1 or 0. The select-input comes from a 2-input AND gate. The inputs of the AND gate are the test mode signal and the output of the test point enable scan register. The AND gate allows you to control the 0 or 1 value during the test session. See [Figure 7-16](#).

Figure 7-16 Control\_0 or Control\_1 Test Point



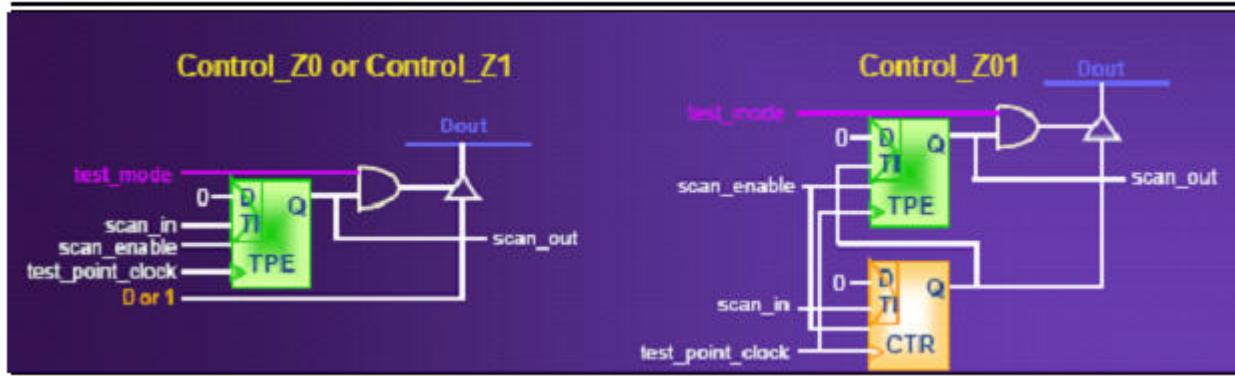
When you use a Control\_01 test point, a “scan control register” is used to control one input of the multiplexer, as Figure 7-17 illustrates.

Figure 7-17 Control\_01 Test Point



The Control\_z0/Control\_z1 and Control\_z01 test points are used for insertion on three-state signals. See Figure 7-18.

Figure 7-18 Control\_z01 Test Point

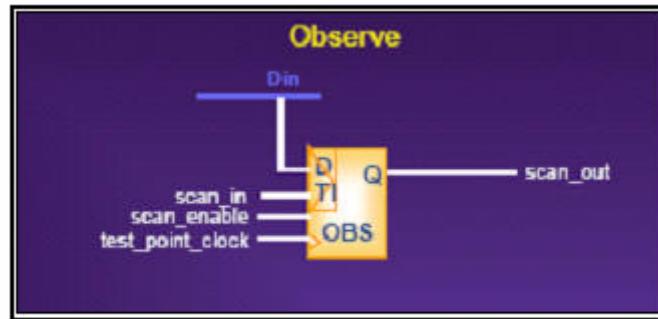


## Observe Test Points

Observe test points are typically inserted at hard-to-observe nodes in a design to reduce test data volume or to increase the coverage.

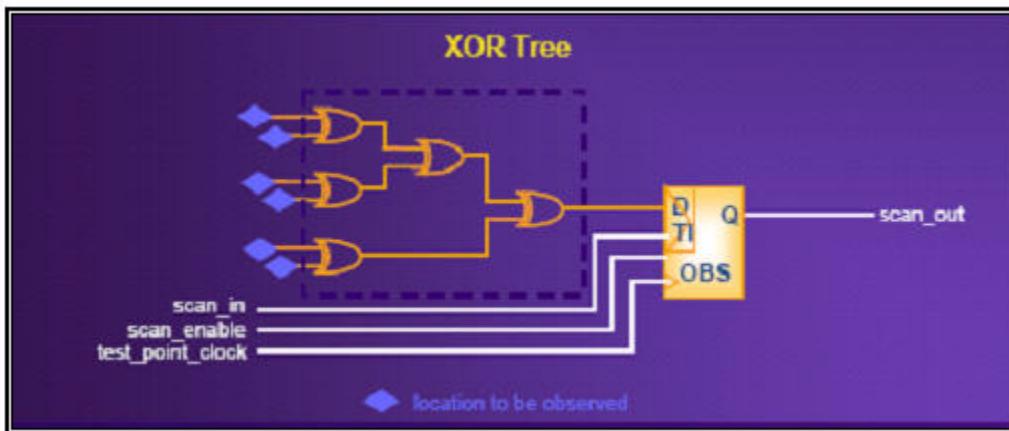
An observe test point is a scan register with its data input connected to the point to be observed. See [Figure 7-19](#).

Figure 7-19 Observe Test Point



Sharing the same observe scan register with more than one observe test point is possible. In such a case, DFT Compiler builds an XOR tree. The output of the XOR tree is connected to the data input of the scan register. The inputs of the XOR gates are connected to the user-specified test point locations. See [Figure 7-20](#).

Figure 7-20 XOR Tree With Observe Test Point



## Test Point Options

The `set_test_point_element` command enables you to specify the location and the type of control point, along with a set of options, in order to achieve your test point requirements.

You can perform the following functions to achieve specific test point requirements:

- Specify a control signal

You can specify an existing port as a control signal, provided that this port has a `TestMode` or `ScanEnable` signal type. If a control signal is not specified, any existing port with a `TestMode` signal type is used as a control signal. DFT Compiler will create a `TestMode` control port type if no such port exists in the design.

- Specify a control clock

An existing clock in the design can be specified as a control clock for observe, control, and test-point-enable scan registers, provided the clock is predefined as a test clock. If a control clock is not specified, DFT Compiler will create a new clock port labeled `tpclk`.

- Specify a source or sink

You can specify the name of the source signal for control points or sink signal for observe points. If source or sink signals are not specified, DFT Compiler will either create a new clock port if `scan_source_or_sink` is disabled or insert a scan register if `scan_source_or_sink` is enabled.

- Specify a test point enable

You can specify a test point enable for control points. The test point enable can be an existing pin or a port.

- Specify the number of test points per test point enable

You can specify the number of test points that can be shared per test-point-enable signal for control test points. This test point signal can be a test-point-enable scan register or a test-point-enable input port. This option is valid only for the control test point types. By default, one test-point-enable signal is used per control test point.

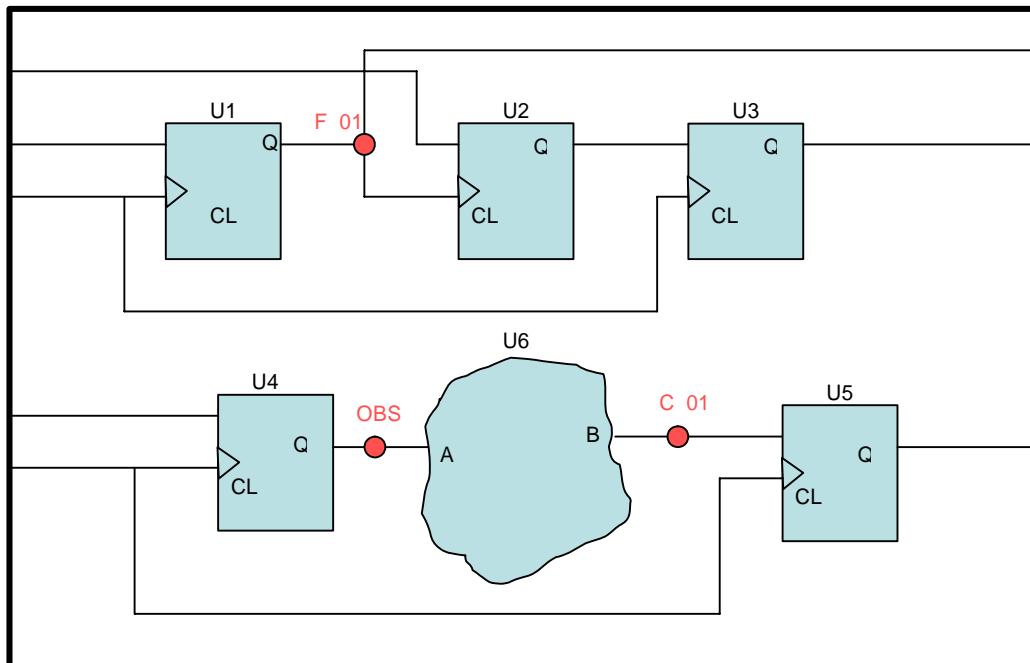
- Specify the power saving option

You can specify the insertion of power saving logic structures (AND gates) in the XOR tree for observe test points. Note that this option is available only for observe test points.

## User-Defined Test Points Example

Consider the simple design shown in [Figure 7-21](#) and the corresponding example using the user-defined test points flow shown in [Example 7-3](#).

*Figure 7-21 An Example Design for User-Defined Test Points*



Note that the design consists of five flip-flops and a cloud of logic that may be untestable. There are two clock ports: CLK1 and CLK2.

The desired locations for user-defined test points are marked by red circles. Next to each red circle is the type of test points that needs to be inserted.

The specification for each user-defined test point type can be made as follows:

- Insert a force\_01 TP at the output of the U1 flip-flop:

```
set_test_point_element -type force_01 U1/Q
```

- Insert an observe TP at the output of the U4 flip-flop, and turn on the power saving option:

```
set_test_point_element -type observe U4/Q -power enable
```

- Insert a control\_01 TP at the input of U5:

```
set_test_point_element -type control_01 U6/B
```

### *Example 7-3 Example UDTP Flow*

```
# Read in the design and synthesize it
read_file -f verilog ./rtl/design.v
current_design TEST
link
compile -scan

# Define the clocks and asynchs in the design
set_dft_signal -view existing_dft -type ScanClock \
-port CLK1 -timing {45 55}
set_dft_signal -view existing_dft -type ScanClock \
-port CLK2 -timing {45 55}
set_dft_signal -view existing_dft -type Reset -port RST \
-active_state 0
set_dft_signal -view spec -type TestMode -port TM
set_scan_configuration -chain_count 10

# Give the UDTP specification. We will specify observe and
# control TPs
# Turn power saving ON for observe test points
set_test_point_element -type observe \
[list alpha_1/Q beta_5/out alpha_3/O] -power_saving enable

# Use an existing clock as clock signal and existing port
# as control signal for control TP
set_test_point_element -type control_1 \
[list U1/u3/omega_2/q alpha_7/Q] -clock_signal CLK2 \
-control_signal TM

# Run pre-DRC
create_test_protocol
dft_drc -verbose
```

```

# Preview the scan chains
preview_dft -show all

# Scan insertion
insert_dft

# Run post-DRC
dft_drc -verbose
report_scan_path

# Write out the netlist
write -hier -f ddc -out TEST_udtp_scan.ddc
change_names -rules verilog
write -hier -f verilog -out TEST_udtp_scan.v

```

---

## Pipelining Scan Enable Architecture

Pipelined scan enables are used for transition delay testing by making use of launch-on-shift (LOS). This method of transition delay testing requires additional circuitry to manipulate the scan enables. Also, the pipelined scan enable signals must be treated as single-cycle paths at the capture clock frequency.

Pipelined scan enables are never used with launch-on-capture (LOC), which does not require additional scan enable circuitry and can use relaxed timing on the scan-enable signals.

These are the differences in the flow when using launch-on-shift (LOS).

Set the following:

```

set test_enable_los_support TRUE
set_scan_configuration -pipeline_scan_enable TRUE
set_scan_configuration -pipeline_fanout_limit <integer>

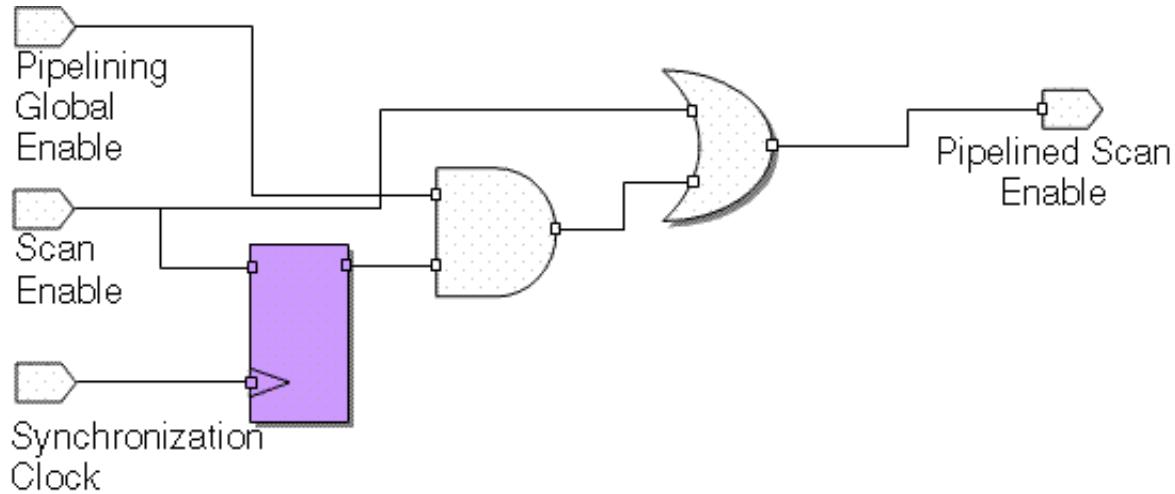
```

**Note1:** `test_enable_los_support` must be set early in the flow, before `create_test_protocol` in order to get a protocol that works for LOS without edits.

**Note2:** `-pipeline_scan_enable TRUE` requires `-clock_mixing no_mix`.

[Figure 7-22](#) shows a representation of the pipelining stage.

Figure 7-22 Pipelined Scan-Enable Architecture



Define the pipeline enable signal. This signal determines the launch mode: launch-on-shift (LOS) when it is active and launch-on-capture (LOC) when it is inactive.

```

set_dft_signal -port pseen -view existing -type LOSPipelineEnable \
    -active_state 1 -test_mode all
set_dft_signal -port pseen -view spec -type LOSPipelineEnable \
    -active_state 1 -test_mode all
    
```

Note: Global pipelining enable is required. The default name is `global_pipe_se`, unless overridden by `set_dft_signal -type LOSPipelineEnable`.

For `dft_drc` runs after `insert_dft`, set the `test_los_drc` variable as shown in the following:

```

set test_los_drc FALSE
dft_drc
set test_los_drc TRUE
dft_drc
    
```

When using the results of `write_test_protocol` in TetraMAX, the `-patternexec` switch must be used. There are two choices:

- `<test_mode>` for LOC
- `<test_mode>_los` for LOS

The `-patternexec` switch is required for the DRC run in TetraMAX. For example, this could be used in TetraMAX for LOS:

```
run drc foo.spf -patternexec Internal_scan_los
```

---

## Using Multimode Scan Architecture

A design's scan interface must accommodate a variety of structural tests. Scan test, burn-in, and other tests performed at various production steps require different types of access to scan elements of a design. To accommodate these different test requirements, multiple scan architectures must be provided on the same scan design. This approach is also useful for complex test schemes such as DFT MAX and core wrapping, which target tests in different parts of a design at different times.

This section explains the process for setting up architectures to perform multimode scans. It includes the following subsections:

- [Reconfiguring Scan Chains](#)
  - [Defining a Test Mode](#)
  - [Assigning Scan Specifications to a Test Mode](#)
  - [Assigning Common Scan Specifications to All Test Modes](#)
  - [Assigning Common Scan Specifications to a Specific Test Mode](#)
  - [Multimode Scan Insertion Script Examples](#)
- 

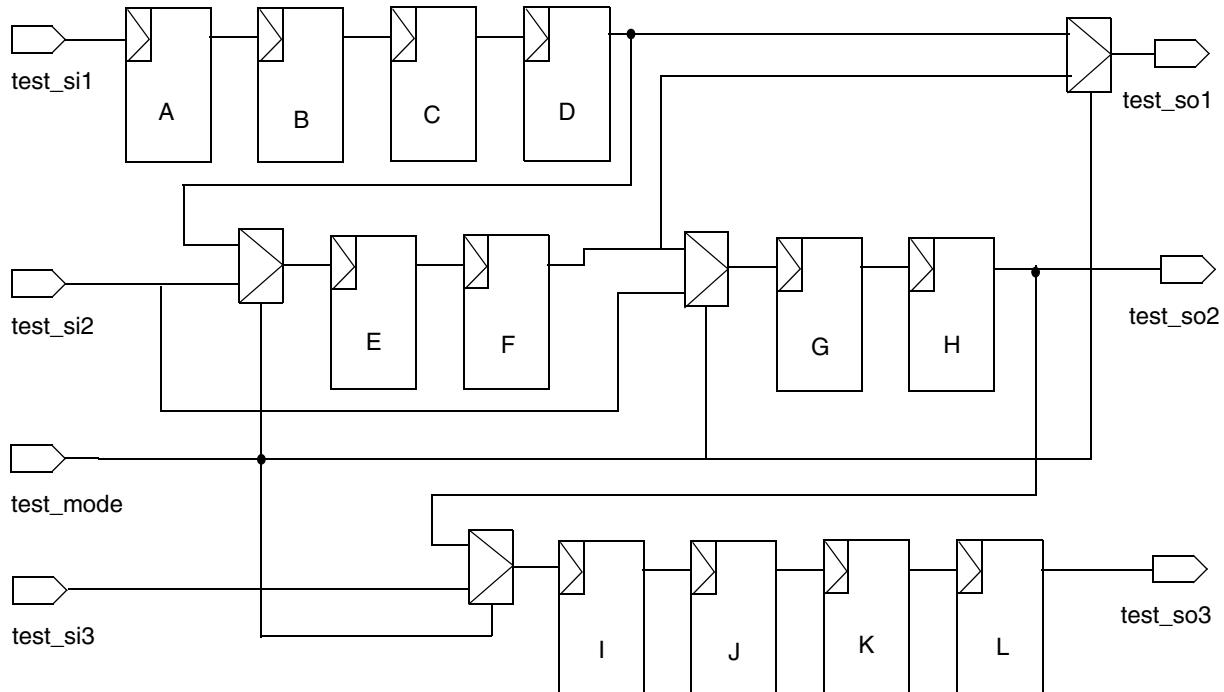
## Reconfiguring Scan Chains

You can reconfigure the scan chains in your design to suit various tester requirements by defining different modes of operation, called *test modes*. For example, suppose you have a simple design with 12 scan cells that must operate in two different scan modes. In one mode, scan data is shifted through two chains (six cells each), and in the other mode, scan data is shifted through three chains (four cells each). [Figure 7-23](#) shows how test mode reconfiguration logic is inserted to support these two scan chain configurations.

Note:

Test mode reconfiguration MUXing logic may appear at any level in your design. This is often dependent on the design level in which scan segments are declared.

Figure 7-23 Configuring Scan Chains With Test Mode Logic



## Defining a Test Mode

You can define different test modes by specifying a test mode label, specifying how the mode will be used, and assigning an encoding value to activate the mode. This is done using the `define_test_mode` command. The syntax for this command is

```
define_test_mode test_mode_name
[-usage mode_usage]
[-encoding <port_name> <0 | 1>]
```

## Defining the Name of a Test Mode

Use the `test_mode_name` label to specify a unique name for your test modes. By default, DFT Compiler will build and name the following test modes:

- Basic scan
  - Internal\_scan – basic scan mode operation

- DFT MAX
  - ScanCompression\_mode – compression scan test
  - Internal\_scan – basic scan mode operation

These modes may be renamed using the `test_mode_name` label as required. Additional modes may also be added.

## Defining the Usage of a Test Mode

Use the `-usage` option specifies how the test mode is to be used. The valid values for the `mode_usage` argument are:

- scan compression – This is the DFT MAX ScanCompression\_mode of operation. In this mode, the internal scan chains are driven by the load compressors, and the scan chains drive the unload compressor. This mode is used for testing all logic internal to the core.
- scan – This is the traditional basic scan mode operation. The scan chains are driven directly by top level scan-in ports, and they drive, in turn, top-level scan-out ports. This mode is used for testing all logic internal to the core.
- wrp\_if – This is the inward facing, or INTEST mode of wrapper operation. This mode is used for testing all logic internal to the core. In this mode, wrappers are enabled and configured to drive and capture data within the design, in conjunction with the internal scan chains.
- wrp\_of – This is the outward facing, or EXTEST mode of wrapper operation. This mode is used for testing all logic external to the design. Wrappers are enabled and configured to drive and capture data outside of the design. In this mode the internal chains are disabled.
- wrp\_safe – This is the safe wrapper mode. In this mode, the internal chains are disabled, and the internal core is protected from any toggle activity. This mode is optional and provides isolation of the core while other cores are being tested. When active, safe mode enables driving steady states into or out of the design.
- bist – In this mode, the internal scan chains are driven by SoCBIST PRPG and these chains, in turn, drive the MISR. It is the main test mode used by SoCBIST. This mode is used for testing all logic internal to the core.
- bist\_controller – This mode of SoCBIST operation is for testing the BIST controller logic. In this mode, all wrapped cores are in the EXTEST mode.
- bist\_scan – This mode of SoCBIST operation is a traditional basic scan mode operation. The scan chains are driven directly by top-level scan-in ports, and they drive, in turn, top-level scan-out ports. This mode is used for testing all logic internal to the core.

## Defining the Encoding of a Test Mode

The `-encoding` option specifies how the test mode port is to be enabled. The syntax of the encoding argument consists of a port name and binary value pair which specify the `test_mode` port and bit value to be used in a particular test mode. These pairs are separated by a comma when multiple ports are used.

Before a port may be used for a test mode port, it must be defined as such through the use of `set_dft_signal -type TestMode`. Example 7-1 and Example 7-2 show specification of test mode ports and the definitions and encoding of test modes which use those ports.

**Identifying Mode Control Ports.** For scan chains to be reconfigurable, dedicated ports must be assigned to drive the reconfigurable logic. These are called test mode ports. Each test mode port can be assigned to a particular test mode, or it can address multiple test modes. If the test mode port controls multiple test modes, either a one hot or a binary decoder is needed.

The following command syntax shows how the decoding style is declared:

```
set_dft_configuration -mode_decoding_style one_hot | binary
```

You must define one test mode port for each test mode when using the `one_hot` decoding style. When you use the `binary` decoding, the number of test mode signals defined must be equal to  $n$  signals which can be used to select  $n^2$  test modes.

**Examples of Test Mode Decoding Styles.** To define a test mode port and indicate which port should be used to drive the scan reconfiguration control logic, use the `set_dft_signal` command, as shown in [Example 7-4](#) and [Example 7-5](#) that follow.

#### *Example 7-4 Specifying One-hot Test Mode Pins*

```
dc_shell> set_dft_signal -view spec \
    -port my_test_mode1 \
    -hookup_pin U1_buff/z \
    -type TestMode \
    -active_state 1
dc_shell> set_dft_signal -view spec \
    -port my_test_mode2 \
    -hookup_pin U2_buff/z \
    -type TestMode \
    -active_state 1
dc_shell> set_dft_signal -view spec \
    -port my_test_mode3 \
    -hookup_pin U3_buff/z \
    -type TestMode \
    -active_state 1
dc_shell> define_test_mode scan_compression1 \
    -usage scan_compression \
    -encoding { my_test_mode1 1 \
        my_test_mode2 0 \
        my_test_mode3 0 }
dc_shell> define_test_mode scan1 \
    -usage scan \
    -encoding { my_test_mode1 0 \
        my_test_mode2 1 \
        my_test_mode3 0 }
dc_shell> define_test_mode scan2 \
    -usage scan \
    -encoding { my_test_mode1 0 \
        my_test_mode2 0 \
        my_test_mode3 1 }
```

In Example 7-4, the one-hot decoding style is used. Three test mode signals are identified: `my_test_mode1`, `my_test_mode2`, and `my_test_mode3`. These ports are connected to pin Z of `U1_buff`, `U2_buff`, and `U3_buff`, respectively.

### *Example 7-5 Specifying Binary Test Mode Pins*

```
dc_shell> set_dft_configuration -mode_decoding_style binary
dc_shell> set_dft_signal -view spec \
    -port my_test_mode1 \
    -hookup_pin U1_buff/z \
    -type TestMode \
    -active_state 1
dc_shell> set_dft_signal -view spec \
    -port my_test_mode2 \
    -hookup_pin U2_buff/z \
    -type TestMode \
    -active_state 1
dc_shell> define_test_mode scan_compression1 \
    -usage scan_compression \
    -encoding { my_test_mode1 0 \
        my_test_mode2 1 }
dc_shell> define_test_mode scan1 \
    -usage scan \
    -encoding { my_test_mode1 1 \
        my_test_mode2 0 }
dc_shell> define_test_mode scan2 \
    -usage scan \
    -encoding { my_test_mode1 1 \
        my_test_mode2 1 }
```

In [Example 7-5](#), the binary decoding style is used. Two test mode signals are identified: `my_test_mode1` and `my_test_mode2`. These ports are connected to pin Z of `U1_buff` and pin Z of `U2_buff`, respectively.

**Error Messages When a Referenced Port Was Not Defined as a Test Mode Port.** If a port has not been specified as a test mode port prior to being used as an encoding port, the following error message will be issued:

### *Example 7-6 Error Message: UIT-1800*

```
Error: <port_name> is not defined as a TestMode port (UIT-1800)
```

If the `TestMode` signal type is not specified for all test mode signals used in the `define_test_mode -encoding` arguments, an error is generated. Specifications of ports and values must done across all modes. Unspecified ports will be detected.

When you have not specified the `TestMode` signal types completely for a port, you see the error messages shown in [Example 7-7](#) and [Example 7-8](#).

### *Example 7-7 Error Message: TEST-1801*

```
Error: Value on test mode port <port_name> not specified in mode <test_mode> (TEST-1801)
```

If two modes have the same encoding, `preview_dft` or `insert_dft` will error out with the following error message:

*Example 7-8 Error Message: TEST-1802*

```
Error: Mode <test_mode> and mode <test_mode> have the same code (TEST-1802)
```

## Defining the Encoding for All Specified Test Modes

You must specify the encoding values for each test mode specified and the values required to enable each on the test mode ports. In the following [Example 7-9](#), the binary decoding style is used. This example shows you how to insert a test mode controller in a basic scan design with the following encoding using pins DBC, IBN, and SBN:

	DBC	IBN	SBN
T0	0	0	0
T1	0	0	1
T2	0	1	0
T3	0	1	1
T4	1	0	0
T5	1	0	1
T6	1	1	0
T7	1	1	1

*Example 7-9 Specifying Basic Scan Encoding Values*

```
dc_shell> set_dft_signal -type TestMode \
           -port [ list DBC IBN SBN] -view spec
dc_shell> define_test_mode T0 -usage scan \
           -encoding {DBC 0 IBN 0 SBN 0}
dc_shell> define_test_mode T1 -usage scan \
           -encoding {DBC 0 IBN 0 SBN 1}
dc_shell> define_test_mode T2 -usage scan \
           -encoding {DBC 0 IBN 1 SBN 0}
dc_shell> define_test_mode T3 -usage scan \
           -encoding {DBC 0 IBN 1 SBN 1}
dc_shell> define_test_mode T4 -usage scan \
           -encoding {DBC 1 IBN 0 SBN 0}
dc_shell> define_test_mode T5 -usage scan \
           -encoding {DBC 1 IBN 0 SBN 1}
```

```

dc_shell> define_test_mode T6 -usage scan \
           -encoding {DBC 1 IBM 1 SBN 0}
dc_shell> define_test_mode T7 -usage scan \
           -encoding {DBC 1 IBM 1 SBN 0}

```

Specifications of encoding must be done for all modes defined with `define_test_mode`. Any mode that does not have encoding specified causes `preview_dft` and `insert_dft` to error out with the following message:

*Example 7-10 Error Message: Test 1804*

Error: Encoding for test mode <test\_mode> is not specified. (TEST-1804)

Any mode that redefines encoding will cause `preview_dft` and `insert_dft` to issue the warning shown in [Example 7-11](#). The encoding data of a `define_test_mode` command which is issued overrides any previous encoding data for that mode.

*Example 7-11 Warning Message: UIT 1803*

Warning: Overriding encoding data for test mode <test\_mode> (UIT-1803)

In [Example 7-12](#), the binary decoding style is used again. This example shows you how to insert a test mode controller in a DFT MAX design with one `ScanCompression_mode` and two `Internal_scan` modes, using the following encoding on pins `i_trdy_de`, `i_trdy_dd`, and `i_cs`:

	<code>i_trdy_de</code>	<code>i_trdy_dd</code>	<code>i_cs</code>
<code>ScanCompression_mode</code>	0	0	1
<code>Internal_scan1</code>	0	1	0
<code>Internal_scan2</code>	0	1	1

*Example 7-12 Specifying DFT MAX Encoding Values*

```

dc_shell> set_dft_signal -view spec -type TestMode \
           -port [list i_trdy_de i_trdy_dd i_cs]
dc_shell> define_test_mode ScanCompression_mode \
           -usage scan_compression \
           -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1}
dc_shell> define_test_mode Internal_scan1 \
           -usage scan \
           -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0}
dc_shell> define_test_mode Internal_scan2 \
           -usage scan \
           -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1}

```

## Examining the Encoding in the Preview Report

This section includes a sample report section created from the `preview_dft -show all` command. The report includes information encoding for each test mode that you defined in [Example 7-12](#). This information comes at the end of the `preview_dft` report and is shown in [Example 7-13](#).

### *Example 7-13 Sample of a preview\_dft Report*

```
dc_shell> preview_dft -show all
*****
Dft signals:
TestMode: i_trdy_de (no hookup pin)
TestMode: i_trdy_dd (no hookup pin)
TestMode: i_cs (no hookup pin)
=====
Test Mode Controller Information
=====

Test Mode Controller Ports
-----
test_mode: i_trdy_de
test_mode: i_trdy_dd
test_mode: i_cs

Test Mode Controller Index (MSB --> LSB)
-----
i_trdy_de, i_trdy_dd, i_cs

Control signal value - Test Mode
-----
011 Internal_scan2 - InternalTest
001 ScanCompression_mode - InternalTest
010 Internal_scan1 - InternalTest
1
```

---

## Assigning Scan Specifications to a Test Mode

You can assign scan specifications to a test mode in one of two methods:

- The first method is implicit or context-dependent. In this method, you set the test mode context first, and then enter scan specification commands associated with that test mode. For example:

```
current_test_mode ManufactTest
```

```
set_scan_configuration -chain_count 16
```

- The second method is explicit or context independent. In this method, you add the `-test_mode test_mode_name` option to the scan specification command. For example:

```
set_scan_configuration -chain_count 1 -test_mode \
BurnInTest
```

You can use the following scan configuration commands in the multimode environment:

```
set_scan_configuration
set_scan_path
set_dft_signal
```

For the `set_scan_configuration` command, the following options can be specified for a test mode (note that all options of `set_scan_configuration` can be specified for a test mode):

```
-chain_count int
-max_length int
-clock_mixing no_mix | mix_clocks | mix_edges |
mix_clocks_not_edges
```

For the `set_scan_path` command, the following options can be specified for a test mode (note that all options of `set_scan_path` can be specified for a test mode):

```
-scan_master_clock clock_name
-exact_length int
```

---

## Assigning Common Scan Specifications to All Test Modes

In some cases, you may have a scan specification command that applies to all test modes. For example, all test modes might use a common clock or a test scan in port called `my_si`. You can apply this scan specification command to all test modes by adding the `-test_mode all` option at the end of the command.

For the common scan example, the syntax is

```
set_dft_signal -view spec -test_mode all -port my_si \
-type ScanDataIn
```

You cannot apply a common scan specification command to all test modes by setting the `current_test_mode all`. This command can only point to one mode at a time.

You can apply a scan specification to all but one test mode by applying the specification to all test modes first by using `-test_mode all` and then specifying a second command for the other test mode, referencing it by the specific mode name. For example, suppose that all test modes except `BurnInTest` have six scan chains and that `BurnInTest` has one scan chain. The command sequence for this case is shown in [Example 7-14](#).

*Example 7-14 The Effect of Using `-test_mode all`*

```
dc_shell> set_scan_configuration -chain_count 6 \
           -test_mode all
dc_shell> set_scan_configuration -chain_count 1 \
           -test_mode BurnInTest
```

Scan specifications are considered globally if `-test_mode all` is used except when a specific test mode is called out by using the specific test mode name as defined in `define_test_mode`.

The `set_scan_configuration` command is cumulative and will overwrite previous specifications of the same type. In [Example 7-15](#), suppose you want the `BurnInTest` test mode to have one scan chain defined and all other test modes to have six. Note that you have to define the chain count twice for `BurnInTest`. This example shows the cumulative nature of the `set_scan_configuration` command.

*Example 7-15 Overwriting Mode specifications*

```
dc_shell> set_scan_configuration -chain_count 1 \
           -test_mode BurnInTest
dc_shell> set_scan_configuration -chain_count 6 \
           -test_mode all
dc_shell> set_scan_configuration -chain_count 2 \
           -test_mode BurnInTest
```

In [Example 7-15](#), two chains are built in `BurnInTest` mode instead of just one because the `set_scan_configuration` command is cumulative. The second directive for `BurnInTest` is the one the tool honors. Apart from the `BurnInTest` mode, the other modes will have six chains. As long as you remember the cumulative nature of these commands, you can issue them in any order.

When specifying scan paths, remember that `-test_mode all` applies across all modes. Specifying a path for a particular mode applies only to that mode. See [Example 7-16](#).

*Example 7-16 Nonconflicting Mode Specifications*

```
dc_shell> set_scan_path Chain1 -test_mode all_dft \
           -view spec
dc_shell> set_scan_path Chain2 \
           -test_mode ManuFactTest -view spec
```

In Example 7-16, the ManuFactTest mode uses two user-defined scan chains, Chain1 and Chain2, because all scan specifications support inheritance from the `all` mode. However, all other scan specifications options and commands are independent of the mode.

---

## Assigning Common Scan Specifications to a Specific Test Mode

When applying scan specifications in a multimode environment, you should carry out these steps in this order:

1. Define TestMode signals with `set_dft_signal -test_mode all`.
2. Define all test modes, their usage and encoding with `define_test_mode`.
3. Define clocks, asynchronous signals and constants that are common to all test modes with `set_dft_signal -test_mode all`.
4. Define any mode specific signals with `set_dft_signal -test_mode test_mode_name`.
5. Specify any scan paths to be used in all test modes with `set_scan_path -test_mode all`.
6. Specify any scan paths to be used in specific test modes with `set_scan_path -test_mode test_mode_name`.

Note: when the `define_test_mode` command is used, it sets the `current_test_mode`. Because of this behavior, if any of the following commands is run without using the `-test_mode` switch, the command will not apply the specification to all the test modes (the usual situation) but will instead apply the specification only to the mode that was last defined with the `define_test_mode` command. The commands are

- `set_scan_configuration`
- `set_scan_path`
- `set_dft_signal`
- `set_scan_compression_configuration`

Be sure to use the `-test_mode` switch with all these commands once the `define_test_mode` command has been issued. If the specification is to be applied to all modes, use `-test_mode all`. If the specification is to be applied to a particular mode only, use `-text_mode mode_name` to apply the specification to the named mode.

---

## Multimode Scan Insertion Script Examples

This section provides examples of basic scan, DFT MAX, and core wrapping scripts in a multimode environment.

[Example 7-17](#) shows a basic scan script that defines four scan test modes. The scan1 mode has one chain, the scan2 mode has two chains, the scan3 mode has four chains and the scan 4 mode has eight chains. Each set of chains uses separate scan-in and scan-out pins.

[Example 7-18](#) shows a DFT MAX script. In this script, three test modes are created. One mode is used for compression testing, a second mode is used for basic scan test and has eight chains, and a third mode uses a single basic scan chain.

[Example 7-19](#) shows a Core Wrapper script. This script defines all the modes that will be created in a wrapper insertion process, which supports at-speed test and shared wrapper cells.

### *Example 7-17 Basic Scan Multimode Script*

```
## Define the pins for use in any test mode with "test_mode all_dft".
for {set i 1} {$i <= 15 } { incr i 1} {
    create_port -direction in test_si[$i]
    create_port -direction out test_so[$i]
    set_dft_signal -type ScanDataIn -view spec -port test_si[$i] -test_mode all
    set_dft_signal -type ScanDataOut -view spec -port test_so[$i] -test_mode all
}

#Define Test Clocks
set_dft_signal -view existing_dft -type TestClock -timing {45 55} -port clk_st

#Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port [list i_trdy_de i_trdy_dd i_cs]

#Define the test modes
define_test_mode scan1 -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1}
define_test_mode scan2 -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0}
define_test_mode scan3 -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1}
define_test_mode scan4 -usage scan \
    -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0}

#Configure the basic scan modes
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
    -test_mode scan1
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks \
    -test_mode scan2
set_scan_configuration -chain_count 4 -clock_mixing mix_clocks \
    -test_mode scan3
set_scan_configuration -chain_count 8 -clock_mixing mix_clocks \
    -test_mode scan4
```

```

## Give a chain spec to be applied in each of the modes
set_scan_path chain1 -view spec -scan_data_in test_si[1] \
    -scan_data_out test_so[1] \
    -test_mode scan1

set_scan_path chain2 -view spec -scan_data_in test_si[2] \
    -scan_data_out test_so[2] \
    -test_mode scan2

set_scan_path chain3 -view spec -scan_data_in test_si[3] \
    -scan_data_out test_so[3] \
    -test_mode scan2

set_scan_path chain4 -view spec -scan_data_in test_si[4] \
    -scan_data_out test_so[4] \
    -test_mode scan3

set_scan_path chain5 -view spec -scan_data_in test_si[5] \
    -scan_data_out test_so[5] \
    -test_mode scan3

set_scan_path chain6 -view spec -scan_data_in test_si[6] \
    -scan_data_out test_so[6] \
    -test_mode scan3

set_scan_path chain7 -view spec -scan_data_in test_si[7] \
    -scan_data_out test_so[7] \
    -test_mode scan3

set_scan_path chain8 -view spec -scan_data_in test_si[8] \
    -scan_data_out test_so[8] \
    -test_mode scan4

set_scan_path chain9 -view spec -scan_data_in test_si[9] \
    -scan_data_out test_so[9] \
    -test_mode scan4

set_scan_path chain10 -view spec -scan_data_in test_si[10] \
    -scan_data_out test_so[10] \
    -test_mode scan4

set_scan_path chain11 -view spec -scan_data_in test_si[11] \
    -scan_data_out test_so[11] \
    -test_mode scan4

set_scan_path chain12 -view spec -scan_data_in test_si[12] \
    -scan_data_out test_so[12] \
    -test_mode scan4

set_scan_path chain13 -view spec -scan_data_in test_si[13] \
    -scan_data_out test_so[13] \
    -test_mode scan4

set_scan_path chain14 -view spec -scan_data_in test_si[14] \
    -scan_data_out test_so[14] \
    -test_mode scan4

set_scan_path chain15 -view spec -scan_data_in test_si[15] \

```

```

-scan_data_out test_so[15] \
-test_mode scan4

set_dft_insertion_configuration -synthesis_optimization none

create_test_protocol
dft_drc
preview_dft -show all

insert_dft

current_test_mode scan1
dft_drc -v

current_test_mode scan2
dft_drc -v

current_test_mode scan3
dft_drc -v

current_test_mode scan4
dft_drc -v

list_test_modes

list_license
change_names -rule verilog -hier
write -format verilog -hier -output vg/top_scan.v
write_test_protocol -test_mode scan1 -out stil/scan1.stil -names verilog
write_test_protocol -test_mode scan2 -out stil/scan2.stil -names verilog
write_test_protocol -test_mode scan3 -out stil/scan3.stil -names verilog
write_test_protocol -test_mode scan4 -out stil/scan4.stil -names verilog

exit

```

*Example 7-18 Basic DFT MAX Multimode Script*

```

## Define the pins for compression/base_mode using "test_mode all_dft".
## These modes are my_comp and my_scan1
for {set i 1} {$i <= 13 } { incr i 1} {
    create_port -direction in test_si[$i]
    create_port -direction out test_so[$i]
    set_dft_signal -type ScanDataIn -view spec -port test_si[$i] -test_mode all
    set_dft_signal -type ScanDataOut -view spec -port test_so[$i] -test_mode all
}
#Define Test Clocks
set_dft_signal -view existing_dft -type TestClock -timing {45 55} -port clk_st

#Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port [list i_trdy_de i_trdy_dd i_cs]

#Define the test modes and usage
define_test_mode my_base1 -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1}
define_test_mode my_base2 -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0}
define_test_mode burn_in -usage scan \

```

```

        -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1}
define_test_mode scan_compression1 -usage scan_compression \
        -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0}
define_test_mode scan_compression2 -usage scan_compression \
        -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 1}

#Enable DFT MAX
set_dft_configuration -scan_compression enable

#Configure DFT MAX
set_scan_compression_configuration -base_mode my_base1 -chain_count 32 -test_mode
scan_compression1 -xtol high
set_scan_compression_configuration -base_mode my_base2 -chain_count 256 -test_mode
scan_compression2 -xtol high

#Configure the basic scan modes
set_scan_configuration -chain_count 4 -test_mode my_base1
set_scan_configuration -chain_count 8 -test_mode my_base2
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks -test_mode burn_in

set_dft_insertion_configuration -synthesis_optimization none

## Give a chain spec to be applied in my_base1
## This will also define the scan ports for scan_compression1
set_scan_path chain1 -view spec -scan_data_in test_si[1] \
        -scan_data_out test_so[1] \
        -test_mode my_base1
set_scan_path chain2 -view spec -scan_data_in test_si[2] \
        -scan_data_out test_so[2] \
        -test_mode my_base1
set_scan_path chain3 -view spec -scan_data_in test_si[3] \
        -scan_data_out test_so[3] \
        -test_mode my_base1
set_scan_path chain4 -view spec -scan_data_in test_si[4] \
        -scan_data_out test_so[4] \
        -test_mode my_base1

## Give a chain spec to be applied in my_base2
## This will also define the scan ports for scan_compression2
set_scan_path chain5 -view spec -scan_data_in test_si[5] \
        -scan_data_out test_so[5] \
        -test_mode my_base2
set_scan_path chain6 -view spec -scan_data_in test_si[6] \
        -scan_data_out test_so[6] \
        -test_mode my_base2
set_scan_path chain7 -view spec -scan_data_in test_si[7] \
        -scan_data_out test_so[7] \
        -test_mode my_base2
set_scan_path chain8 -view spec -scan_data_in test_si[8] \
        -scan_data_out test_so[8] \
        -test_mode my_base2
set_scan_path chain9 -view spec -scan_data_in test_si[9] \
        -scan_data_out test_so[9] \
        -test_mode my_base2
set_scan_path chain10 -view spec -scan_data_in test_si[10] \
        -scan_data_out test_so[10] \
        -test_mode my_base2
set_scan_path chain11 -view spec -scan_data_in test_si[11] \

```

```

-scan_data_out test_so[11] \
-test_mode my_base2
set_scan_path chain12 -view spec -scan_data_in test_si[12] \
-scan_data_out test_so[12] \
-test_mode my_base2

## Give a chain spec to be applied in burn_in
set_scan_path chain4 -view spec -scan_data_in test_si[13] \
-scan_data_out test_so[13] \
-test_mode burn_in

create_test_protocol
dft_drc
preview_dft -show all

insert_dft

list_test_modes

current_test_mode scan_compression1
report_dft_signal
dft_drc -v

current_test_mode scan_compression2
report_dft_signal
dft_drc -v

current_test_mode my_base1
report_dft_signal
dft_drc -v

current_test_mode my_base2
report_dft_signal
dft_drc -v

current_test_mode burn_in
report_dft_signal
dft_drc -v

change_names -rule verilog -hier
write -format verilog -hier -output vg/10x_xtol_moxie_top_scan_mm.v
write_test_protocol -test_mode scan_compression1 -out stil/scan_compression1.stil \
-names verilog
write_test_protocol -test_mode scan_compression2 -out stil/scan_compression2.stil \
-names verilog
write_test_protocol -test_mode my_base1 -out stil/my_base1.stil \
-names verilog
write_test_protocol -test_mode my_base2 -out stil/my_base2.stil \
-names verilog
write_test_protocol -test_mode burn_in -out stil/10x_xtol_moxie.burn_in.stil \
-names verilog

exit

```

*Example 7-19 Basic Core Wrapper Multimode Script*

```
#Define Test Clocks
```

```

set_dft_signal -view existing_dft -type TestClock -timing {45 55} -port clk_s

#Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port [list i_trdy_de i_trdy_dd i_cs i_wr]

#Define the test modes and usage
define_test_mode burn_in -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 0 i_wr 1}
define_test_mode domain -usage scan \
    -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1 i_wr 0}
define_test_mode my_wrp_if -usage wrp_if \
    -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1 i_wr 1}
define_test_mode my_wrp_if_delay -usage wrp_if \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0 i_wr 0}
define_test_mode my_wrp_if_scl_delay -usage wrp_if \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0 i_wr 1}
define_test_mode my_wrp_of -usage wrp_of \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1 i_wr 0}
define_test_mode my_wrp_of_delay -usage wrp_of \
    -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1 i_wr 1}
define_test_mode my_wrp_of_scl_delay -usage wrp_of \
    -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0 i_wr 0}
define_test_mode my_wrp_safe -usage wrp_safe \
    -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0 i_wr 1}

#Set scan chain count as desired
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks -test_mode burn_in
set_scan_configuration -chain_count 5 -test_mode domain
set_scan_configuration -chain_count 8 -test_mode wrp_if
set_scan_configuration -chain_count 8 -test_mode wrp_of

# Enable and configure wrapper client
set_dft_configuration -wrapper enable

#Configure for shared wrappers, using existing cells and \
#                           create glue logic around existing cells
set_wrapper_configuration -class core_wrapper \
    -style shared \
    -shared_cell_type WC_S1 \
    -use_dedicated_wrapper_clock true \
    -safe_state 1 \
    -register_io_implementation in_place \
    -delay_test true

#Create the test protocol and run pre-drc
create_test_protocol
dft_drc -v

#Report the configuration of the wrapper utility, optional
report_wrapper_configuration

preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all

report_dft_configuration

#Run scan insertion and wrap the design

```

```

set_dft_insertion_configuration -synthesis_optimization none
insert_dft

list_test_modes

current_test_mode burn_in
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_burn_in.rpt

current_test_mode domain
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_domain.rpt

current_test_mode my_wrp_of
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_of.rpt

current_test_mode my_wrp_of_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_of_delay.rpt

current_test_mode my_wrp_of_scl_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_of_scl_delay.rpt

current_test_mode my_wrp_if
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_if.rpt

current_test_mode my_wrp_if_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_if_delay.rpt

current_test_mode my_wrp_if_scl_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_if_scl_delay.rpt

report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hier

write -format ddc -hier -output db/scan.ddc
write -format verilog -hier -output vg/scan_wrap.vg
write_test_model -o db/des_unit.scan.ctldb
write_test_protocol -test_mode burn_in -o stil/burn_in.spf
write_test_protocol -test_mode domain -o stil/domain.spf
write_test_protocol -test_mode my_wrp_if_delay -o stil/my_wrp_if_delay.spf
write_test_protocol -test_mode my_wrp_if_scl_delay -o stil/my_wrp_if_scl_delay.spf
write_test_protocol -test_mode my_wrp_if -o stil/my_wrp_if.spf
write_test_protocol -test_mode my_wrp_of -o stil/my_wrp_of.spf
write_test_protocol -test_mode my_wrp_of_delay -o stil/my_wrp_of_delay.spf
write_test_protocol -test_mode my_wrp_of_scl_delay -o stil/my_wrp_of_scl_delay.spf
write_test_protocol -test_mode wrp_if -o stil/extrawrp_if.spf

exit

```

Note that you can also run the `report_scan_path -test_mode tms` command, which displays a report containing test-related information about the current design, as shown in [Example 7-20](#).

*Example 7-20 Sample report\_scan\_path Output*

```
dc_shell> report_scan_path -test_mode tms
Number of chains: 3
Scan methodology: full_scan
Scan style: multiplexed_flip_flop

Clock domain: mix_clocks
Chn Scn Prts (si --> so) #Cell Inst/Chain Clck (prt, tm,edge)
-----
S 1 test_si1 --> test_so1 2 U2/1 (s) CK3, 45.0,rising)
S 2 test_si2 --> test_so2 2 U2/2 (s) (CK2, 45.0,rising)
W 3 test_si3 --> test_so3 8 U2/WrapperChain_0 (s) (WCLK,45.0, rising)
```

---

## Multivoltage Support

The increasing presence of multiple voltages in designs has resulted in the need for DFT insertion to build working scan chains with minimal voltage crossings and minimal level shifters. This section describes the methodology for running DFT insertion in designs containing multiple voltages.

The following subsections describe multivoltage support:

- [Configuring Scan Insertion for Multivoltagess](#)
- [Configuring Scan Insertion for Multiple Power Domains](#)
- [Mixture of Multivoltagess and Multiple Power Domain Specifications](#)
- [Reusing a Multivoltage Cell](#)
- [DFT Considerations for Low-Power Design Flows](#)
- [Previewing a Multivoltage Scan Chain](#)
- [Running in UPF Mode](#)
- [Limitations](#)

---

## Configuring Scan Insertion for Multivolts

The `set_scan_configuration` command instructs the DFT insertion process to build scan chains that could cross different voltage regions. Using following option minimizes the number of voltage crossings:

```
dc_shell> set_scan_configuration -voltage_mixing true
```

Note:

Voltage crossings are minimized to make sure only necessary level shifters are added.

The default is false.

After running this command, proceed through the normal scan insertion flow. Any necessary level shifters will be automatically inserted after scan insertion. You can also use the `insert_level_shifters` command to insert level shifters between voltage crossings, as in following example:

```
dc_shell> insert_level_shifters
```

---

## Conflicting Voltage Mixing Specifications

If you use any commands, such as `set_scan_path`, that violate the voltage mixing specification, the `preview_dft` and `insert_dft` commands issue the following warning message and continue with scan insertion:

```
Warning: elements are supplied by different voltages. (TEST-1026)
```

---

## Configuring Scan Insertion for Multiple Power Domains

The following command instructs the DFT insertion process to build scan chains that could cross different power domains:

```
dc_shell> set_scan_configuration \
           -power_domain_mixing true
```

Note:

Power domain crossings are minimized to make sure the required number of isolation cells are minimized.

The default is false.

After running this command, proceed through the normal scan insertion flow. However, the required isolation cells are not inserted by the `insert_dft` command. You can use the `insert_isolation_cells` command to insert isolation cells between power domain crossings.

Scan insertion does not check or remove existing isolation cells.

## Conflicting Power Domain Mixing Specifications

If you use any commands, such as `set_scan_path`, that violate the power domain mixing specification, the `preview_dft` and `insert_dft` commands issue the following warning message and continue with scan insertion:

```
Warning: elements are supplied by different power domains.  
(TEST-1029)
```

After scan insertion, if you are not running in UPF mode, inserting isolation cells is required.

## Generating a SCANDEF File in the Presence of Isolation Cells

After DFT and isolation cell insertion and before you generate a SCANDEF file, you must specify any constraints that are needed at the enable pin of the isolation cells to generate the correct SCANDEF file.

## Scan Extraction Flows in the Presence of Isolation Cells

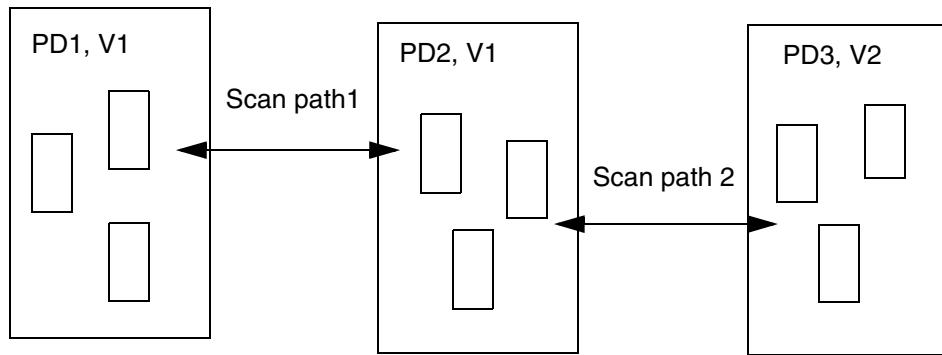
To run the scan extraction flow, specify any constraints that are needed at the enable pin of isolation cells in addition to the constraints that may be required for DFT signals. If this not done, you might fail to extract scan chains. DFT Compiler does not check the validity of isolation logic.

---

## Mixture of Multivoltages and Multiple Power Domain Specifications

The interaction between `-voltage_mixing` and `-power_domain_mixing` options is as shown in [Figure 7-24](#).

*Figure 7-24 Interaction Between Voltage Mixing and Power Domain Mixing*



Here the scan cells are contained within three blocks as follows:

- Power domain PD1, Voltage V1
- Power domain PD2, Voltage V1
- Power domain PD3, Voltage V2

By default, DFT Compiler does not allow voltage mixing or power domain mixing within the same scan path. [Table 7-1](#) shows the allowable scan paths with the various combinations of `-voltage_mixing` and `-power_domain_mixing`:

*Table 7-1 Allowable Scan Paths for Different Values of Voltage Mixing and Power Domain Mixing*

If <code>-voltage_mixing</code> is:	And <code>-power_domain_mixing</code> is:	Allowable scan paths
False	False	None
False	True	Scan path 1 only
True	False	None
True	True	Scan paths 1 and 2

---

## Reusing a Multivoltage Cell

By default, `insert_dft` will reuse existing level shifters and isolation cells if they are on the scan path. It will reuse only combinational multivoltage cells and will not reuse sequential multivoltage cells, such as latch-based isolation cells. If you do not want `insert_dft` to reuse existing multivoltage cells (level shifters and isolation cells), use the following command:

```
dc_shell> set_scan_configuration \
           -reuse_mv_cells false
```

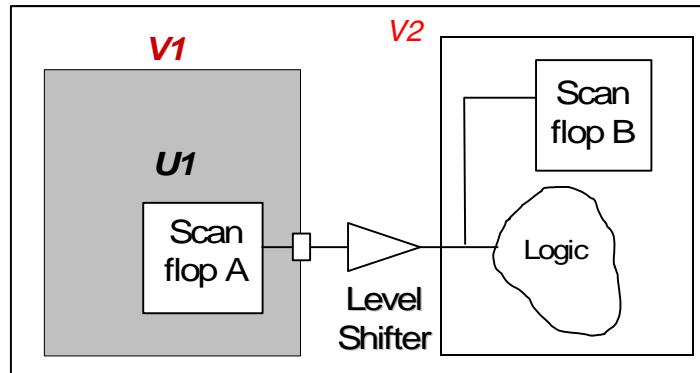
This section includes the following subsections:

- [Level Shifter as Part of Scan Path](#)
- [Isolation Cell as Part of Scan Path](#)

### Level Shifter as Part of Scan Path

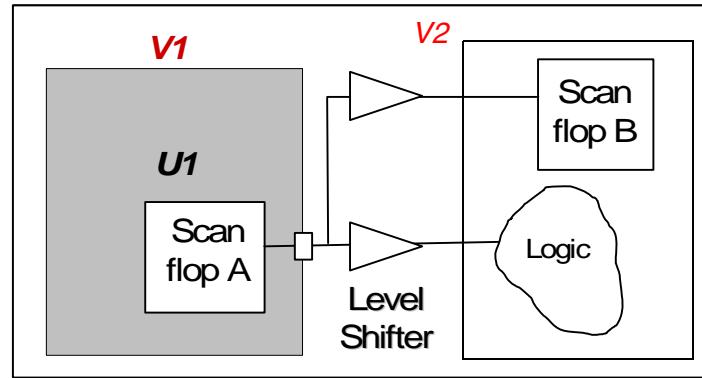
If a scan path goes through a level shifter and you reuse the existing level shifter, scan insertion connects the scan chain at the output side of the level shifter, as shown in [Figure 7-25](#).

*Figure 7-25 Existing Level Shifter as Part of Scan Path*



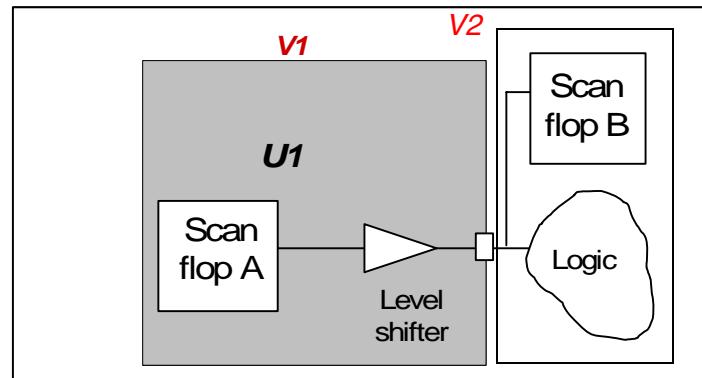
If the scan path goes through a level shifter and you do not reuse the existing level shifter, a new level shifter is added to connect to the scan path without creating a new hierarchical port. See [Figure 7-26](#).

Figure 7-26 Level Shifter as Part of Scan Path



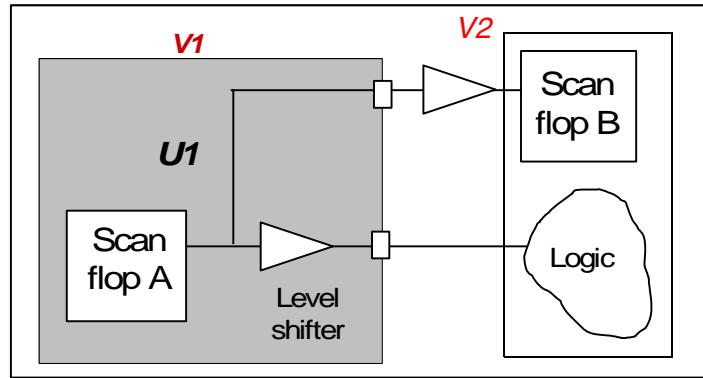
If the level shifter is within a block and you reuse the existing level shifter, then scan insertion reuses the existing hierarchical port, as shown in Figure 7-27.

Figure 7-27 Existing Level Shifter in a Block as Part of Scan Path



If you choose not to reuse existing level shifters and the existing level shifter is within a block, then the new level shifter is added within the block and a new hierarchical port is added. See Figure 7-28.

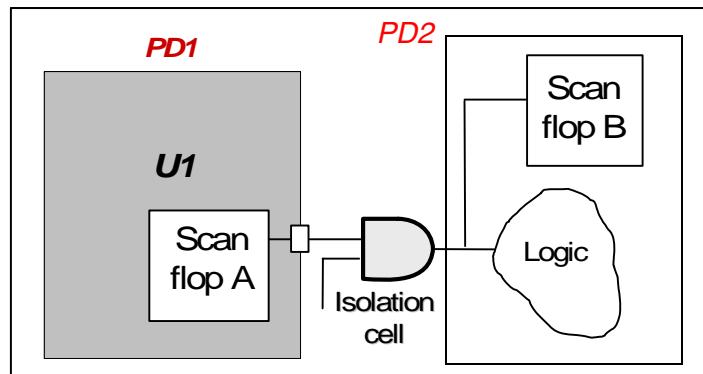
Figure 7-28 Level Shifter in a Block as Part of Scan Path



## Isolation Cell as Part of Scan Path

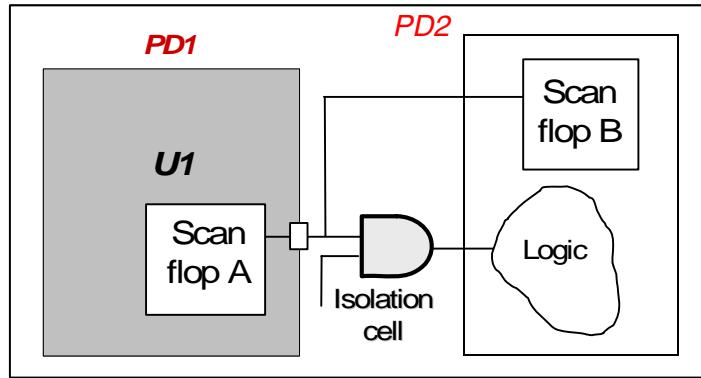
If a scan path goes through an isolation cell and you use the existing isolation cell, then scan insertion connects the scan chain at the output side of the isolation cell, as shown in [Figure 7-29](#).

Figure 7-29 Existing Isolation Cell as Part of Scan Path



If you do not reuse the existing isolation cells, then the scan path is connected to the net before the isolation cell, as shown in [Figure 7-30](#).

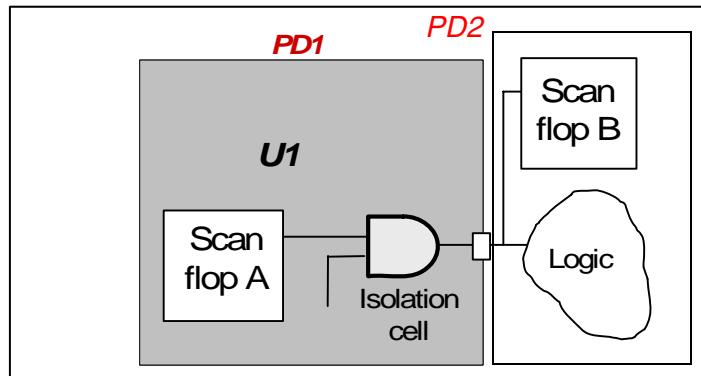
Figure 7-30 Isolation Cell as Part of Scan Path



You must then insert the isolation cell on the new net.

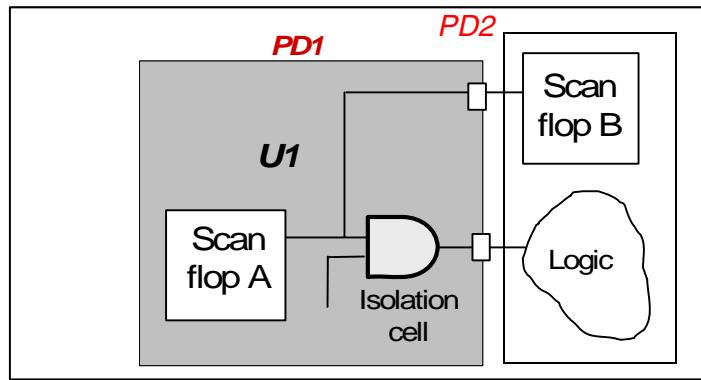
If the isolation cell is within a block and you reuse the existing isolation cell, then scan insertion reuses the existing hierarchical port, as shown in [Figure 7-31](#).

Figure 7-31 Existing Isolation Cell in a Block as Part of Scan Path



If you do not reuse the existing isolation cells and the existing isolation cell is within a block, then a new hierarchical port is added, as shown in [Figure 7-32](#).

Figure 7-32 Isolation Cell in a Block as Part of the Scan Chain



You must then insert the isolation cell on the new net.

---

## DFT Considerations for Low-Power Design Flows

In addition to level shifter cells, low-power designs often utilize the following special cells:

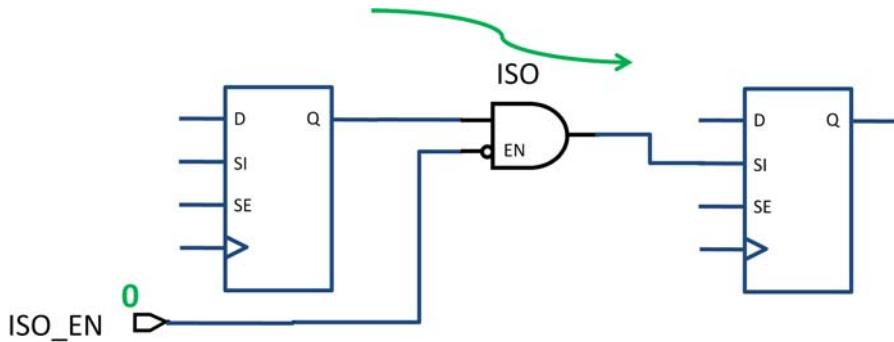
- Isolation cells
- Retention registers

You must configure these special cells such that the data shifts through the scan chains during test operations. The `dft_drc` command identifies the design rule violations, if any, that would prevent scan shifting through such cells.

### Isolation Cells

For example, if a scan chain traverses an isolation cell, you must ensure that the isolation cell passes the scan data to the flip-flop driven by the isolation cell during the test operation, as shown in [Figure 7-33](#).

Figure 7-33 Proper Configuration of a Scan Chain That Includes an Isolation Cell



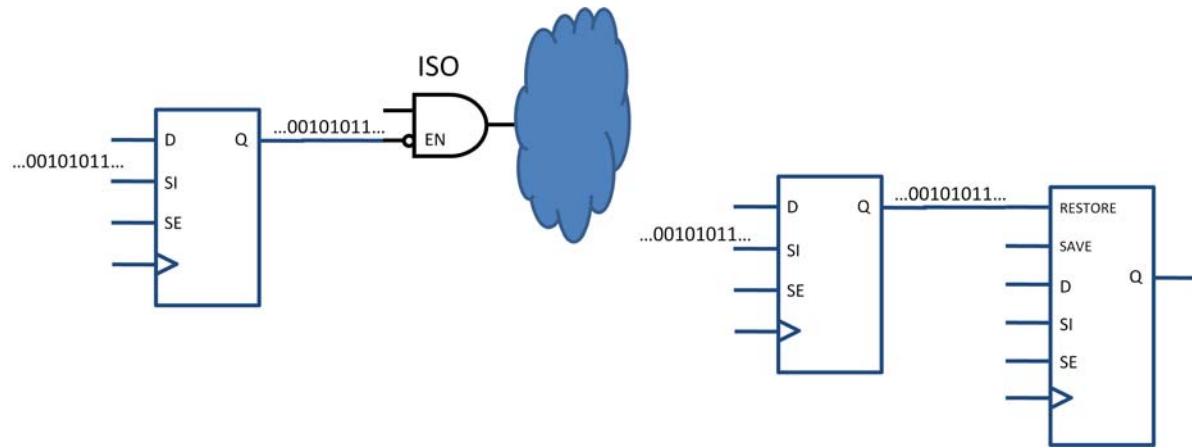
### Retention Registers

The design's retention registers must be in normal or power-up mode during the test operation. You should check that any save or restore signals are at their correct states.

### Registers Driven by Low-Power Control Signals

If the design contains registers that drive low-power control signals, such as the enable signal of the isolation cells or the save/restore signals of the retention registers, you must not put these registers onto the scan chains. Otherwise, this could cause these control signals to switch during test operations. [Figure 7-34](#) shows the consequences of putting such registers on the scan chains.

*Figure 7-34 Consequences of Putting Registers That Drive Low-Power Control Signals on the Scan Chains*



## Previewing a Multivoltage Scan Chain

The `preview_dft -show all` report indicates the operating condition and the power domain of a scan cell whenever a scan path crosses a voltage or power domain. It also indicates whether a scan cell is driving a level shifter or an isolation cell.

In [Example 7-21](#), (v) indicates that the scan cell drives a level shifter and (i) indicates that the scan cell drives an isolation cell.

### *Example 7-21 Preview Report With Voltage and Power Domains*

```
*****
Preview_dft report
For   : 'Insert_dft' command
Design : seqmap_test
Version: Z-2007.03
Date   : Tue Jan 30 17:02:47 2007
*****
Number of chains: 1
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks
Voltage Mixing: True
Power Domain Mixing: True

(i) shows cell scan-out drives an isolation cell
(l) shows cell scan-out drives a lockup latch
(s) shows cell is a scan segment
(t) shows cell has a true scan attribute
(v) shows cell scan-out drives a level shifter cell
(w) shows cell scan-out drives a wire

Scan chain '1' (test_si --> test_so) contains 56 cells:

u2/q_reg[3]  (voltage 1.08) (pwr domain 'pd_2') (clk, 55.0, falling)
u2/q_reg[4]
...
u2/q_reg[26]
u2/q_reg[27]
u1/q_reg[3]  (v)(i)  (voltage 0.80) (pwr domain 'pd_1')
u1/q_reg[4]
...
u1/q_reg[26]
u1/q_reg[27]
u2/q_reg[0]  (v)(i)  (voltage 1.08) (pwr domain 'pd_2') (clk, 45.0, rising)
u2/q_reg[1]
...
u2/q_reg[22]
u2/q_reg[23]
u1/q_reg[0]  (v)(i)  (voltage 0.80) (pwr domain 'pd_1')
u1/q_reg[1]
...
```

---

## **Running in UPF Mode**

You can run DFT Compiler in Design Compiler UPF mode by using the `-upf_mode` switch when you invoke `dc_shell` at the system prompt:

```
$ dc_shell-xg-t -upf_mode
```

When you run DFT Compiler in Design Compiler UPF mode, any necessary multivoltage special cells (level shifters and isolation cells) are automatically inserted according to the UPF specification. The UPF specification determines

- The type of additional level shifters or isolation cells that are needed when a scan path crosses a power domain
- The locations where these cells are inserted

The multivoltage options of the `set_scan_configuration` command, `-voltage_mixing`, `-power_domain_mixing`, and `-reuse_mv_cells`, work the same in UPF mode as in nonUPF mode except for isolation cells, which are inserted during DFT insertion in UPF mode.

Note:

For the `insert_dft` command to properly insert level shifters and isolation cells, you must specify the level shifter and isolation cell strategies on a power-domain basis, even if you have specified similar strategies on blocks and individual ports. If you only specify the strategies on the blocks and individual ports, the `insert_dft` command might not be able to automatically insert level shifters and isolation cells on any new ports that it creates.

For details of designing using UPF, see the *Synopsys Low-Power Flow User Guide, version A-2007.12*.

---

## Limitations

The limitations associated with multivoltage and multipower domains are as follows:

- Multivoltage and multipower domains are supported only in multiplexed flip-flop scan style.
- Multivoltage and multipower domains are supported only in the following flows:
  - Basic scan, including AutoFix, observe point insertion, user-defined test point insertion, HSS
  - Adaptive Scan
- Multivoltage and multipower domains are not supported in the following flows:
  - BSD Insertion
  - Deterministic BIST and X-tolerant deterministic BIST
  - Core integration

---

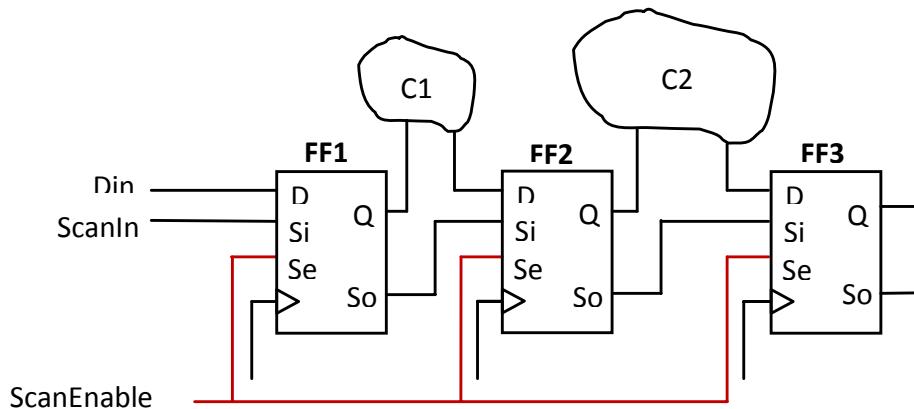
## Power-Aware Functional Output Gating

During scan testing, while scan data shifts through scan chains, the functional logic driven by the scan flip-flops is also toggling. This can cause increased power dissipation during testing, which could damage the design under test.

DFT can insert gating logic on functional output of scan elements during `insert_dft`. This power-aware feature will insert gating logic on the functional output of scan flip-flops that you have specified.

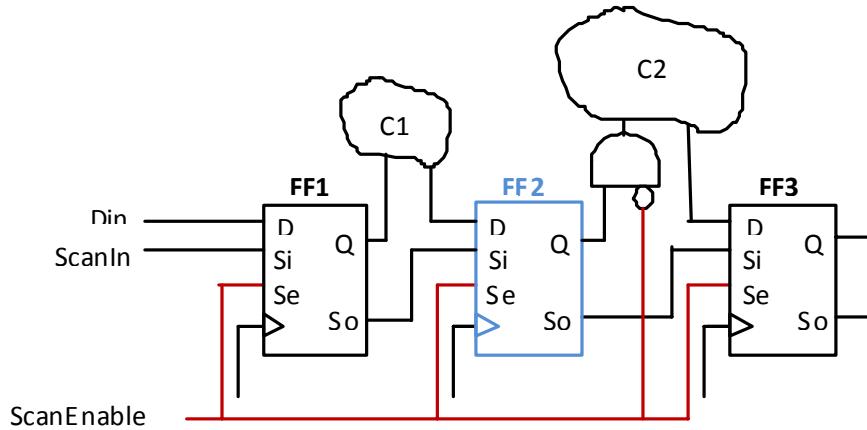
[Figure 7-35](#) shows the design after DFT insertion but without any gating logic inserted on the functional output.

*Figure 7-35 Design After DFT Insertion Without Gating Logic on the Functional Output*



[Figure 7-36](#) shows the design after DFT insertion when the user has specified that gating logic should be inserted on the FF2 flip-flop.

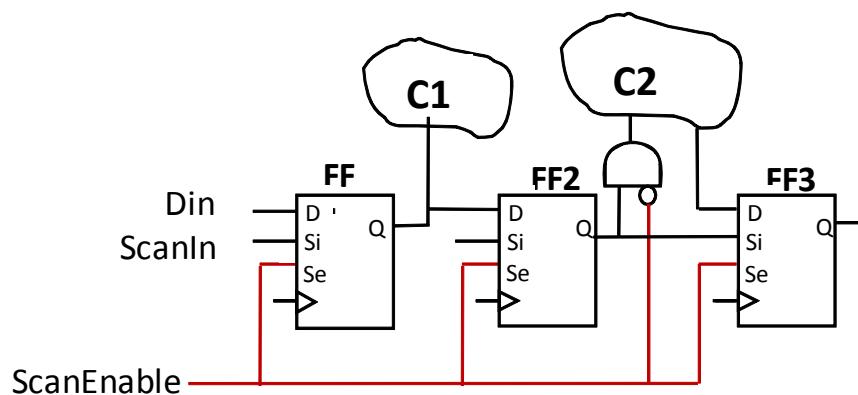
Figure 7-36 Design After DFT Insertion With Clock Gating Inserted on the FF2 Functional Output



In Figure 7-36 example, the logic cone C2 does not toggle during scan shifting because the Q output of FF2 is gated by the extra AND gate which is disabled by the Scan Enable signal.

If your scan flip-flops have only a single output pin that is shared between functional and scan output, the logic path going from the flip-flop output pin to the functional logic will be gated, as shown in the example provided in Figure 7-37.

Figure 7-37 Design After DFT Insertion With Gating Logic Inserted on the Functional Output of FF2, Where FF2 Has a Single Output Pin Shared Between Functional and Scan Output



## How to Use This Feature

To use this feature, run the following command as part of your DFT specifications *before* you run `insert_dft`:

```
set_scan_suppress_toggling -include <scan_element(s)_to_be_gated>
```

The specification is cumulative, which means that you can issue this command multiple times to add to the list of elements that should be gated. You can use a list or a collection to specify the elements to be gated.

Functional gating logic will be inserted if the scan flip-flop is part of a scan chain, with the following exceptions:

- The flip-flop is part of a shift register segment identified by `compile_ultra`
- In a bottom-up flow, the flip-flop is within a block where test models are used or you have specified a `set_dont_touch` on the block.

To report on what you have previously specified, use `report_scan_suppress_toggling`.

To remove what you have previously specified, use `remove_scan_suppress_toggling`. This command removes all elements that you previously specified to have gating logic inserted.

Note:

This feature only works with designs using the multiplex flip-flop scan style.

---

## Internal Pins Flow

The DFT Compiler internal pins flow is a test methodology that provides a way to use internal pins, such as scan-in, scan-out, and clocks ports, as top-level ports. This section has the following subsections:

- [Understanding the Architecture](#)
- [DFT Commands](#)
- [Scan Insertion Flow](#)

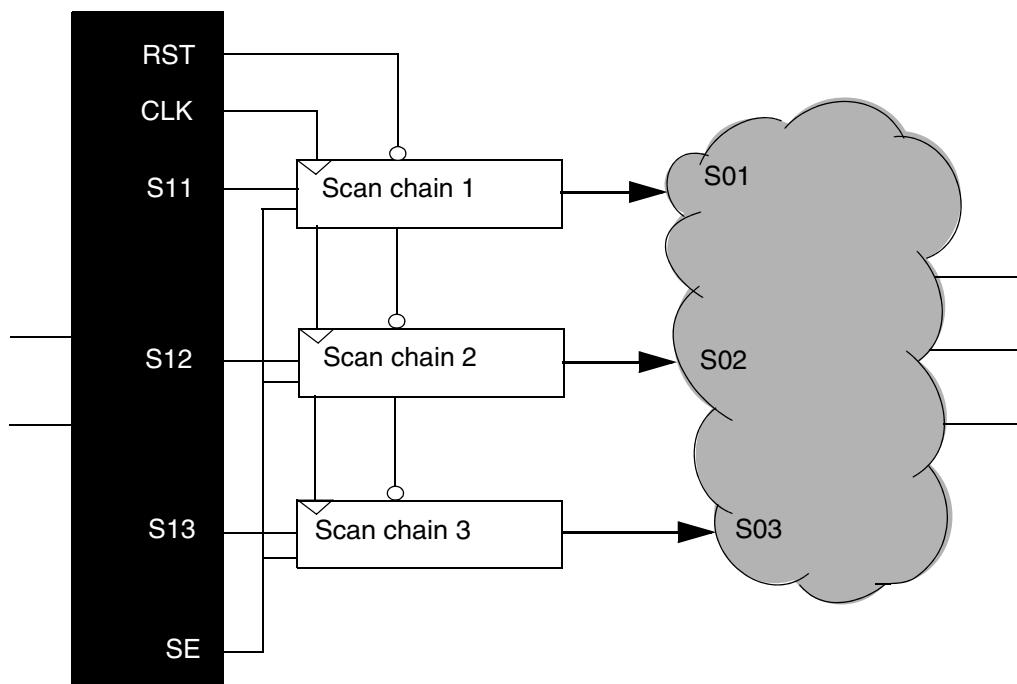
---

## Understanding the Architecture

In the traditional scan architecture approach, the scan chain clock, set/reset, scan-in, scan-out, and scan-enable signals are top-level ports. The internal pins flow enables the use of internal pins instead of top-level ports. The diagram in [Figure 7-38](#) shows the architecture of a typical top-level design with internal pins used for scan inputs, scan outputs, clocks, and resets.

*Figure 7-38 Top-Level Design With Internal Signal Connections*

Top-Level Design



In the internal pins flow, the original design is modified in the internal database so that the internal pins are used as top-level ports. These changes are transparent and do not affect the netlist and protocol that is written out.

---

## DFT Commands

The internal pins flow is controlled primarily by two commands:

- `set_dft_drc_configuration` – enables the internal pins flow
- `set_dft_signal` – specifies all hookup pins

## Enabling the Internal Pins Flow

The `set_dft_drc_configuration` command enables the internal pins flow. See the following example:

```
dc_shell> set_dft_drc_configuration \
           -internal_pins enable
```

The default value of the `-internal_pins` switch is disable.

## Specifying Hookup Pins

All hookup pins are specified by use of the `-hookup_pin` switch in the `set_dft_signal` command. The following examples show how to use the `-hookup_pin` switch for various data types:

### Clocks:

```
dc_shell> set_dft_signal -view existing \
           -type ScanClock \
           -hookup_pin {pinName} -timing [list 45 55]
```

### Reset:

```
dc_shell> set_dft_signal -view existing -type Reset \
           -hookup_pin {pinName} -active_state 0|1
```

### Constant:

```
dc_shell> set_dft_signal -view existing -type Constant \
           -hookup_pin {pinName} -active_state 0|1
```

### TestMode:

```
dc_shell> set_dft_signal -view spec -type TestMode \
           -hookup_pin {pinName} -active_state 0|1
```

### Scan\_in:

```
dc_shell> set_dft_signal -view spec -type ScanDataIn \
           -hookup_pin {pinName}
```

### Scan\_out:

```
dc_shell> set_dft_signal -view spec -type ScanDataOut \
           -hookup_pin {pinName}
```

### Scan\_enable:

```
dc_shell> set_dft_signal -view spec -type ScanEnable \
           -hookup_pin {pinName}
```

#### Scan path definition for using hookup pins:

```
dc_shell> set_scan_path {scanChainName} -scan_data_in \
           {pinName} -scan_data_out {pinName}
```

##### Note:

The internal pins flow does not work when inputs are used for the `-hookup_pin` option.

##### Note:

For all scan signals without an as-yet existing path between the port and hookup pin, only the `-hookup_pin` option should be used. That is, do not use `-port` for the internal pins flow.

---

## Scan Insertion Flow

This section describes the scan insertion flow, which is currently the only flow that supports internal pins modification.

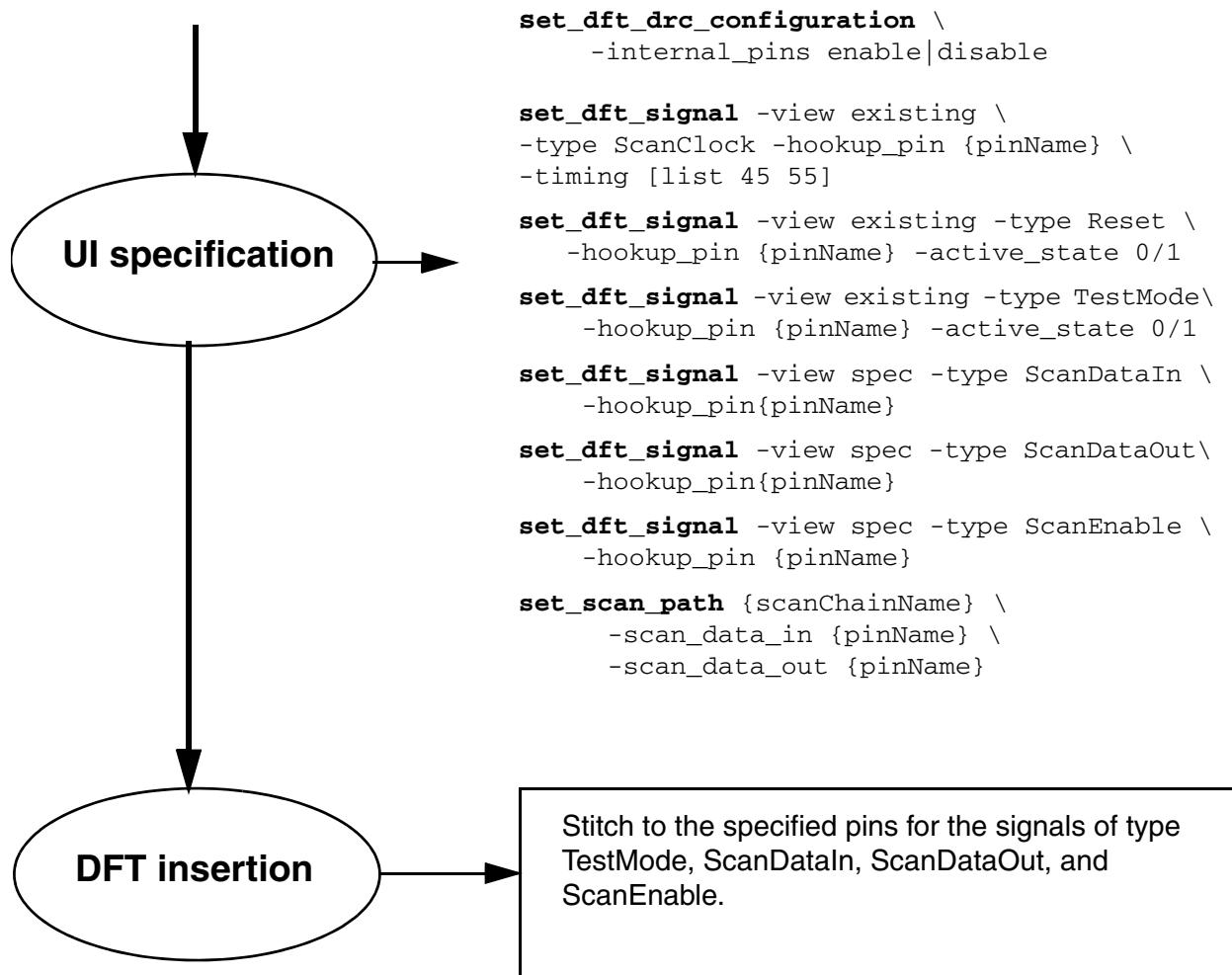
The following data types are supported in this flow as internal signals:

- ScanMasterClock
- MasterClock
- ScanClock
- Reset
- Constant
- TestMode
- ScanEnable
- ScanDataIn
- ScanDataOut

Internal pins are stitched during DFT insertion, based on what you specify with the `set_dft_signal` command. These specifications should precede the running of `dft_drc` command.

[Figure 7-39](#) shows a typical internal pins flow for scan insertion.

Figure 7-39 Simple Scan Insertion Flow

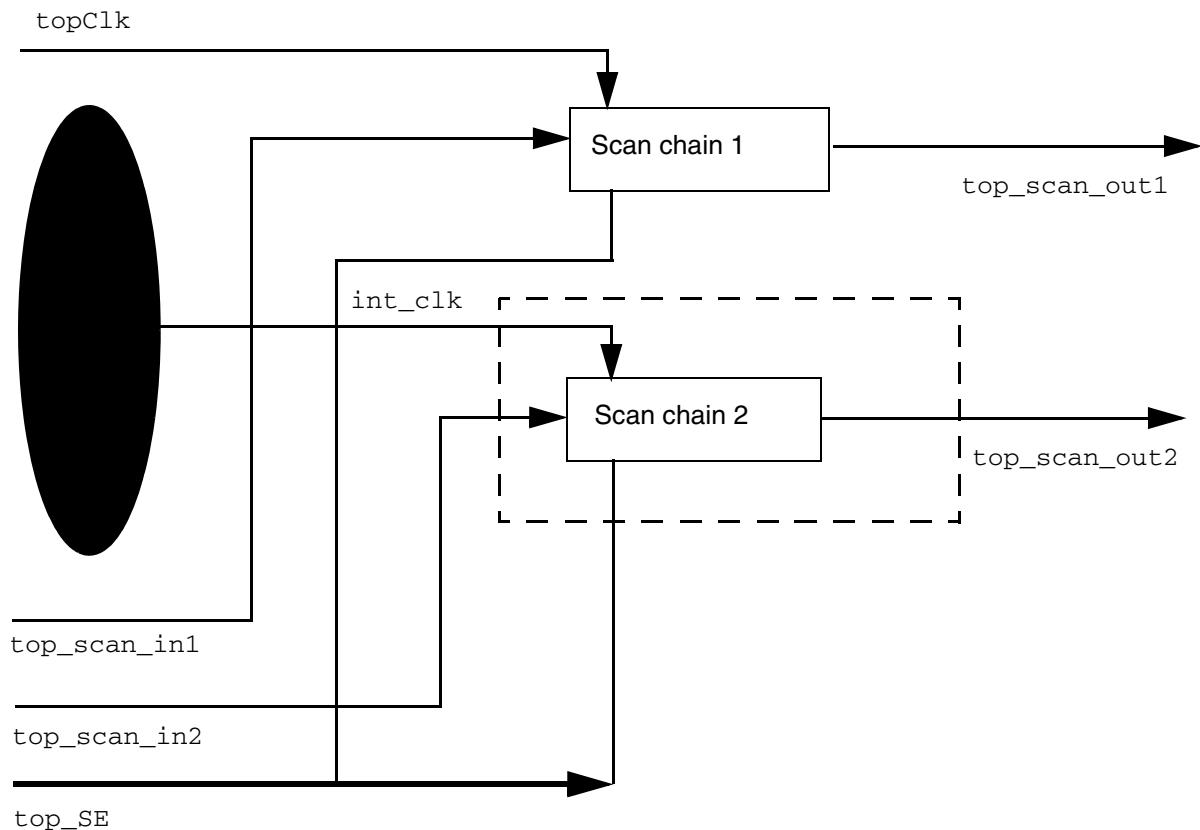


---

## Mixing Ports and Internal Pins

You can mix the specification of top-level ports with user-specified internal pins for all the signal types. For example, you can specify both a top-level port and an internal pin as a clock, as shown in [Figure 7-40](#).

*Figure 7-40 Mixing Specifications of Top-Level Ports and Internal Pins*



```
set_dft_signal -port topClk -view existing -type ScanClock \
    -timing {45 55}

set_dft_signal -view existing -type ScanClock \
    -hookup_pin blackBox/int_clk -timing {45 55}
```

---

## **Limitations**

The following limitations currently apply to this feature:

- The output protocol is not accurate and cannot be used in TetraMAX.
- DBIST, phase-locked loop (PLL), core wrapper, boundary scan, and scan extraction flows do not support the internal pin flow.
- You cannot run Hierarchical Scan Synthesis on blocks that were created using the internal pin flow.

---

## **Support for Implicit Scan Chain Elements**

This feature provides a mechanism to specify one or more “implicit” scan segments as scan chains that are outside the design in which the DFT MAX IP will be inserted. An implicit scan segment is a scan group that is included in a scan chain. The tool characterizes this chain by name, length, scan input, scan output, and scan clock access ports.

An implicit scan chain is considered a pre-stitched scan chain. It is instantiated at some location in the design other than under the design level in which the DFT MAX IP is to be inserted.

The feature provides the ability to connect the DFT MAX load compressor outputs to ports in the current design which then connect to the implicit scan chain input. Likewise, DFT Compiler will connect the current design port connected to the implicit scan chain output to the appropriate DFT MAX unload compressor input. Finally, DFT Compiler will produce a partial test protocol. This protocol is not complete and contains only a partial ScanChain definition for the implicit chain. The protocol cannot be used in TetraMAX directly.

Note:

For information concerning DFT MAX load compressor and other DFT MAX functions, see the *DFT MAX User Guide: Adaptive Scan*.

---

## **Command Usage**

Two commands are used to specify implicit scan chains:

- `set_scan_group`
- `set_scan_path`

When using the `set_scan_group` command, you must specify the implicit group name, the length, the scan in and out access pins, as well as specifying that the chain is serially routed with the `serial_routed` switch. You can also specify the scan clock, but this is not required.

The `set_scan_path` command is used to define a path and to include an implicit group into the chain. For the `set_scan_path` command, you specify the name, the name of the implicit segment, and that the chain is complete. An example is provided below, and a complete script is given later in this document.

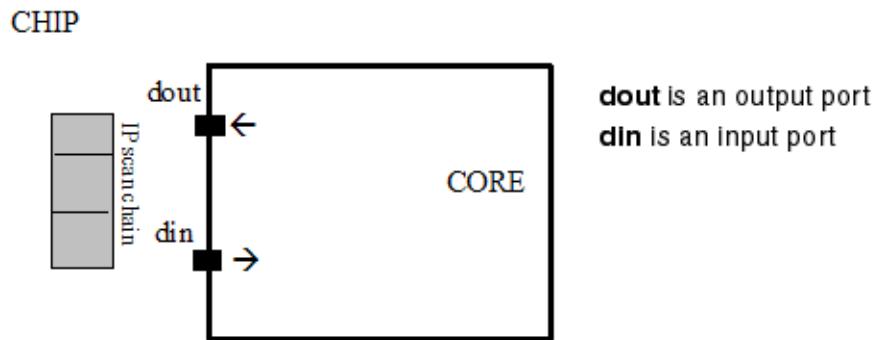
---

## Usage Scenario

Note the following important considerations when using the implicit scan chain feature:

- DFT MAX adaptive scan logic is inserted at the CORE level of the design.
  - The CORE contains several subsystems implemented as a single physical unit.
- Top-level IPs remain outside these subsystems for physical integration and optimization.
- CORE-level DFT MAX insertion must take these top-level IP chains into account:
  - DFT MAX load compressor is connected to the IP scanin.
  - DFT MAX unload compressor is connected to the IP scanout.
  - Reconfiguration muxes are properly placed.

*Figure 7-41 Example Partial Circuit Before Implicit Scan Chain Insertion*



- In the partial circuit of [Figure 7-41](#), the intention is to insert adaptive scan logic on the CORE and ultimately to integrate this logic at the chip level.

Note: automated top-level CORE integration using implicit scan chains is not supported by DFT Compiler.

- The CHIP level IP scan chain must be driven by the DFT MAX load compressor/ unload compressor inserted at the CORE level through dout and din.
- While inserting adaptive scan logic at the CORE level, you must be able to specify an implicit scan segment between ports dout and din so that the tool can make appropriate connections to these access ports.
- Multiple IP scan segments may exist.
- The implicit scan chains (IP scan chains) will be pre-stitched scan chains.
- The connection from the pin CORE/dout to the IP scan chain scanin will not be made automatically by DFT Compiler. You must perform this task manually.
- The connection from the IP scan chain scanout to CORE/din will not be made automatically by DFT Compiler. You must perform this task manually.
- The implicit scan segment is characterized by access ports (length, scanin, scanout, clock associations) in the test protocol. Therefore, by default, each implicit scan segment will be on a separate scan chain. This can be overridden by using set\_scan\_path – clock\_mixing mix\_clocks and multimode specifications to control clock mixing.

Figure 7-42 Example Partial Circuit With Implicit Scan Chain Insertion

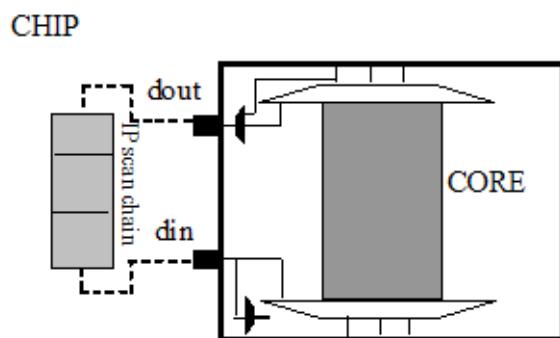


Figure 7-42 shows the results of scan insertion using implicit scan chain support. In the figure, note the following:

- The dotted line from CORE/dout to the IP scan chain scanin indicates the manual connection you must make.
- The dotted line from the IP scan chain scanout to CORE/din indicates the manual connection you must make.

- The muxes within CORE that are associated with dout and din are the test mode reconfiguration logic. These muxes are inserted in all DFT MAX flows and provide a means to reconfigure the compression mode chains into longer chains with fewer top-level scan inputs and outputs.
- 

## Example Script

```
# process CORE which hooks up to the implicit IP scan chains
current_design core1

# create scan port which will connect to implicit scan chain scanin
create_port -dir out dout

# create scan port which will connect to implicit scan chain scanout
create_port -dir in din

# define all test signals
# test clocks must be defined before they can used in set_scan_group -clock
set_dft_signal -view existing_dft -type ScanMasterClock \
    -timing {45 55} -port clk_st

# define the implicit scan chain and access pins
# in the CORE design, dout drives the scan input of IP scan chain
# din is driven by the scan out of the IP scan chain
set_scan_group IMPLICIT1 -segment_length 67 -serial_routed true \
    -access [list ScanDataIn dout ScanDataOut din] \
    -clock clk_st

# define each implicit chain as a scan path
set_scan_path c1 -order [list IMPLICIT1] -test_mode all -complete true

# enable DFT MAX scan compression insertion
set_dft_configuration -scan_compression enable

# configure DFT MAX
set_scan_compression_configuration -minimum_compression 25 \
    -xtolerance high
set_scan_configuration -chain_count 8 -test_mode all

# create the test protocol
create_test_protocol

# run pre-drc
dft_drc -v

# preview DFT insertion
preview_dft -show all

# specify no optimization during DFT insertion
set_dft_insertion_configuration -synthesis_optimization none
```

```

# perform DFT insertion
insert_dft

#post DRC is not supported

change_names -rule verilog -hier

# write out the compression mode protocol for TetraMAX
write_test_protocol -test_mode ScanCompression_mode \
    -out stil/scan_compression.stil -names verilog

# write out the pure scan protocol for TetraMAX
write_test_protocol -test_mode Internal_scan \
    -out stil/internal_scan.stil -names verilog

# write out the inserted design in Verilog and Synopsys ddc format
write -format verilog -hier -output vg/design_with_implicit.v
write -format ddc -hier -output db/design_with_implicit.ddc

```

**Note:**

You must specify a nonzero segment length through the `-segment_length` option for the implicit scan groups. Otherwise, the scan architecting will be incorrect.

## Example Protocol

The following example is taken from the ScanStructures section of the test protocol written out by DFT Compiler. The implicit scan segment is characterized by name, scan length, scan data in, scan data out, and the scan clock in the test protocol.

Example definition of an implicit scan chain:

```

ScanStructures {
    ScanChain "c1" {
        ScanLength 67;
        ScanIn "dout";
        ScanOut "din";
        ScanMasterClock "clk_st";
    }
}

```

In the same protocol, an example definition of a normal (non-implicit) scan chain:

```

ScanChain "2" {
    ScanLength 10;
    ScanEnable "test_se";
    ScanMasterClock "clk_st";
}

```

Each implicit scan segment has clock information contained within the ScanStructures-ScanChain definition. By default, each implicit chain is placed on a separate scan chain because DFT Compiler does not mix clock domains on a scan chain. This default behavior can be overridden by using `set_scan_path -clock_mixing mix_clocks` along with multimode specifications to control clock mixing.

---

## Limitations

It is important to note the following limitations:

- There is no support for post-DFT DRC in the presence of implicit scan segments.
- Only positive edge chains are supported
- The feature is only supported in XG mode.
- It is assumed that the IP scan chain will be stitched manually at the CHIP level.
- The CORE/dout connection to the IP scan chain scanin is not performed by DFT Compiler and must be done manually by you.
- The IP scan chain scanout connection to CORE/din is not performed by DFT Compiler and must be done manually by you.
- The test protocol produced in DFT Compiler with this flow requires that you run edits from the top level of the design.
- *You must specify a nonzero segment length through the -segment\_length option for implicit scan groups. Otherwise, scan architecting will be incorrect.*

---

## Creating Scan Groups

DFT Compiler offers a methodology that enables you to define certain scan chain portions so that they can be efficiently grouped with other scan chains. This is done without the need to insert scan at the submodule levels.

This section includes the following subsections:

- [Configuring Scan Grouping](#)
- [Scan Group Flows](#)
- [Known Limitations](#)

---

## Configuring Scan Grouping

DFT Compiler includes a new set of commands that enable you to configure scan grouping:

- `set_scan_group` – creates scan groups
- `remove_scan_group` – removes scan groups
- `report_scan_group` – reports scan groups

### Creating Scan Groups

The `set_scan_group` command enables you to create a set of sequential cells, scan segments, or design instances that should be grouped together within a scan chain. If a design instance is specified, all sequential cells and segments within it are treated as a group and will be logically ordered.

The syntax for this command is as follows:

```
set_scan_group scan_group_name
-include_elements {list_of_cells_or_segments}
[-access {list_of_access_pins}]
[-serial_routed true|false]

scan_group_name
    Specifies a unique group name.

-include_elements {list_of_cells_or_segments}
    Specifies a list of cell names or segment names that are included in the group.

-access {list_of_access_pins}
    Specifies a list of all access pins. Note that these access pins represent only a serially
    routed scan group specification. If the -serially_routed option is set to false, all
    specified access pins are ignored.

-serial_routed true | (false)
    Specifies whether the scan group is serially routed (true) or not (false). The default
    value is false.
```

Note the following:

- There is no `-view` option to the `set_scan_group` command, because the options only work in the specification view.
- All scan group specifications are applied across all test modes. You cannot specify a scan group to be applied on a particular test mode.

- An element specified as part of a scan group cannot be specified as part of a scan path, and vice versa.
- Scan group specifications are not cumulative. If you specify the same group name, the last scan group specification prior to `insert_dft` is used.
- Grouping elements implies that they must be adjacent in a scan chain.

**Example.**

```
dc_shell> set_scan_group G1 -include_elements \
           [list ff1 ff3] -access \
           [list ScanDataIn ff1/TI ScanDataOut \
           ff3/QN] -serial_routed true

dc_shell> set_scan_group G2 -include_elements \
           [list ff2 a/ff1]

dc_shell> set_scan_group G3 -include_elements \
           [list U1/1]

dc_shell> set_scan_group G4 -include_elements \
           [list U1/3]
```

## Removing Scan Groups

The `remove_scan_group` command removes all specified scan groups. The syntax of this command is as follows:

```
remove_scan_group scan_group_name
scan_group_name
```

This option specifies the name of the scan group to be removed.

**Example.**

```
dc_shell> remove_scan_group G1
```

## Integrating an Existing Scan Chain Into a Scan Group

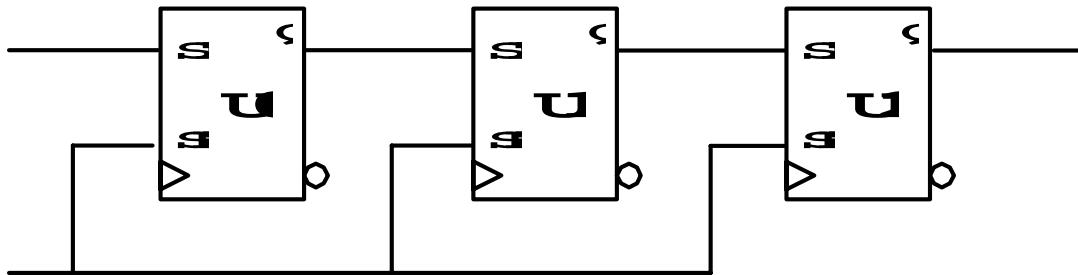
If you have existing serial routed segments at the current design level that you want to incorporate as part of a longer scan chain, you can use the `set_scan_group -serial_routed true` command to accomplish this. When you run `insert_dft`, it will then connect to this segment while keeping the segment in tact.

You need to provide the following information to the `set_scan_group -serial_routed true` command:

- Use the `-include_elements` option to specify the names of the elements within the segment.
- Use the `-access` option to specify how `insert_dft` should connect to this segment.

Consider the existing scan chain shown in [Figure 7-43](#).

*Figure 7-43 Integrating an Existing Scan Chain*



In this example, the scan chain connects the flip-flops U0 through U1 to U2. The `insert_dft` command treats U0 through U2 as a subchain or group and connects through the scan-in pin of U0, the output pin of U2, and the scan-enable pin of U0. You can then use the following command to specify the existing scan segment:

```
set_scan_group group1 -include_elements [list U0 U1 U2] \
    -access [list ScanDataIn U0/SI ScanDataOut U2/Q ScanEnable U0/SE]
```

If you do not know the names of the individual elements within the scan group, you can try extracting the element names by using the `dft_drc` command. Note, however, that this strategy works only if the scan segment starts and ends at the top level of the current design. See [“Performing Scan Extraction” on page 7-2](#) for details on how to extract pre-existing scan chains in your design. After you have extracted the names of the elements of the scan chain, you can specify them using the `-include_elements` option. You might need to disconnect the net connecting the top-level scan-in port to the scan-in pin of the first flip-flop of the chain, as well as the net connecting the data-out or scan-out pin of the last flip-flop of the chain to the scan-out port.

## Reporting Scan Groups

The `report_scan_group` command reports the names of the sequential cells or scan segments associated with a particular scan group, as specified by the `set_scan_group` command).

The syntax of this command is as follows:

```
report_scan_group [scan_group_name]  
scan_group_name
```

This option specifies the name of the scan group used for reporting purposes. If a group name is not specified, all serially routed and unrouted groups will be reported.

### Examples.

```
dc_shell> report_scan_group G1  
dc_shell> report_scan_group
```

---

## Scan Group Flows

You can specify scan groups in DFT Compiler in either the standard flat flow or the Hierarchical Scan Synthesis (HSS) flow.

In the standard flat flow, you can specify valid scan cells as input to scan groups. In the logical domain, these cells get ordered logically and placed as a group in a scan chain.

In the HSS flow, you can specify core segment names as part of a scan group and then reuse them in a scan path specification.

---

## Known Limitations

The following known limitations apply when you specify scan groups in DFT Compiler:

- A scan group can contain only a set of elements that belong to one clock domain.
- You cannot specify a collection as a scan group element.
- You cannot specify a group as part of another group.
- You cannot specify a previously defined scan path in a scan group.

---

## Identification of Shift Registers

DFT Compiler does not identify shift registers but uses any shift register identified and optimized by Design Compiler (`compile_ultra -scan`). Furthermore, DFT Compiler does not re-identify shift registers, but the tool does optimize the existing shift registers for scan stitching and remove any flip-flops containing DRC violations that were not fixed by DFT Compiler. The identified shift registers can be broken or optimized for scan

configuration, and once they are optimized, they will be treated as scan segments in the DFT Compiler design flow. A shift register will go to a single chain and should not be broken up for balancing.

Note:

ASCII netlist flows are also supported with shift-register identification. When you are reading in an ASCII netlist (Verilog or VHDL) in which scan-replacement has already been performed with automatic shift-register identification, use `set_scan_state test_ready` to indicate that the netlist is scan-replaced. This allows Design Compiler and DFT Compiler to correctly handle the identified shift registers from a previous compile.

The `insert_dft` command recognizes shift registers that were identified by `compile_ultra -scan` based on structural analysis. After the shift registers are recognized, DFT Compiler unscans all elements of the shift register except the first one.

The `preview_dft -show all` command reports any registers that were identified by `compile_ultra -scan` as shift registers. After the shift registers are identified, they are considered fixed segments.

For more information about identifying shift registers, see the section on test-ready compile in the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

# 8

## Wrapping Cores

---

This chapter shows you how to add a test wrapper to a core design. A test wrapper provides both test access and test isolation during scan pattern application. The commands necessary to implement a test wrapper around a core are described in the following sections:

- [Core Wrapping Commands](#)
- [Wrapper Cells](#)
- [Core Wrapping Flows](#)
- [Core Wrapping Scripts](#)

---

## Core Wrapping Commands

[Table 8-1](#) shows the principal commands used in core wrapping.

*Table 8-1 Principal Core Wrappings Commands*

Commands (XG mode)
remove_boundary_cell
remove_scan_path
reset_wrapper_configuration
set_boundary_cell -class core_wrapper
define_dft_design
set_wrapper_configuration -class core_wrapper
set_scan_configuration -max_length -test_mode
set_scan_configuration -chain_count-test_mode
set_scan_path -class wrapper -test_mode
set_dft_configuration -wrapper enable
set_dft_signal

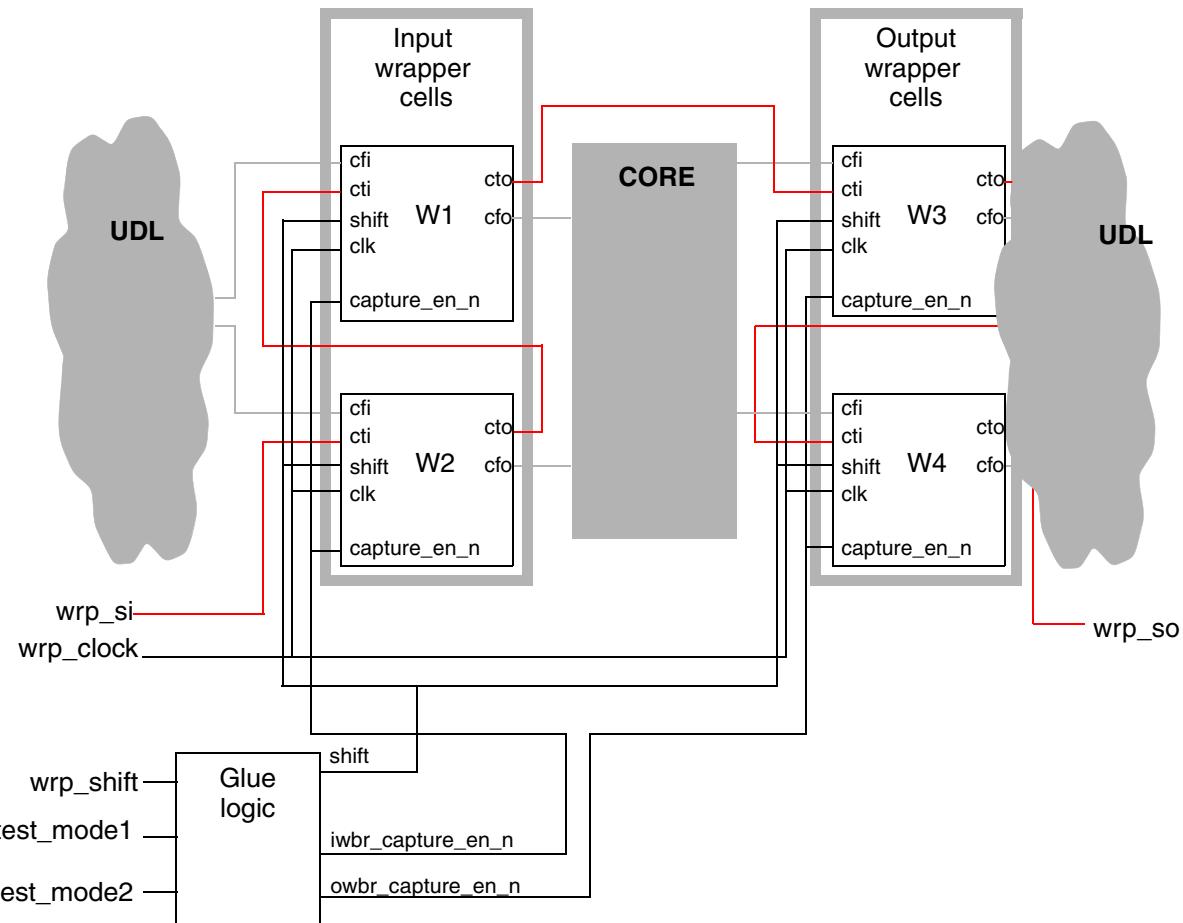
---

---

## Wrapper Cells

A scan test wrapper consists of a chain of wrapper cells. Wrapper cells provide both controllability at the inputs and observability at the outputs of the core. [Figure 8-1](#) illustrates a core with wrapper cells inserted for each signal between the core and the user-defined logic (UDL). Depending on the mode, the input wrapper cells can control the inputs to the core and observe responses from the surrounding logic. The output wrapper cells can control inputs to the surrounding logic and observe core responses.

Figure 8-1 Test Wrapper



This section includes the following subsections:

- [Test Wrapper Operation](#)
- [Test Wrapper Control Interface](#)
- [Test Wrapper Cell Interface](#)
- [Delay Test Wrapper Insertion](#)
- [Separate Input and Output Wrapper Chains and Control](#)
- [Wrapping Three-State and Bidirectional Ports](#)
- [Specifying Multiple Wrapper Modes](#)
- [Test Wrapper Exceptions](#)
- [Specifying Wrapper Chains](#)

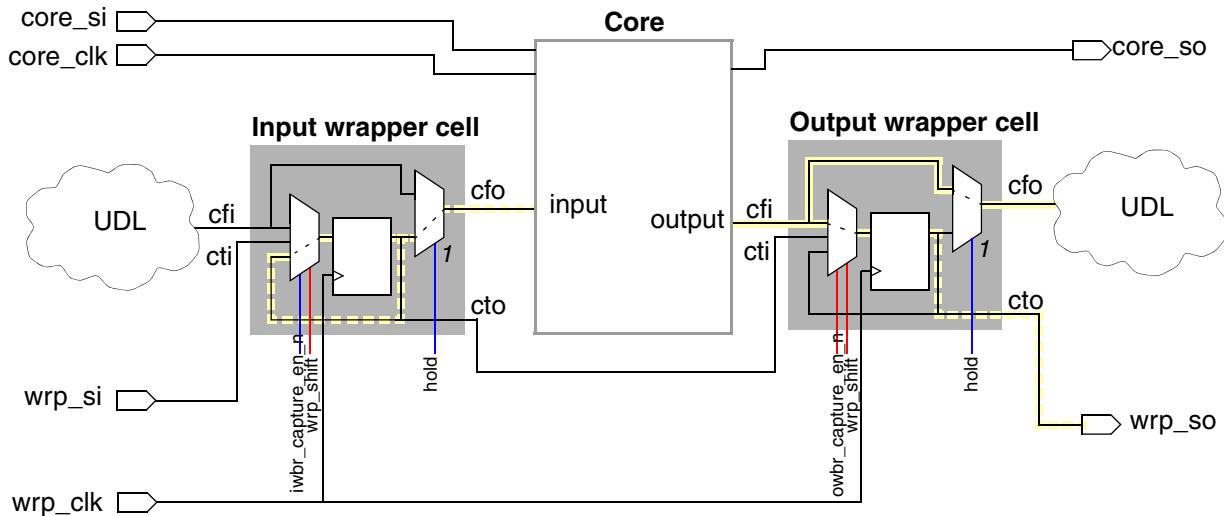
## Test Wrapper Operation

The test wrapper can operate in one of four modes of operation:

- INTEST: inward-facing mode

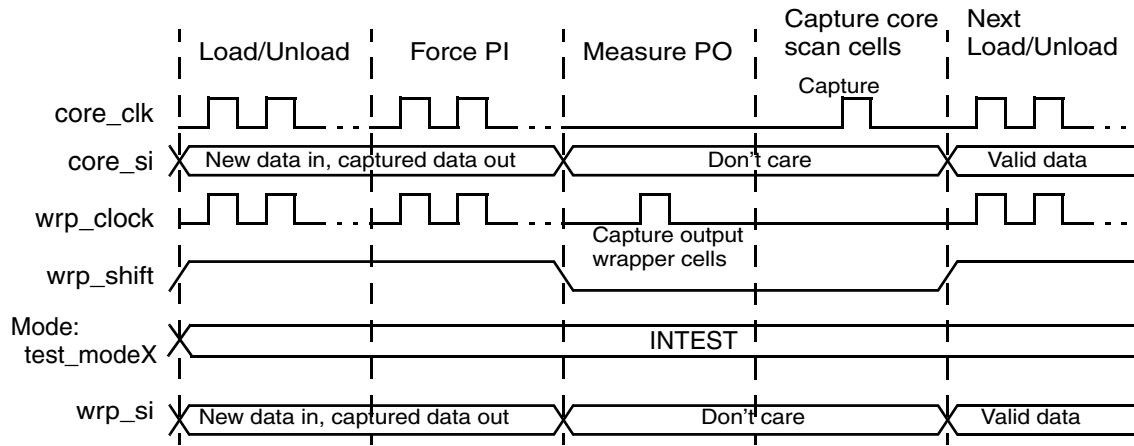
This mode is used when input vectors need to be applied to the core and the core response needs to be observed at the output. The wrapper cells on the core inputs provide controllability, and the wrapper cells on the core outputs provide observability. This mode is used to test the core and isolate the surrounding user-defined logic. To protect the user-defined logic from the core output response, safe value logic is added at the core outputs. [Figure 8-2](#) shows operation of the design in INTEST mode, and [Figure 8-3](#) shows the timing in INTEST mode.

*Figure 8-2 INTEST Mode of Operation of Wrapper Cells*



Signal	Value
wrp_shift	0
iwbr_capture_en_n	1
owbr_capture_en_n	0

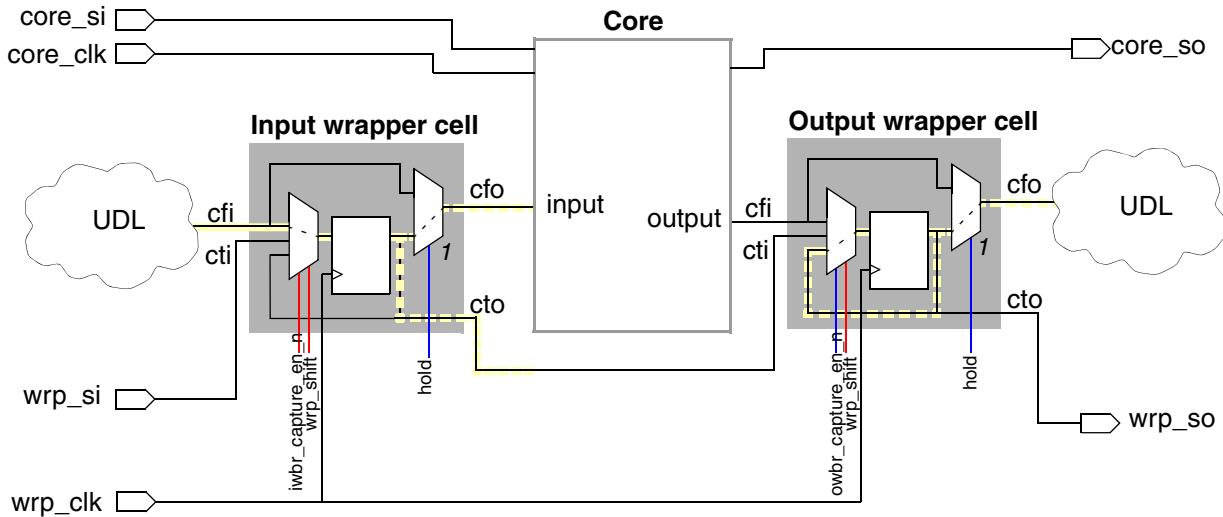
Figure 8-3 INTEST Mode Timing



- EXTEST: outward-facing mode

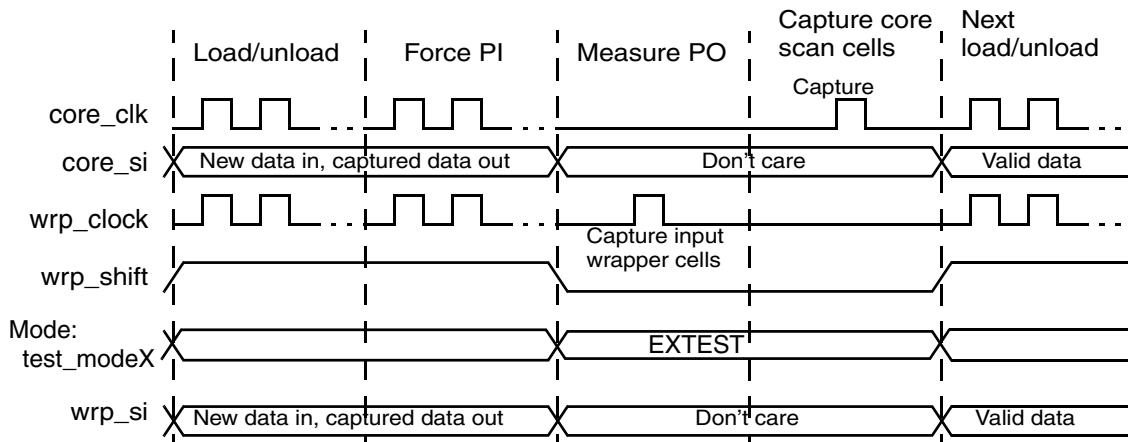
This mode is used to test the user-defined logic surrounding the core and isolate the core itself. The wrapper cells on the core inputs provide observability, and the wrapper cells on the core outputs provide controllability. To protect the core inputs from the user-defined logic responses, safe value logic is added at the core inputs. Figure 8-4 shows operation of the design in EXTEST mode, and Figure 8-5 shows the timing in EXTEST mode.

Figure 8-4 EXTEST Mode of Operation of Wrapper Cells



Signal	Value
wrp_shift	0
iwbr_capture_en_n	0
owbr_capture_en_n	1

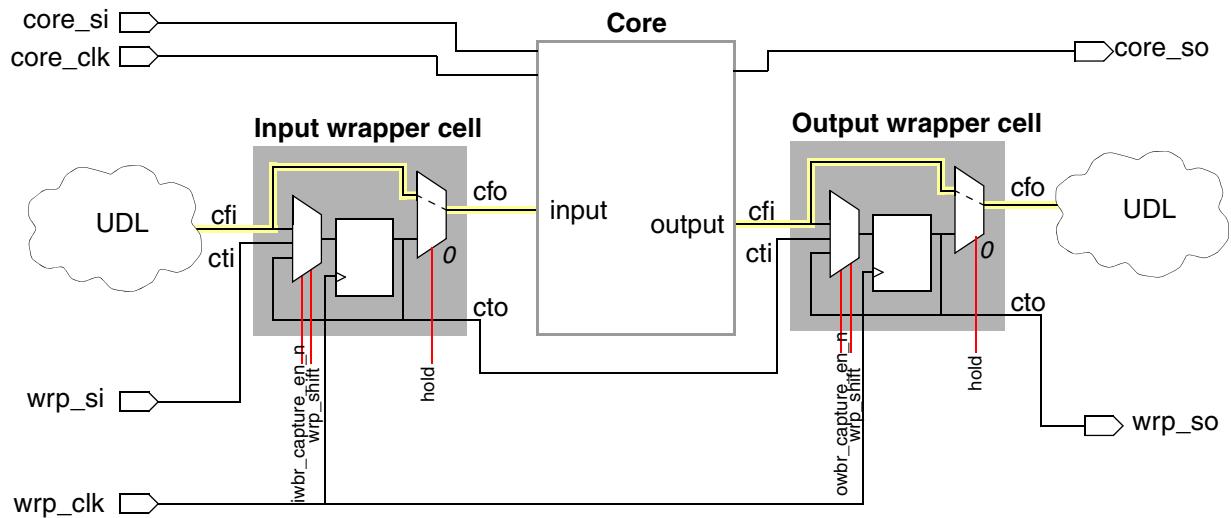
Figure 8-5 EXTEST Mode Timing



- NORMAL: normal mode of operation

This is the functional mode of operation. The test wrapper is transparent. [Figure 8-6](#) shows operation of the design in NORMAL mode.

Figure 8-6 Normal Mode Operation of Wrapper Cells



Signal	Value
wrp_shift	0
iwbr_capture_en_n	0
owbr_capture_en_n	0

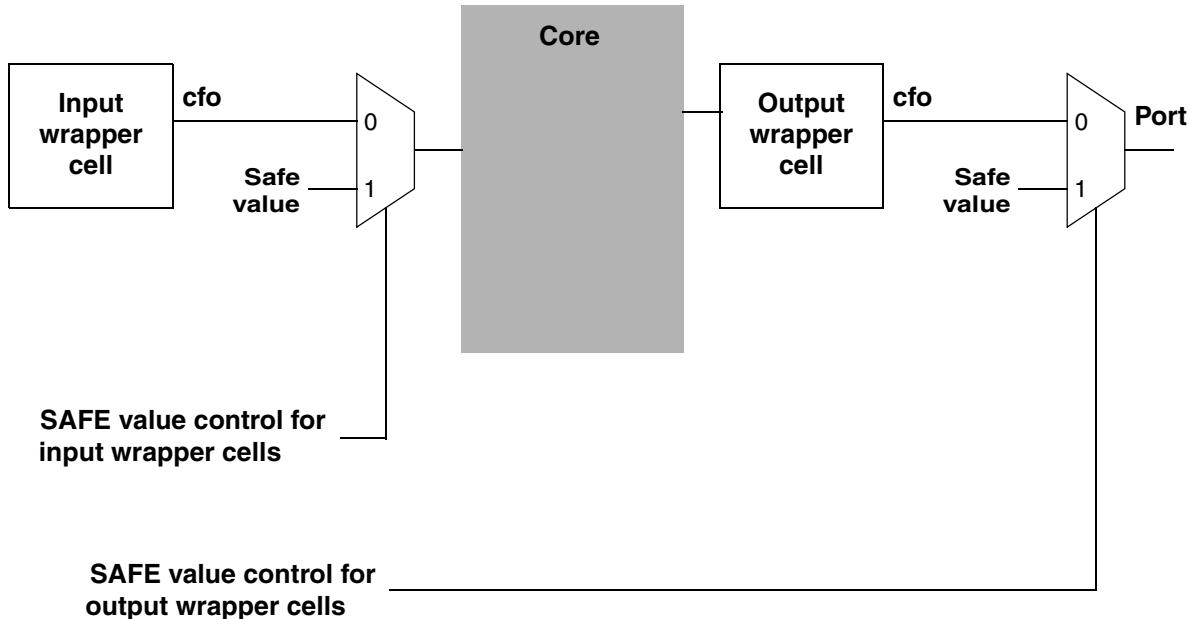
- SAFE: safe mode

In addition to automatically protecting the core inputs while testing the user-defined logic and protecting the user-defined logic while observing the core outputs, this mode enables you to put the core in safe mode independently. In this mode, a constant value of 0 or 1 is applied to the core inputs or user-defined logic accordingly. The addition of this safe-value logic depends on the user specification. If you do not specify safe-value for a particular design port, the wrapper cell for that port does not have the safe-value logic.

## Activating the SAFE Mode

The logic for safe-mode operation is added as glue logic after the input or output wrapper cell for a port if you specify a safe-value for the port. [Figure 8-7](#) illustrates this scheme. The safe-value control is generated differently for input and output wrapper cells.

Figure 8-7 SAFE-Value Logic




---

## Test Wrapper Control Interface

The test wrapper control interface provides a standard means of controlling the wrapper cells. The wrapper control interface is made up of the ports at the core level that control the test wrapper cells. You can specify any existing design port to be used as a wrapper control signal by using the `set_dft_signal` command. If you do not specify a port, DFT Compiler creates a new design port, if applicable, using the signal names shown in [Table 8-2](#).

*Table 8-2 Test Wrapper Control Interface Signals*

Signal	Description
wrp_shift	Wrapper shift

Core wrapping uses test mode ports (for example, `test_mode1` and `test_mode2`) to enable wrapper-safe, wrapper-inward-facing, and wrapper-outward facing modes.

---

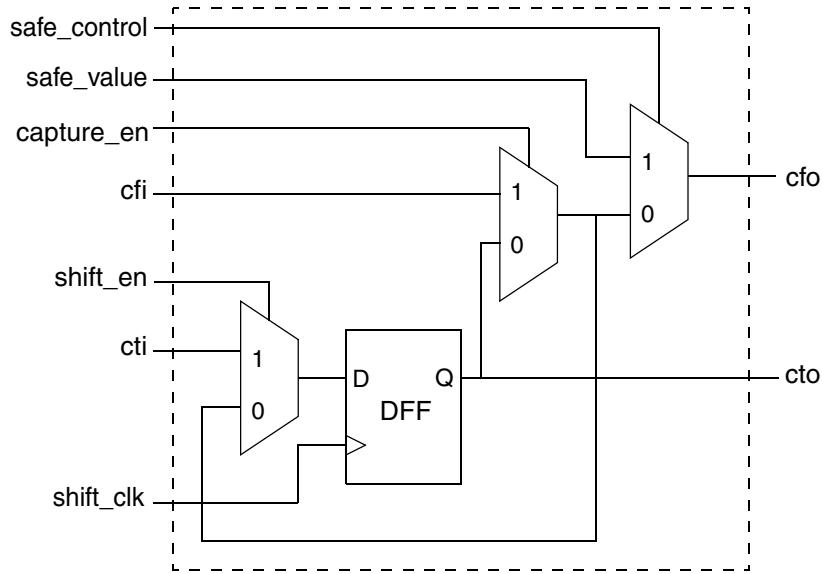
## Test Wrapper Cell Interface

DFT Compiler can insert a dedicated wrapper cell to wrap input and output ports, but if you have existing boundary registers, you can share them with the wrapper logic.

## Dedicated Wrapper Cell

Figure 8-8 illustrates the default dedicated WC\_D1 wrapper cell that DFT Compiler uses.

Figure 8-8 WC\_D1\_S Wrapper Cell



**Wrapper Cell Signals.** The interface to the test wrapper cell consists of the following signals:

cti: Core test input

This is the test input to the wrapper cell. It can come from either a primary input (if the cell is the first cell in the wrapper chain) or the cto signal of the previous wrapper cell in the chain.

cto: Core test output

This is the test output of the wrapper cell. It can drive either a primary output (if the cell is the last cell in the wrapper chain) or the cti signal of the next cell in the wrapper chain.

cfi: Core functional input

For input wrapper cells, this input is fed from the user-defined logic. For output wrapper cells, this input is fed from the core.

cfo: Core functional output

For input wrapper cells, this input drives the core. For output wrapper cells, this output drives the user-defined logic.

clk: Wrapper clock

This is controlled by the wrp\_clock control signal. It clocks the sequential elements in the wrapper cell.

**shift:** Shift enable

This is like a test enable to enable shifting through the sequential element of the wrapper cell. It is controlled differently for input and output wrapper cells. This prevents capture of undesirable values in wrapper cells during the test modes.

**capture\_en:** Capture enable

This signal, when low, enables capture of the cfi input.

**safe\_control:** Control signal

If a safe state is specified for a wrapper cell, this signal controls when the safe value is selected.

**safe\_value:** Logic value

If a safe state is specified for a wrapper cell, this is the safe state value.

Use the `set_wrapper_configuration` command to specify use of the WC\_D1 wrapper cell. The WC\_D1 cell is the default wrapper cell, so you usually do not need to specify its use.

```
dc_shell> set_wrapper_configuration \
           -class core_wrapper \
           -style dedicated -dedicated_cell_type WC_D1
```

The `set_wrapper_configuration` command applies to all ports in the design. To specify a wrapper cell for a subset of ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
           -type WC_D1 | WC_S1 | WC_D1_S -port_list list
```

## Sharing Existing Design Registers for Wrapping

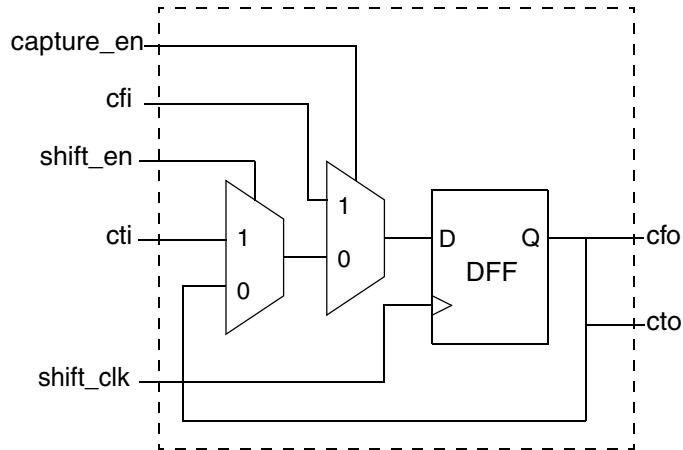
If your design has existing boundary I/O registers, the core wrapping feature enables you to reduce the area required for DFT by adding logic to the registers to create a wrapper cell.

In order for the core wrapping feature to share the existing register, the I/O register and the port must meet several conditions:

- The register data input or output must be connected to a boundary port with a wire or logic path. The logic path must be sensitized to produce a buffering or inverting effect, and the sensitization must be controlled by a test hold (static value of 0 or 1) on a primary input or output.
- The shared registers must be clocked by a functional clock. However, if the functional clock for the shared registers also clocks other functional registers internal to the core, it can cause problems, such as increased power usage or unknown data being clocked into the core, when the core is supposed to be isolated. For this reason, you should provide a separate functional clock for shared wrapper cells.

[Figure 8-9](#) shows the WC\_S1 wrapper cell. The signals are identical to the signals for the WC\_D1 wrapper cell described in [Figure 8-8 on page 8-9](#).

*Figure 8-9 WC\_S1 Wrapper Cell*

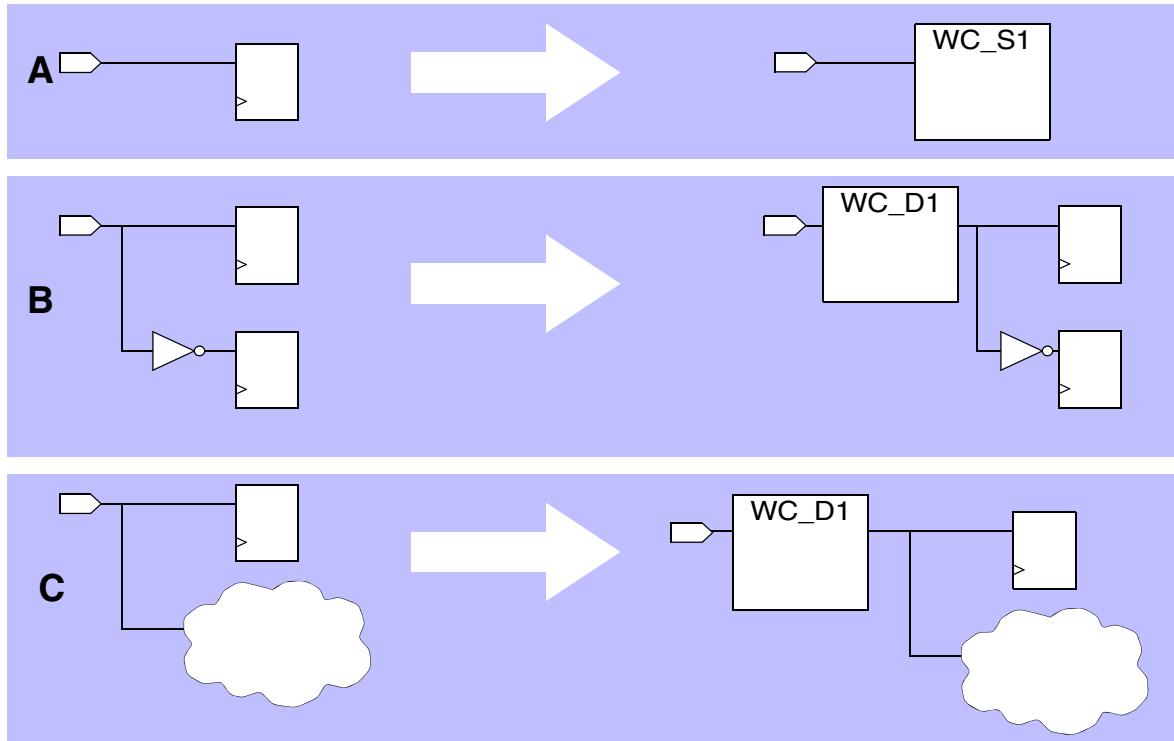


Use the `set_wrapper_configuration` command to specify use of the WC\_S1 wrapper cell. The WC\_S1 cell is the default shared wrapper cell, so you usually do not need to specify it.

```
dc_shell> set_wrapper_configuration \
           -class core_wrapper \
           -style shared -shared_cell_type WC_S1
```

If you specify the use of a shared wrapper cell, DFT Compiler inserts one wherever possible. However, not all I/O registers meet the requirements for shared registers. If a register does not meet the requirements, a dedicated wrapper cell is used instead. [Figure 8-10](#) illustrates some special cases of wrapping inputs.

Figure 8-10 Examples of Sharing Existing Design Registers



In case A, an I/O register is replaced with a shared wrapper cell. In cases B and C, however, shared wrapper cells would result in an unwrapped input, so a dedicated wrapper cell is created instead.

If you want to override the dedicated wrapper cell, which is inserted when a shared wrapper cell cannot be inserted (as in case C of Figure 8-10), you can use the `-dedicated_cell_type` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration \
    -class core_wrapper -style shared \
    -shared_cell_type WC_S1 \
    -dedicated_cell_type none
```

This command turns off wrapping with the dedicated wrapper cell.

The `-dedicated_cell_type` option applies to all ports within the design. To specify only a subset of the ports, use the `set_boundary_cell -class core_wrapper` command with the `-type` option.

Note:

You can turn off the use of a dedicated wrapper cell on a given port, but you cannot force a shared wrapper cell to be inserted if the I/O register does not meet the requirements for a shared wrapper cell.

By default, DFT Compiler replaces, or swaps, the entire I/O register with a WC\_S1 cell. You can use the `-register_io_implementation` option of the `set_wrapper_configuration` command to implement the functionality of the WC\_S1 cell by using discrete logic gates around the existing I/O register:

```
dc_shell> set_wrapper_configuration \
           -class core_wrapper -style shared \
           -shared_cell_type WC_S1 \
           -register_io_implementation in_place
```

When you swap an existing register with a shared wrapper cell, the new cell is placed at the same location in the hierarchy as the original register. If you are using the in-place implementation, the location of the original I/O register is not disturbed, but the additional MUX logic is placed in a separate hierarchy used for dedicated wrapper cells and mode logic.

DFT Compiler uses a dedicated wrapper clock for dedicated wrapper cells and a functional clock for shared wrapper cells. You can, instead, specify a single clock for both dedicated and shared wrapper cells using the following command:

```
set_wrapper_configuration
    -use_dedicated_wrapper_clock true
```

This command creates a port named `wrp_clock` and connects the dedicated wrapper clock to it and all wrapper cells. You can use the `set_dft_signal` command to use an existing port instead of creating the `wrp_clock` port:

```
dc_shell> set_dft_signal -view spec -type wrp_clock \
           -port port_name
```

The `set_scan_configuration -clock_mixing mix_clocks -test` mode command mixes dedicated and shared wrapper cells if required. When cells that use different clocks are mixed in a wrapper chain, lock-up latches are inserted between wrapper cells that do not use the same clock.

The `set_wrapper_configuration` command applies to all ports of the design. To specify the wrapper cell configuration for an individual port, use the `set_boundary_cell` command. To specify use of a shared wrapper cell for a list of specific ports, for example, use this command:

```
dc_shell> set_boundary_cell -class core_wrapper -type WC_S1 \
           -port_list {my_port1, my_port2, my_port3}
```

Note:

Typical flows should not have to use the `-type` option, because other settings direct the tool to use the supplied wrapper cell types as needed.

[Example 8-1](#) shows the preview report of the shared I/O registers for a sample design.

#### *Example 8-1 Preview Report of Shared I/O Registers*

```
dc_shell> preview_test -test_wrappers all
Test wrapper plan report
Design : coreJF
Version: 2004.12
Date : Wed Feb 2 12:00:12 2005
*****
Number of designs to be wrapped : 1
coreJF
Number of Wrapper Interface ports : 5
port type port name
-----
WRP_CLOCK wrp_clock
WRP_CLOCK ck1
WRP_CLOCK ck2
WRP_CLOCK ck3
WRP_SHIFT wrp_shift
Note: Dedicated wrapper cells are grouped into a hierarchical instance
named:
"coreJF_Wrapper_inst" (Module name: "coreJF_Wrapper_inst_design")
Wrapper Length: 10
      Wrapper   Wrapper   Control   Safe   Wrapper
Index Port Function Cell Type  Cell Impl  Value   Clock    Cell Name
----- -----  -----  -----  -----  -----  -----
9     A1   input    WC_S1   - SWP   -   ck1    I1_reg
8     A2   input    WC_D1   - SWP   -   wrp_clock coreJF_A2_wrp0_8
7     A3   input    WC_D1   - SWP   -   wrp_clock coreJF_A3_wrp0_7
6     A4   input    WC_D1   - SWP   -   wrp_clock coreJF_A4_wrp0_6
5     Q1   output   WC_S1   - SWP   -   ck1    Q1_reg
4     Q2   output   WC_S1   - SWP   -   ck2    iQ2_reg
3     Q3   output   WC_S1   - SWP   -   ck3    iQ3_reg
2     ck1  input    WC_D1   - SWP   -   wrp_clock coreJF_ck1_wrp0_2
1     ck2  input    WC_D1   - SWP   -   wrp_clock coreJF_ck2_wrp0_1
0     ck3  input    WC_D1   - SWP   -   wrp_clock coreJF_ck3_wrp0_0
```

---

## Delay Test Wrapper Insertion

Use the `-delay_test` wrapper insertion option to apply at-speed ATPG vectors to the wrapped core. The delay test wrapper flow uses the dedicated and shared wrapper cells but builds a different test controller. This test controller configures the wrapper chains into the various modes required to use transition patterns to perform at-speed test of the wrapped core and top-level user-defined logic associated with the core.

In a delay-test wrapper insertion flow, the `scan_enable` is gated with the test control module (TCM) to ensure that the input and output wrapper cells shift correctly in the various test modes that are created.

**Table 8-3** describes the shift and capture behavior of the delay wrapper cells for the various test modes.

*Table 8-3 Shift/Capture Behavior of the Wrapper Cells When Using the delay\_test Option*

	<b>Internal_scan</b>	<b>wrp_if</b>	<b>wrp_of</b>
Internal cells	SC <sup>a</sup>	SC	SC
Input wrapper cells	XX <sup>b</sup>	SS <sup>c</sup>	SC
Output wrapper cells	XX	SC	SS <sup>c</sup>

a. SC: Shift and capture

b. XX: Do not shift in shift/capture

c. SS: Shift in shift/capture cycles

To enable the insertion of delay wrapper cells, use the following command:

```
dc_shell> set_wrapper_configuration -delay_test true
```

## Delay Wrapper Test Modes

When delay wrapper insertion is enabled, the following wrapper test modes and default configurations are created:

### **Internal\_scan**

- Wrapper is transparent.

### **wrp\_if: wrapper with delay support for INTEST**

- Wrapper chains are configured in INTEST.
- Input wrapper chains are separate from output wrapper chains.
- Input chains shift in shift cycle and shift in capture cycle.
- Output chains shift in shift cycle and capture in capture cycle.
- Separate scan\_enable for input, output, and core chains.
- Used for launch-on-last-shift transition ATPG of wrapped core.

### **wrp\_of: wrapper with delay test support for EXTEST**

- Wrapper chains are configured in EXTEST.

- Core chains are not active.
- Input wrapper chains are separate from output wrapper chains.
- Input chains shift in shift and capture in capture cycles.
- Output chains shift in shift and shift in capture cycles.
- Separate `scan_enable` for input and output chains.
- Used for launch-on-last-shift transition ATPG of top-level user-defined logic.

#### **wp\_safe: wrapper safe**

- Wrapper has safe applied values.

## **Separate Input and Output Wrapper Chains and Control**

The previous section, “[Delay Wrapper Test Modes](#),” describes a way to construct an architecture which *automatically* segregates input and output wrapper chains and allows at-speed pattern generation, using the wrappers. However, several features of the wrapper client allow you to specify wrapper scan chains that separate the input and output wrapper cells. In addition, you can create individual scan-enable controls for these chains.

The process by which you can create these special chain configurations begins by creating two more scan-enable signals for the input and output wrapper chains, using the commands

```
set_dft_signal -type wrp_shift -port wrp_ishift -test_mode all_dft
set_dft_signal -type wrp_shift -port wrp_oshift -test_mode all_dft
```

Next, you associate each scan-enable signal with its respective wrapper chain segments as follows:

```
set_wrapper_configuration -class core_wrapper \
    -chain_count 2 -mix_cells false \
    -input_shift_enable wrp_ishift \
    -output_shift_enable wrp_oshift
```

The `-mix_cells` option of this command, when set to false, insures that input and output wrapper cells are not mixed in the same wrapper segments. In this command example, two wrapper chain segments will be constructed. However, you can allocate more wrapper chains, if desired. Note that the `wrp_ishift` signal will be used as the wrapper shift signal for the wrapper chain segment associated with the input wrapper cells. Similarly, the `wrp_oshift` signal will be used to control the wrapper chains composed of output wrapper cells only.

In addition, further architectural constraints can be set by using the `set_scan_path` command. For example,

```
set_scan_path input_wrapper_only_seg \
    -input_wrapper_cells_only_enable
    -scan_enable wrp_ishift
    -test_mode all

set_scan_path output_wrapper_only_seg \
    -output_wrapper_cells_only_enable
    -scan_enable wrp_oshift
    -test_mode all
```

would create two wrapper segments that segregate the input and output wrapper cells and automatically control these segments for proper balancing.

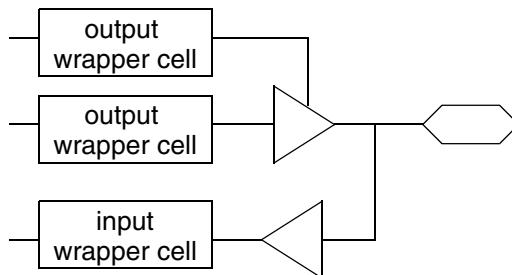
That is, DFT Compiler can usually create properly balanced wrapper segments automatically for typical flows with multiple wrapper segments. Also remember to use the `-test` option in these commands to insure that control is established in the correct modes. You might need to define the test modes before assigning synthesis constraints to them, using the `test_mode` command.

---

## Wrapping Three-State and Bidirectional Ports

To prevent contention during core integration, three-state and bidirectional ports are wrapped differently from regular input and output ports. The wrapper cells are inserted in a different location, as shown in [Figure 8-11](#), which permits selective disabling of three-state drivers.

*Figure 8-11 Three-State or Bidirectional Port Wrapping*



Specifying a safe value for a three-state port has no effect, because DFT Compiler automatically inserts safe-value logic on the enable pin of the three-state driver. The safe value depends on the sense (inverted or noninverted) on the enable pin, but it holds the three-state driver in a high impedance state.

DFT Compiler applies a safe value specified for a bidirectional port to the output pin of the three-state driver.

Degenerate three-state and bidirectional ports are not wrapped.

---

## Specifying Multiple Wrapper Modes

DFT Compiler allows the construction of scan chain and compression architectures that are differently composed for various test modes of operation. Specifically, you can specify more than the default modes provided by the default feature set, modifying the specifications defined by the default scan chain and compression architecting. These types of modifications are considered multimodal, and are discussed in the Using Multimode Scan Architecture section of Chapter 7, Advanced DFT Architecture Methodologies, of this user guide.

---

## Test Wrapper Exceptions

Certain ports are not wrapped, and you can manually exclude any port when you do not want a wrapper inserted for it.

The following ports are excluded from wrapping:

- Any port specified as a global test signal
- Any port with a test hold attribute specified
- Functional and test clocks
- Any asynchronous set or reset signal.

You can exclude output ports from being wrapped by using the `set_boundary_cell` command after using the `set_wrapper_configuration` command.

```
dc_shell> set_wrapper_configuration \
           -class core_wrapper \
           -style dedicated -dedicated_cell_type WC_D1

dc_shell> set_boundary_cell -class core_wrapper \
           -function output \
           -port {my_port4, my_port5, my_port6} \
           -type none
```

Because doing this reduces test coverage, you should exclude output ports only when necessary. For example, wrapping clocked primary outputs leads to a potential race condition. You should never exclude input ports other than test signals, test holds, clocks, and other asynchronous signals from being wrapped.

---

## Specifying Wrapper Chains

You can specify the insertion of multiple wrapper chains for a core, you can specify the order of cells in a wrapper chain, and you can specify the wrapper scan inputs and outputs.

### Specifying Multiple Wrapper Chains

When specifying a chain count or maximum length, you must have defined a test mode with the `define_test_mode` command and linked it to the core wrapper logic with the `set_scan_configuration` command. For example,

```
set_scan_configuration -chain_count 8 -test_mode wrp_if  
set_scan_configuration -chain_count 4 -test_mode wrp_of
```

You can use the `set_scan_configuration -chain_count -test_mode` command to specify the number of chains for DFT Compiler to insert. The number of chains you specify should include both scan chains and wrapper chains.

By default, the number of wrapper chains is one if there are no shared wrapper cells created for the design, or if you specify the `-use_dedicated_wrapper_clock true` or the `-clock_mixing` options. If there are shared wrapper cells in the design, the default number of wrapper chains is the number of wrapper clocks used in the design, including both dedicated and shared functional clocks.

The `set_scan_configuration -max_length -test_mode` command enables you to specify a maximum number of wrapper cells for each wrapper chain. This number includes both wrapper chains and scan chains. DFT Compiler then creates as many wrapper chains as necessary to keep the length of any chain below the specified limit.

Do not specify both the `-chain_count` and `-max_length` options. If you do, `-max_length` has precedence, and DFT Compiler ignores the `-chain_count` argument.

When specifying a chain count or a maximum length for a wrapper chain, you must define a test mode with the command and link it to the core wrapper logic.

### Specifying Wrapper Cell Order

Note:

When using the `set_scan_path -class wrapper` command, you must have defined a test mode with the `define_test_mode` command and linked it to the core wrapper logic with the `set_dft_logic_usage` command.

Use the `set_scan_path -class wrapper -test_mode` command to specify the order in which the ports in a wrapper chain are connected.

```

set_scan_path -class wrapper scan_chain_name
    [-ordered_elements ordered_port_list]
    [-complete true | false]
    [-input_wrapper_cells_only enable | disable]
    [-output_wrapper_cells_only enable | disable]
    [-scan_enable se_port]
    [-test_mode test_mode_name]
    [-set_data_in si_port]
    [-scan_data_out so_port]

```

By default, DFT Compiler can add additional wrapper cells to the beginning of the specified chain, unless you set the `-complete` option to `true`. You cannot use this command to add cells to an already specified chain because the last specification overwrites any previous specification. Wrapper cells cannot belong to more than one chain, so if you specify a cell as belonging to more than one chain, the last one specified takes precedence.

Use the `remove_scan_path -view name -chain chain_name -test_mode mode_name` command to remove the specified wrapper or scan chain.

**Specifying Wrapper Cells for Bidirectional and three-State Ports.** As shown in [Figure 8-11 on page 8-17](#), multiple wrapper cells are inserted for three-state and bidirectional ports. You can reference these wrapper cells in the ordered list of ports by appending `/out`, `/en`, or `/in` to the name of the bidirectional or three-state port. For example, the following command specifies an ordering for ports named a, b, c, and d, where a is an input port, b is a three-state port, and c and d are bidirectional ports.

```

dc_shell> set_scan_path Wchain 0 -class wrapper \
    -ordered_elements [list "a" "b/in" "b/out" "c/out" \
    "d/out" "b/en""c/en" "d/en"] -complete true
    -test_mode test_mode_name

```

Note:

All of the wrapper cells for an individual three-state or bidirectional port must be in the same wrapper chain. For example, you cannot specify that the enable wrapper cell be in a different chain than the output wrapper cell for that port.

## Specifying Wrapper Chain Ports

Use the `set_dft_signal` command to specify the wrapper scan-in or wrapper scan-out signals for a chain:

```

dc_shell> set_scan_path WC1 -class wrapper WC1 \
    -ordered_elements [list "a" "b" "c"] \
    -test_mode test_mode_name
dc_shell> set_dft_signal test_scan_in -port wsi \
    -chain WC1
dc_shell> set_dft_signal -type ScanDataOut -port wso \
    -chain WC1

```

## Specifying Wrapper Chain Length

By default, wrapper scan chains are optimized along with other scan chains in order to balance the scan chains appropriately. To specify further details about wrapper chains, the `set_wrapper_configuration` command includes the `-max_length` and `-mix_cells` options.

The command syntax is

```
set_wrapper_configuration  
    [-max_length <maximum_length>]  
    [-chain_count <count>]  
    [-mix_cells true | false]
```

The `-max_length` option specifies the maximum length of any wrapper chain, and the `-count` option specifies the number of requested wrapper. The `-mix_cells` option specifies whether input wrapper cells will be mixed in the scan chains with output wrapper cells. The default is true, which means the two kinds of wrapper cells can be mixed.

## Specifying Input-Only or Output-Only Wrapper Chains

To create wrapper chains that contain either all-input or all-output cells, specify the following commands:

```
set_scan_path -class wrapper chain_name  
    [-input_wrapper_cells_only enable | disable]  
    [-output_wrapper_cells_only enable | disable]  
    -test_mode test_mode_name  
  
set_scan_configuration  
    [-chain_count n | -max_length n]  
    [-test_mode test_mode_name]
```

The number of wrapper chains that contain only inputs or only outputs depends on the `-chain_count` and `-max_length` options.

If shared wrapper cells are used in wrapping the design, ensure that all shared wrapper cells in any wrapper chain use the same system clock. If the shared wrapper cells use different system clocks, do one of the following to allow the wrapper cells in the wrapper chain:

- Specify a dedicated wrapper clock with

```
set_wrapper_configuration \  
    -use_dedicated_wrapper_clock TRUE
```

- Specify that clock mixing is allowed with

```
set_scan_configuration -clock_mixing mix_clocks \  
    -test_mode test_mode_name
```

In this case, core wrapping will insert lock-up latches between the shared wrapper cells that use different clocks. Lock-up latches are also inserted for internal scan chains.

## Controlling Wrapper Chains with Scan Inputs, Scan Outputs, and Shift Enables

Control a user-defined wrapper chain with a specific wrapper scan-in, scan-out, or scan-enable signal with the `set_dft_signal` and `set_scan_path` command.

## Automatically Create Additional Wrapper Modes With Separate Wrapper Chains

To automatically create additional wrapper INTEST and EXTEST modes with separate wrapper chains for inputs and outputs, specify the `-delay_test true` option with the `set_scan_path` command. Input and output wrapper chains for these modes are controlled by different wrapper shift-enables. For example,

```
define_test_mode wrp_if_new -usage wrp_if
set_dft_signal -view spec -type ScanDataIn -port wrp_si1
set_dft_signal -view spec -type ScanDataOut -port wrp_so1
set_dft_signal -view spec -type wrp_shift -port wrp_shift_in
set_scan_path WCH_1 -class wrapper \
    -ordered_elements list_elements \
    -scan_enable wrp_shift_in -scan_data_in wrp_si1 \
    -scan_data_out wrp_so1 -test_mode wrp_if_new
```

---

## Core Wrapping Flows

To insert a core wrapper, use one of these flows:

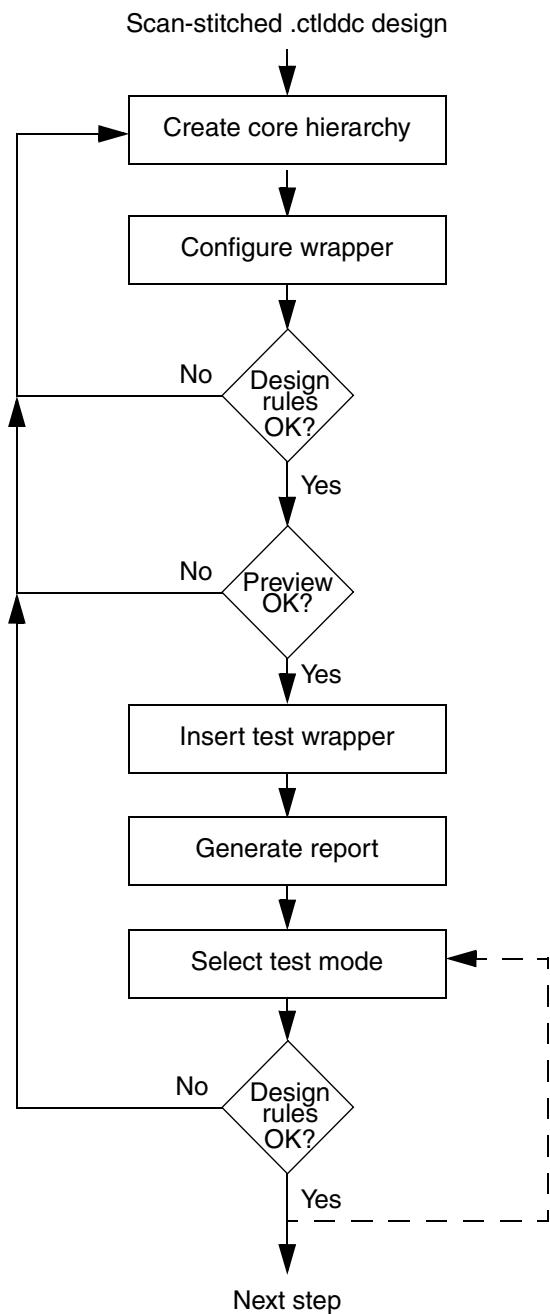
- [Scan-Stitched Core Flow](#)
- [Core Creation Flow](#)

---

### Scan-Stitched Core Flow

To wrap a scan-replaced and routed design, you should use the scan-stitched core flow. [Figure 8-12](#) illustrates this flow.

Figure 8-12 Scan-Stitched Core Flow Diagram



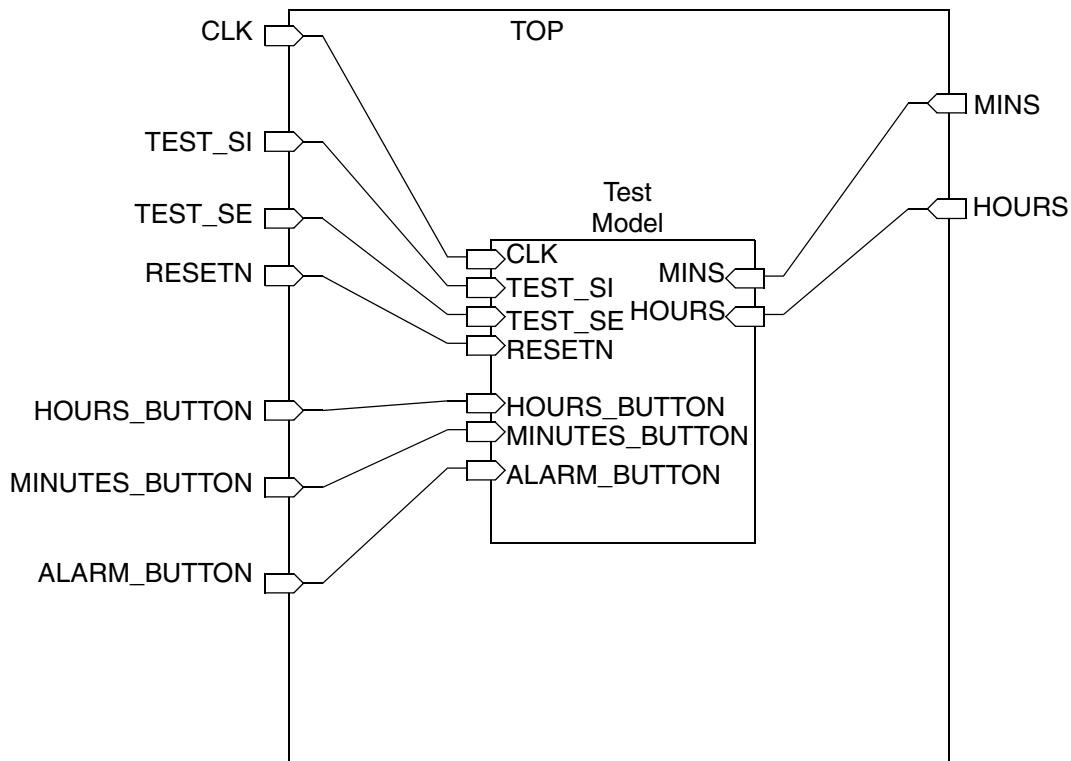
To wrap a scan-stitched core, follow these steps. Start the flow with a test model of the core. If you use a test model, DFT Compiler cannot touch the scan logic in the model during wrapping.

1. Create the core hierarchy.

[Figure 8-13](#) shows the hierarchy you should create. You must create an empty top-level design with the same input and output pins as the core design, instantiate the test model within the top-level design, and connect all of the input and output pins. DFT Compiler inserts the wrapper in the top-level design.

You should use a text editor to create RTL code to accomplish this step.

*Figure 8-13 Creating Core Hierarchy*



2. Configure the wrapper.

Use the `set_dft_configuration -wrapper enable` command to specify core wrapping. If you do not issue this command, the other core wrapping commands will not have any effect. To set the default configuration for wrapping, use the `set_wrapper_configuration` command. You can override the wrapper configuration on any ports by using the `set_boundary_cell` command.

```

dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration \
           -class core_wrapper -style dedicated \
           -dedicated_cell_type WC_D1

dc_shell> set_boundary_cell -class core_wrapper \
           -function input
           -port {my_port1, my_port2, my_port3} \
           -type none

dc_shell> set_boundary_cell -class core_wrapper \
           -function output
           -port {my_port4, my_port5, my_port6} \
           -type none

```

### 3. Set the wrapper signals and configure the wrapper chains.

By default, control signals are added to the design to control the wrapper configuration. Use the `set_dft_signal` command to set the names and connections of these signals.

You can also use the `set_wrapper_configuration -chain_count` or `-max_length` option to control the number or length of the wrapper scan chains and the `set_scan_path` to specify the order of the wrapper cells.

```

dc_shell> set_dft_signal -view spec \
           -type wrp_clock -port my_wclock

dc_shell> set_scan_configuration \
           -chain_count 3 -test_mode test_mode_name

dc_shell> set_scan_path \
           -class wrapper "chain0" \
           -ordered_elements [list "in1" "in2"] \
           -test_mode test_mode_name

```

### 4. Check test design rules.

Use the `dft_drc` command to check test design rules.

```

dc_shell> create_test_protocol
dc_shell> dft_drc

```

### 5. Preview the wrapper and scan cells before inserting them.

Use the `preview_dft` command to report on the wrapper and scan cells before you actually insert them.

```
dc_shell> preview_dft -test_wrapper all
```

### 6. Insert the wrapper cells.

Use the `insert_dft` command to insert the wrapper cells into the design and stitch the wrapper chain.

```
dc_shell> insert_dft
```

7. (Optional) Select the test mode.

You should check the design rules for each test mode created. Use the `current_test_mode` command to set the test mode to each of the modes of operation:

```
dc_shell> current_test_mode wrp_if
```

8. Check that the design is ready for ATPG by using the `dft_drc` command.

```
dc_shell> dft_drc
```

This verifies that the scan chains and wrapper cells operate correctly for the current test mode. You can repeat this step for additional test modes.

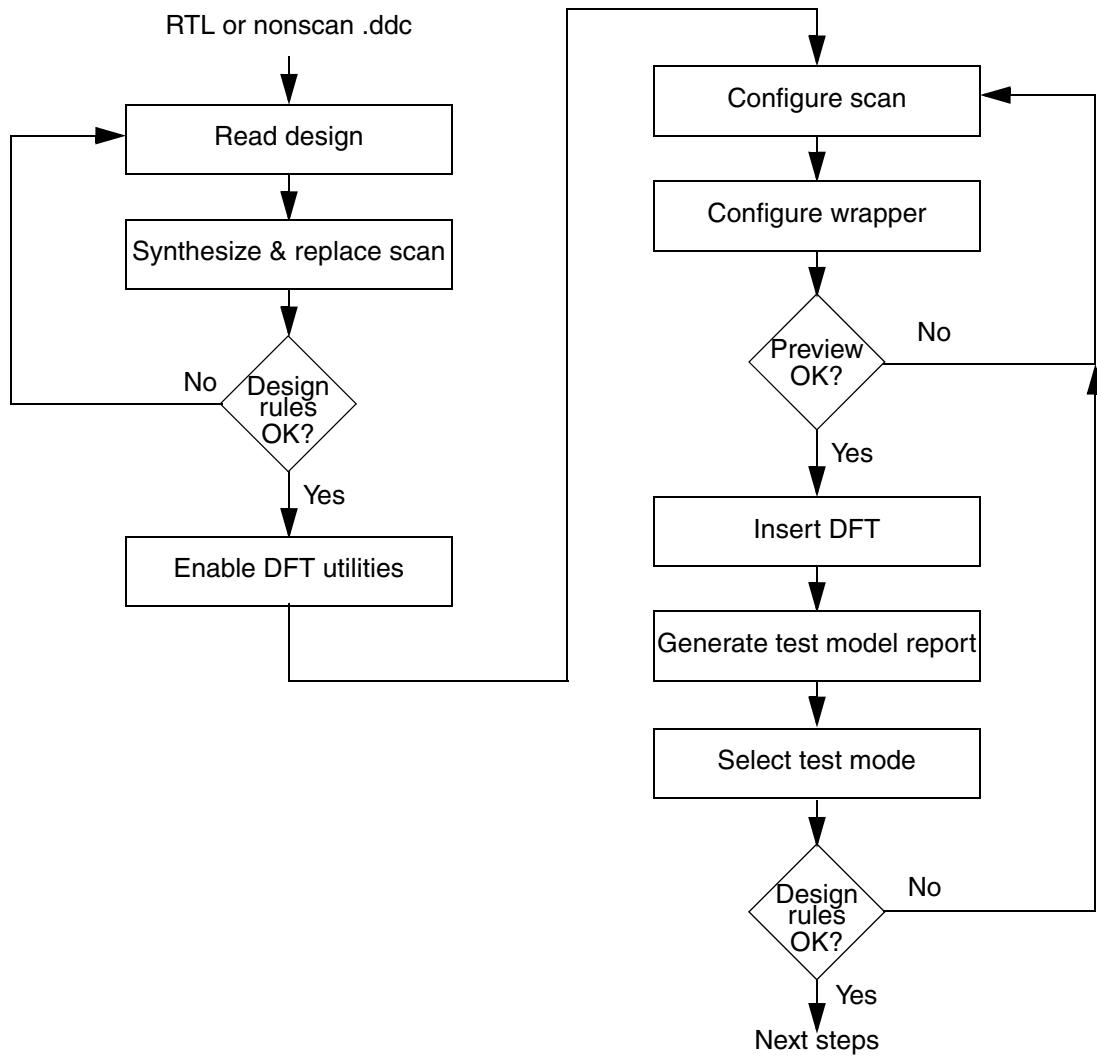
---

## Core Creation Flow

To wrap a design without scan, use the core creation flow.

[Figure 8-14](#) shows a core creation flow diagram.

Figure 8-14 Core Creation Flow Diagram



To create and wrap a core, follow these steps:

1. Read the design.

Use the Design Compiler `read_file` command to read in a .ddc design or an RTL netlist.

```

dc_shell> read_file -format ddc my_design.ddc
dc_shell> read_file -format verilog my_design.v
dc_shell> read_file -format vhdl my_design.vhdl

```

2. Perform synthesis and scan replacement.

Use the `compile -scan` command to perform synthesis and scan replacement.

```
dc_shell> compile -scan
```

For more information, see [Chapter 4, “Performing Scan Replacement.”](#)

3. Enable the DFT utilities.

Use the `set_dft_configuration` command to enable core wrapping and other desired DFT functions, such as AutoFix and Shadow LogicDFT.

```
dc_shell> set_dft_configuration -wrapper enable
```

4. Set the scan configuration.

```
dc_shell> set_scan_configuration \
           -multiplexed_flip_flop \
           -test_mode test_mode_name

dc_shell> set_dft_signal -type ScanEnable -port SE

dc_shell> set_dft_signal -view existing_dft \
           -type Constant -active_state 1 port TM

dc_shell> set_dft_signal -view existing_dft \
           -type ScanClock -timing {45 55} -port CLK
```

Specify the necessary configuration. For more information, see [Chapter 4, “Performing Scan Replacement.”](#)

5. Configure the wrapper.

Use the `set_wrapper_configuration` command to set the default configuration for wrapping. Override the wrapper configuration on any ports by using the `set_boundary_cell` command.

```
dc_shell> set_wrapper_configuration \
           -class core_wrapper -style dedicated
           -dedicated_cell_type WC_D1 -safe_state 1
dc_shell> set_boundary_cell -class core_wrapper \
           -function input -port {ASYNC1} -type none
```

6. Set the wrapper signals.

By default, control signals are added to the design to control the wrapper configuration. Use the `set_dft_signal` command to set the names and connections of these signals.

```
dc_shell> set_dft_signal -view spec \
           -type wrp_clock -port my_wclock
```

7. Check design rules and preview the wrapper and scan cells before inserting them.

Use the `preview_dft` command to report on the wrapper and scan cells before you actually insert them.

```
dc_shell> create_test_protocol  
dc_shell> dft_drc  
dc_shell> preview_dft -test_wrapper all
```

8. Insert the test wrapper.

Use the `insert_dft` command to insert DFT structures and to insert and stitch the test wrapper around the design.

```
dc_shell> insert_dft
```

9. (Optional) Select the test mode.

You can specify a test mode on which to check test design rules. You should check design rules for each of the test modes.

```
dc_shell> current_test_mode wrp_safe
```

10. Check test design rules.

After inserting scan, use the `dft_drc` command to check whether there are problems with the scan chains and the test wrapper by verifying that they operate properly.

```
dc_shell> dft_drc
```

Continue specifying test modes and checking design rules for those modes until all test modes are checked.

---

## Core Wrapping Scripts

The following sample scripts illustrate core wrapping.

---

### Core Wrapping With a Dedicated Wrapper

[Example 8-2](#) wraps a core with a dedicated wrapper.

#### *Example 8-2 Sample Script for Dedicated Wrapper*

```
read_ddc ddc/des_unit.ddc  
current_design des_unit  
uniquify
```

```

link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port clk_st

set test_enable_dft_drc true

# enable and configure wrapper client
set_dft_configuration -wrapper enable
set_wrapper_configuration -class core_wrapper \
    -style dedicated \
    -use_dedicated_wrapper_clock true \
    -safe_state 1

#Set scan chain count as desired
set_scan_configuration -chain_count 10

#Create the test protocol and run pre-drc
create_test_protocol
dft_drc -v

#Report the configuration of the wrapper utility, optional
report_wrapper_configuration

#preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

#Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode Internal_scan
report_scan_path -view existing -cell all > reports/
xg_wrap_dedicated_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing -cell all > reports/xg_wrap_dedicated_wrp_of.rpt

current_test_mode wrp_if
report_scan_path -view existing -cell all > reports/xg_
wrap_dedicated_wrp_if.rpt

report_dft_signal -view existing_dft -port *

report_area

change_names -rules verilog -hier

write -format ddc -hier -output ddc/scan.ddc
write -format verilog -hier -output vg/scan_wrap.vg
write_test_model -o ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan -o stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if -o stil/wrp_if.spf
write_test_protocol -test_mode wrp_of -o stil/wrp_of.spf

```

---

## Core Wrapping With a Shared Wrapper

[Example 8-3](#) wraps a core with a shared wrapper.

*Example 8-3 Sample Script for Shared Wrapping*

```
read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port clk_st

set test_enable_dft_drc true
# enable and configure wrapper client
set_dft_configuration -wrapper enable

#Configure for shared wrappers, using existing cells and create glue logic around
existing cells
set_wrapper_configuration -class core_wrapper \
    -style shared \
    -shared_cell_type WC_S1 \
    -use_dedicated_wrapper_clock true \
    -safe_state 1 \
    -register_io_implementation in_place

#Set scan chain count as desired
set_scan_configuration -chain_count 10

#Create the test protocol and run pre-drc
create_test_protocol
dft_drc -v

#Report the configuration of the wrapper utility, optional
report_wrapper_configuration

#preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

#Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode Internal_scan
report_scan_path -view existing -cell all > reports/
xg_wrap_dedicated_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing -cell all > reports/xg_wrap_dedicated_wrp_of.rpt

current_test_mode wrp_if
report_scan_path -view existing -cell all > reports/xg_wrap_dedicated_wrp_if.rpt

report_dft_signal -view existing_dft -port *
```

```

report_area

change_names -rules verilog -hier

write -format ddc -hier -output ddc/scan.ddc
write -format verilog -hier -output vg/scan_wrap.vg
write_test_model -o ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan -o stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if -o stil/wrp_if.spf
write_test_protocol -test_mode wrp_of -o stil/wrp_of.spf

```

---

## Core Wrapping With Dedicated Delay Wrapper

[Example 8-4](#) wraps a core with a dedicated delay wrapper.

*Example 8-4 Sample Script for Dedicated Delay Wrapping*

```

read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port clk_st
set test_enable_dft_drc true

# enable and configure wrapper client
set_dft_configuration -wrapper enable
set_wrapper_configuration -class core_wrapper \
    -style dedicated \
    -use_dedicated_wrapper_clock true \
    -safe_state 1 \
    -delay_test true
#Set scan chain count as desired
set_scan_configuration -chain_count 10

#Create the test protocol and run pre-drc
create_test_protocol
dft_drc -v

#Report the configuration of the wrapper utility, optional
report_wrapper_configuration

#preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

#Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode Internal_scan

report_scan_path -view existing -cell all > \

```

```

reports/xg_wrap_dedicated_delay_path_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing -cell all >
    reports/xg_wrap_dedicated_delay_path_wrp_of.rpt

current_test_mode wrp_of_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_delay.rpt

current_test_mode wrp_of_scl_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_scl_delay.rpt

current_test_mode wrp_if
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if.rpt

current_test_mode wrp_if_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if_delay.rpt

current_test_mode wrp_if_scl_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if_scl_delay.rpt

report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hier

write -format ddc -hier -output ddc/scan.ddc
write -format verilog -hier -output vg/scan_wrap.vg
write_test_model -o ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan -o stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if -o stil/wrp_if.spf
write_test_protocol -test_mode wrp_if_delay -o stil/wrp_if_delay.spf
write_test_protocol -test_mode wrp_if_scl_delay -o stil/wrp_if_scl_delay.spf
write_test_protocol -test_mode wrp_of -o stil/wrp_of.spf
write_test_protocol -test_mode wrp_of_delay -o stil/wrp_of_delay.spf
write_test_protocol -test_mode wrp_of_scl_delay -o stil/wrp_of_scl_delay.spf

```

## Core Wrapping With Shared Delay Wrapper

[Example 8-5](#) wraps a core with a shared delay wrapper.

### *Example 8-5 Sample Script for Shared Delay Wrapping*

```

read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port clk_st
set test_enable_dft_drc true

```

```

# enable and configure wrapper client
set_dft_configuration -wrapper enable

#Configure for shared wrappers, using existing cells and create glue logic around
existing cells
set_wrapper_configuration -class core_wrapper \
    -style shared \
    -shared_cell_type WC_S1 \
    -use_dedicated_wrapper_clock true \
    -safe_state 1 \
    -register_io_implementation in_place \
    -delay_test true

#Set scan chain count as desired
set_scan_configuration -chain_count 10

#Create the test protocol and run pre-drc
create_test_protocol
dft_drc -v

#Report the configuration of the wrapper utility, optional
report_wrapper_configuration

#preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

#Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none
insert_dft

current_test_mode Internal_scan

report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of.rpt

current_test_mode wrp_of_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_delay.rpt

current_test_mode wrp_of_scl_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_scl_delay.rpt

current_test_mode wrp_if
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if.rpt

current_test_mode wrp_if_delay
report_scan_path -view existing -cell all > \
    reports/
xg_wrap_dedicated_delay_path_wrp_if_delay.rpt

```

```

current_test_mode wrp_if_scl_delay
report_scan_path -view existing -cell all > \
    reports/xg_wrap_dedicated_delay_wrp_if_scl_delay.rpt

report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hier

write -format ddc -hier -output ddc/scan.ddc
write -format verilog -hier -output vg/scan_wrap.vg
write_test_model -o ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan -o stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if_delay -o stil/wrp_if_delay.spf
write_test_protocol -test_mode wrp_if_scl_delay -o stil/wrp_if_scl_delay.spf
write_test_protocol -test_mode wrp_if -o stil/wrp_if.spf
write_test_protocol -test_mode wrp_of -o stil/wrp_of.spf
write_test_protocol -test_mode wrp_of_delay -o stil/wrp_of_delay.spf
write_test_protocol -test_mode wrp_of_scl_delay -o stil/wrp_of_scl_delay.spf

```



# 9

## On-Chip Clocking Support

---

On-Chip Clocking (OCC) support is common to all scan ATPG (Basic-Scan and Fast-Sequential) and adaptive scan environments. This implementation is intended for designs that require ATPG in the presence of phase-locked loop (PLL) and clock controller circuitry.

OCC support includes phase-locked loops, clock shapers, clock dividers and multipliers and so on. In the scan-ATPG environment, scan chain `load_unload` is controlled through an automatic test equipment (ATE) clock. However, internal clock signals that reach state elements during capture are PLL-related.

OCC flows can use either the user-defined clock controller and clock chains or the Synopsys clock controller IP. If you use an existing user-defined clock controller, you would need a set of user-defined commands to identify the existing clock controller outputs with their corresponding control bits.

This chapter includes the following sections:

- [Background](#)
- [Supported Flows](#)
- [Definitions](#)
- [Capabilities](#)
- [Limitations](#)
- [Design Flows](#)

- Basic Processes
- OCC Supported Flows
- Reporting Clock Controller Information
- DRC Support
- Example Configurations on a Design
- Waveform and Capture Cycle Example

Note:

This manual covers flows that are intended for a DFT-to-TetraMAX implementation. For information on using OCC in a non-DFT-to-TetraMAX implementation, see the *TetraMAX ATPG User Guide*.

---

## Background

At-speed testing for deep-submicron defects requires not only more complex fault models for ATPG and fault simulation, such as transition faults and path delay faults, but also requires the accurate application of two high-speed clock pulses to apply the tests for these fault models. The time delay between these two clock pulses, referred to as the launch clock and the capture clock, is the effective cycle time at which the circuit will be tested.

A key benefit of scan-based at-speed testing is that only the launch clock and the capture clock need to operate at the full frequency of the device under test. Scan shift clocks and shift data can operate at a much slower speed, thus reducing the performance requirements of the test equipment. However, complex designs often have many different high-frequency clock domains, and the requirement to deliver a precise launch and capture clock for each of these from the tester can add significant or prohibitive costs to the test equipment. Furthermore, special tuning is often required for properly controlling the clock skew to the device under test.

One common alternative for at-speed testing is to leverage existing on-chip clock generation circuitry. This approach uses the active controller, rather than off-chip clocks from the tester, to generate the high-speed launch and capture clock pulses. This type of approach generally reduces tester requirements and cost and can also provide high-speed clock pulses from the same source as the device in its normal operating mode without additional skews from the test equipment or test fixtures.

For using this approach, additional on-chip controller circuitry is included to control the on-chip clocks in test mode. The on-chip clock control is then verified, and at-speed test patterns are generated that apply clocks through proper control sequences to the on-chip clock circuitry and test mode controls. DFT Compiler and TetraMAX support a comprehensive set of features to ensure that

- The test mode control logic for the OCC operates correctly and has been connected properly
- Test mode clocks from the OCC circuitry can be efficiently used by TetraMAX for at-speed test generation
- OCC circuitry can operate asynchronously to shift other clocks from the tester
- TetraMAX patterns do not require additional modifications to use the OCC and to run properly on the tester

---

## Supported Flows

OCC is supported in the following flows:

- Basic scan flow
- Top-down, nonhierarchical adaptive scan insertion flow
- Bottom-up basic scan flow with OCC controller stitching at the top level
- Bottom-up hierarchical adaptive scan synthesis flow, which replaces subblock gate level with test models that are OCC controller stitched at the top level)
- Hierarchical adaptive scan synthesis and hybrid flow, which stitch the cores at the top level with integration at the top level

---

## Definitions

Note the following definitions as they apply to OCC in this chapter:

- **Reference clocks** – The frequency reference to the PLL. It must be maintained as a constantly pulsing and free-running oscillator or the circuitry will lose synchronization.
- **PLL clocks** – The output of the PLL. A free-running source that also runs at a constant frequency which might or might not be the same as the reference clock.
- **ATE clocks** – Shifts the scan chain, typically more slowly than a reference clock. This signal must preexist, or you must manually add this signal (that is, port) when inserting the OCC. Usually the ATE clock is not used as a reference clock, but it must be treated as a free-running oscillator so that it cannot capture predictable data. The ATE clock is said to be a dual clock signal when the same port drives both the ATE clock and the reference clock.
- **Internal clocks** – The OCC is responsible for gating and selecting the PLL and ATE clocks and for creating the internal clocks to satisfy ATPG requirements.
- **External clocks** – The primary clock inputs of a design that directly clock the flip-flops through the combinational logic not generated from PLLs.

---

## Capabilities

The following OCC support features are available:

- Synthesis of individual or multiple clock controllers and clock chains as Synopsys clock controller IP

Note:

The PLL controller that is included with DFT Compiler is an example that is not guaranteed to be appropriate for use in your design. If you decide to use this design, you are responsible for validating that this functionality works in the context of your design.

- Support of pre-DFT DRC, scan-chain stitching, and post-DFT DRC in documented OCC support flows
- Support of a PLL-bypass configuration when an external (ATE) clock is used for capture, thus bypassing the PLL clock(s)
- Generation of STIL protocol files with internal clock control details for use with TetraMAX
- Support of post-DFT DRC, scan chain shifting, and adaptive scan
- Support of user-defined clock controller logic and clock chains that are already instantiated in the design

---

## Limitations

Note the following limitations:

- Inferencing internal PLL or any reference clocks is not supported. For prescan DRC, you must explicitly define your reference clock, ATE clock, and PLL clocks.
- The OCC controller generated by DFT Compiler does not test faults across internal clock domains.
- When inserting new OCC controllers into your design, you must synthesize and insert multiple clock OCC controllers with a single `insert_dft` command.
- The only supported scan style is multiplexed flip-flop.
- Bottom-up stitching of OCC controller signals is not supported.
- Post DRC is not supported in Hierarchical Adaptive Scan Synthesis (HASS) flows.

- For adaptive scan, clock chains are not shared with internal scan chains; that is, clock chain flip-flops are not mixed with design scan flip-flops. In adaptive scan, clock chain flip-flops are connected to dedicated scan inputs and scan outputs and are outside the decompressor to compressor path.
- Fast-sequential patterns with OCC support cannot measure the primary outputs between system pulses. The measure primary output is placed before the first system pulse and measures only Xs. You have to use pre-clock-measure, with the strobe being placed before the clock.
- External clocks, which have a direct connection to scan flip-flops, cannot serve as ATE clocks for the OCC controller.
- The `set_dft_clock_controller -ateclocks` command accepts only one port. The user can have multiple OCC controllers, but only one port can be specified at the `-ateclocks` switch per controller.
- Postclock strobe measure (that is, end-of-cycle measure) is not compatible with PLL reference clocks.
- Use the `set_clock_gating_check` command to perform a manual clock gating check. This check is needed to check the timing between the fast-clock-enable registers and the “FastClock” gates (multiplexers between the fast clocks and the slow clocks), because, by default, PrimeTime does not perform clock gating checks on multiplexers.

## Design Flows

[Figure 9-1](#) and [Figure 9-2](#) show a high-level view of an on-chip clock generator with accompanying clock control circuitry designed for at-speed scan chain clocking relevant in DFT Compiler and TetraMAX.

Note:

This implementation is intended only as an example and might not satisfy all requirements.

Figure 9-1 OCC and Clock Chain Synthesis Insertion Flow

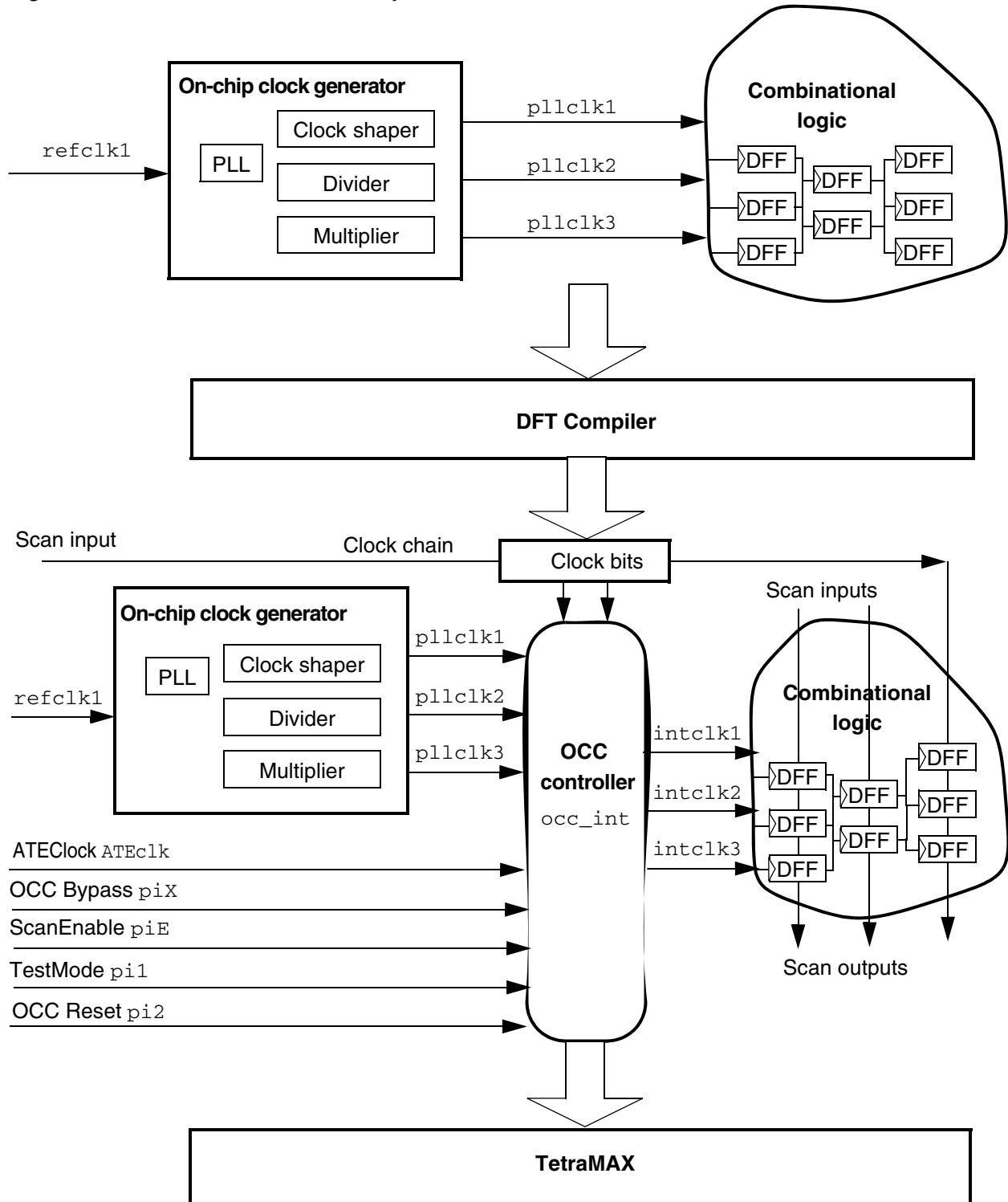
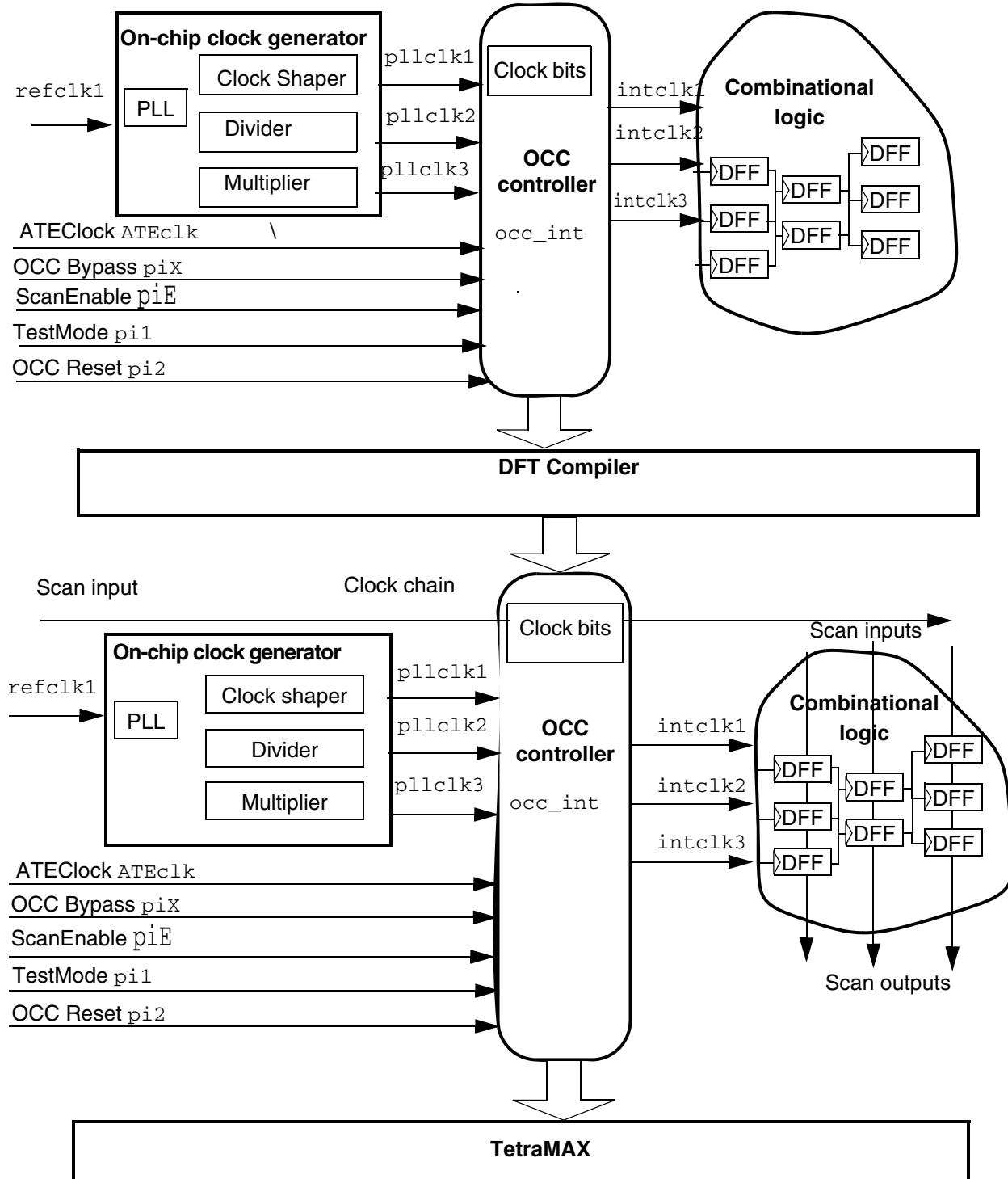


Figure 9-2 User-defined Instantiated Clock Controller and Clock Chain flow



For [Figure 9-1 on page 9-7](#) and [Figure 9-2 on page 9-8](#), note the following:

- The reference clock (`refclk1`) is always free-running. It is used as a test default frequency input to the PLL.
- The PLL clocks (`pllclk1`, `pllclk2`, and `pllclk3`) are free-running clock outputs from the on-chip clock generator; they can be divided, shaped, or multiplied. They are used for the launch and capture of internal scanable elements that become internal clocks.
- The ATE clock (`ATEclk`) shifts the scan chain per tester specifications. Each PLL might have its own ATE clock.

See “[Waveform and Capture Cycle Example](#)” on page [9-30](#) for a waveform diagram that demonstrates the relationship between the various clocks.

- The OCC controller (`occ_int`) serves as an interface between the on-chip clock generator and internal scan chains. This logic typically contains clock multiplexing logic that allows internal clocks to switch from a slow ATE clock during shift to a fast PLL clock during capture.
- Internal clocks (`intclk1`, `intclk2`, and `intclk3`) are outputs of the PLL control logic driving the scan cells. Each internal clock is controlled or enabled by the clock chain and is connected to the sequential elements within the design.
- The OCC Bypass signal (`piX`) allows connection of the ATE clock signal directly to the internal clock signals, thus bypassing the PLL clocks.
- The ScanEnable signal (`piE`) enables switching between the ATE shift clock and output PLL clock signals. ScanEnable must be inactive during every capture procedure. The STIL procedure file (SPF) automatically created from DFT Compiler will force the ScanEnable to be inactive during all capture procedures. This signal is the same signal that drives the flip-flop scan enable. There can be individual ScanEnable signals for each PLL clock signal. The ScanEnable signal is the same signal that is used to select between the scan shift operation and the scan capture operation.
- The TestMode signal (`pi1`) must be active in order for the circuit to work.
- The OCC Reset (`pi2`) is used once during test setup and initialization.
- The clock chain is a scan chain segment of one or more scan cells. This chain allows for a per-pattern clock selection mechanism by ATPG. Clock selection values are loaded into the clock chain as part of the regular scan load process.

Note the following:

- For Scan ATPG, the clock chain can be a dedicated scan chain or segment from any existing internal scan chain.

- For adaptive scan, the clock chain is a dedicated internal chain that is created in addition to scan compression architecture requirements. This makes OCC support architecture requirements independent of scan compression goals.
- If you are using any kind of hierarchical flow, the PLL must be in a block that has not been DFT-inserted before the final integration step. In other words, it must never be replaced by a test model.

DFT Compiler inserts the on-chip clock controller and clock chain and gates each clock controller output to a corresponding internal clock.

In [Figure 9-2](#), you first need to instantiate any user-defined block used as a clock controller or for clock chains in the design. When you do this,

- Ensure that the global and clock signals of the user-defined block are connected to the rest of the design.
- Confirm that connections exist for the internal and external clocks, the reset and mode signals of a clock controller, and the clock signals to a clock chain. This enables the block to be controlled from the top level in the pre-DFT netlist.

Note:

In this design, the clock chains that provide access to clock control bits are embedded within the clock controller, but they could exist as a separate block also.

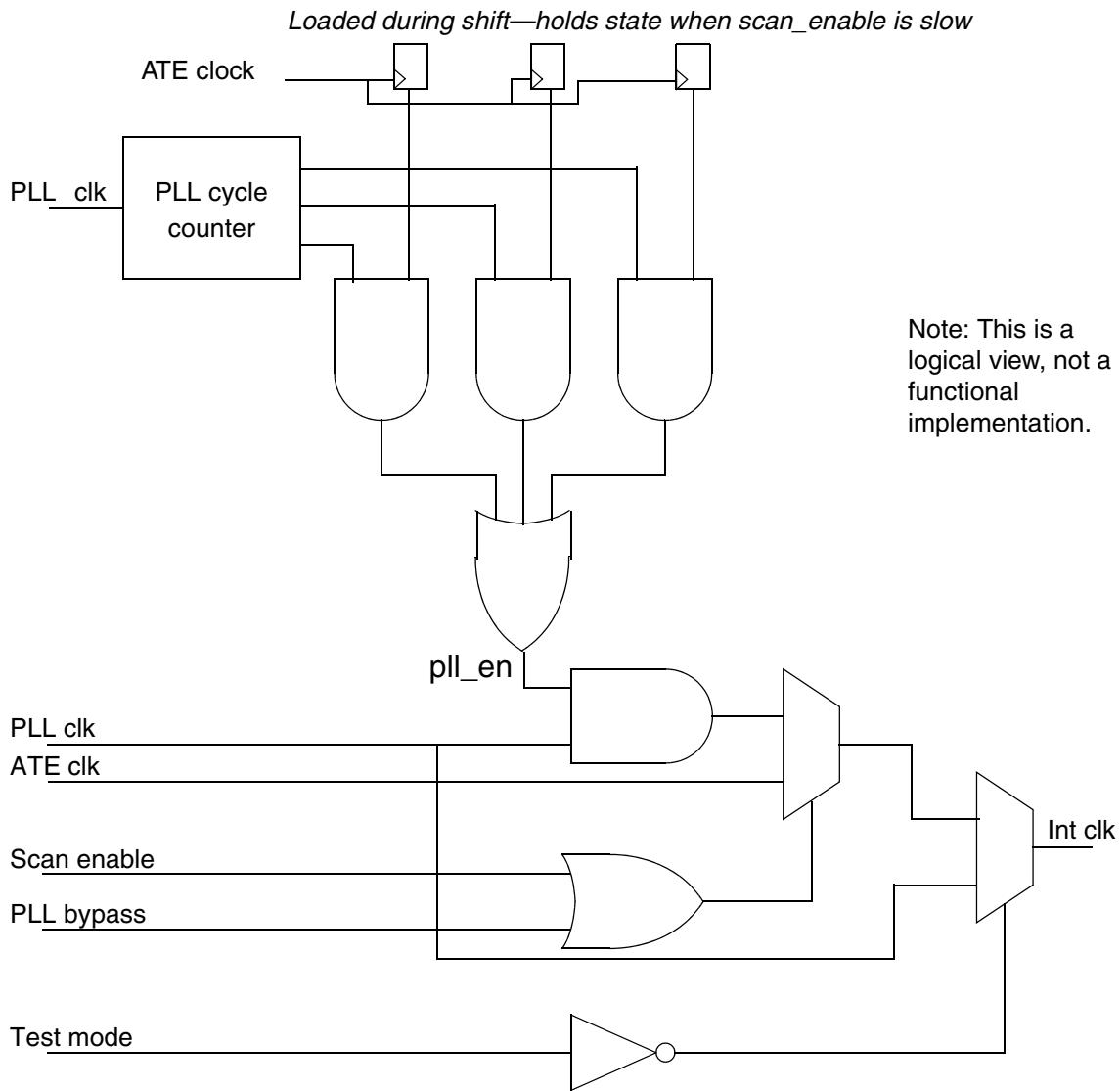
You can insert and validate the clock controller cells at the RTL and choose a preferred synthesis flow and then get to DFT MAX with a gate-level netlist.

## **Logical Representation of an OCC Controller**

In [Figure 9-3](#), scan cells from the clock chain provide control bits that enable or disable internal clock pulsing for each capture cycle. Cycle counter values are used to control the number of PLL clocks that get multiplexed through the block during capture cycles. They are also used to qualify clock chain values on a cycle-by-cycle basis.

For designs containing multiple OCC generators, DFT Compiler adds one or more clock control bits to the clock chain for each additional on-chip clock generator.

Figure 9-3 Logical Representation of a OCC Controller



---

## Basic Processes

The following basic processes are associated with OCC support:

- [Setting Up Your Environment](#)
  - [Enabling On-Chip Clocking Support](#)
- 

## Setting Up Your Environment

To prepare DFT Compiler for OCC support, you need to set up your environment as shown in the following examples:

```
dc_shell> set synthetic_library \
           [concat dw_foundation.sldb dft_lbist.sldb]

dc_shell> set link_library \
           [concat $link_library $synthetic_library \
           $target_library]
```

---

## Enabling On-Chip Clocking Support

To enable OCC support, use the `set_dft_configuration` command and the `-clock_controller` option, as shown in the following example:

```
dc_shell> set_dft_configuration \
           -clock_controller enable
```

---

## OCC Supported Flows

This section covers the on-chip clocking control flows in DFT Compiler.

- [OCC and Clock Chain Synthesis Insertion Flow](#)
  - [User-Defined Instantiated Clock Controller and Clock Chain Insertion Flow](#)
- 

## OCC and Clock Chain Synthesis Insertion Flow

If you are running a design that contains an OCC generator, such as PLL, but not an OCC Controller and clock chain, DFT Compiler inserts the OCC and clock chain as Synopsys clock controller IP. Note that the Synopsys clock controller IP supports only one ATE clock. This section describes the flow associated with this type of implementation.

The PLL clock is expected to already be connected in the design being run through this flow. DFT Compiler will disconnect this PLL clock at the hookup location and insert the newly synthesized clock controller at this location.

This section has the following subsections:

- [Defining Clocks](#)
- [Defining Global Signals](#)
- [Specifying Scan Configuration](#)
- [Configuring the OCC Controller](#)
- [Sample Script](#)

## Defining Clocks

You need to define the reference, PLL, and ATE clocks by using the `set_dft_signal` command. Note that this command does not require you to specify the primary inputs.

This section has the following subsections:

- [Reference Clock](#)
- [ATE Clock](#)
- [PLL-Generated Clocks](#)

**Reference Clock.** The following example shows how to define a PLL reference clock that has the same period as the `test_default_period` variable (assumed to be 100 ns).

```
dc_shell> set_dft_signal -view existing \
                  -type MasterClock -port my_clock \
                  -timing [list 45 55]

dc_shell> set_dft_signal -view existing \
                  -type refclock -port my_clock \
                  -period 100 -timing [list 45 55]
```

Note that the need to define the reference clock with type MasterClock and refclock is required only when the reference clock has the same period as the `test_default_period` variable. Otherwise, it is not needed and not accepted.

To define a reference clock by using a period other than the `test_default_period`, use the following command:

```
dc_shell> set_dft_signal -view existing \
                  -type refclock -port my_clock \
```

```
-period 10 -timing [list 3 8]
```

**Note:** Do not define the signal as any signal type other than refclock.

Also note the following caveats when defining a reference clock associated with the test\_default period:

- If the reference clock period is an integer divisor of the test\_default\_period, then patterns can be written in a variety of formats, including STIL, STIL99, and WGL.
- If the reference clock is not an integer divisor to the test\_default\_period, the only format that can be written in a completely correct way is STIL. Other formats, including STIL99, cannot include the reference clock pulses, and a warning is printed, indicating that these pulses must be added back to the patterns manually.
- Do not define a reference clock period or timings with resolution finer than 1 picosecond. TetraMAX cannot work with finer timing resolutions.

**ATE Clock.** The following examples show how to define the signal behavior of the ATE-provided clock required for shifting scan elements:

```
dc_shell> set_dft_signal -view existing \
                  -type ScanClock -port ATclk \
                  -timing [list 45 55]

dc_shell> set_dft_signal -view existing \
                  -type Oscillator -port ATclk
```

If you want to identify a precise hookup pin where the ATE clock signal should be connected, use -hookup\_pin in the set\_dft\_signal command and specify the hierarchical name of the pin. The set\_dft\_signal command should be issued separately with -view spec, in addition to the two commands with -view existing, because this defines how the signal is connected.

The following example shows how to specify the hookup pin.

```
dc_shell> set_dft_signal -view spec -type ScanClock \
                  -port ATclk -hookup_pin U2005_ate_clk_2/z
```

**Note:**

ATclk and refclk both need to be defined as -type ScanClock and -type Oscillator.

**PLL-Generated Clocks.** For DFT Compiler to correctly insert the OCC, you must define the PLL-generated clocks as well as the point at which they are generated. The following examples show how to define a set of launch and capture clocks for internal scannable elements controlled by the OCC controller:

```

dc_shell> set_dft_signal -view existing \
                     -type Oscillator -hookup_pin pll/pllclk1

dc_shell> set_dft_signal -view existing \
                     -type Oscillator -hookup_pin pll/pllclk2

dc_shell> set_dft_signal -view existing \
                     -type Oscillator -hookup_pin pll/pllclk3

```

## Defining Global Signals

You should identify the following global signals in your design.

- [Global Clock Controller Signals](#)
- [Test Mode Signal](#)

**Global Clock Controller Signals.** You must identify the top-level interface to control the clock controller. This includes the top-level signals, such as the OCC Bypass, OCC Reset and ScanEnable signals.

The following examples show how to define a set of global controller signals for the example design:

```

dc_shell> set_dft_signal \
           -type pll_reset \
           -port pi2 \
           -view spec

dc_shell> set_dft_signal \
           -type pll_bypass \
           -port pix \
           -view spec

dc_shell> set_dft_signal \
           -type ScanEnable \
           -port piE \
           -view spec

```

**Test Mode Signal.** You must specify the top-level TestMode signal that is connected to the clock controller logic. To specify this signal, use the command syntax shown in the following example:

```

dc_shell> set_dft_signal \
           -type TestMode \
           -port pi1 \
           -view spec

```

Note:

You can specify the hookup pin for any OCC global signal by using `-hookup_pin` in the `set_dft_signal` command.

## Configuring the OCC Controller

To configure the OCC controller, use the `set_dft_clock_controller` command. Note the following syntax and descriptions:

```
int set_dft_clock_controller
  [-cell_name cell_name]
  [-design design_name]
  [-pllclocks ordered_list]
  [-ateclocks clock_name]
  [-chain_count integer ]
  [-cycles_per_clock integer]
```

Option	Description
<code>-cell_name <i>cell_name</i></code>	Specifies the hierarchical name of the clock controller cell.
<code>-design <i>design_name</i></code>	Specifies the OCC controller design name. You must specify <code>smps_clk_mux</code> .
<code>-pllclocks <i>ordered_list</i></code>	Specifies the ordered list of hierarchical PLL clocks you want to control.
<code>-ateclocks <i>clock_name</i></code>	Specifies the ATE clock (port) you want to connect to the OCC controller. Note: You cannot specify multiple clocks per controller.
<code>-chain_count <i>integer</i></code>	Specifies the number of clock chains. The default number of clock chains is one.
<code>-cycles_per_clock <i>integer</i></code>	Specifies the maximum number of capture cycles per clock. The default number of clock cycles is two. You should specify a minimum of two. Capture cycles are cycles during capture when capture clocks are pulsed. Typically, for at-speed transition testing, there are two capture cycles: one is used for launching a transition and the other for capturing the effect of that transition.

The following example shows a typical usage of the `set_dft_clock_controller` command:

```
dc_shell> set_dft_clock_controller \
```

```

-cell_name occ_int \
-design_name snps_clk_mux \
-pllclocks { pll/pllclk1 pll/pllclk2 \
pll/pllclk3 } \
-ateclocks { ATEclk } \
-cycles_per_clock 2 -chain_count 1

```

## Specifying Scan Configuration

Specify the `set_scan_configuration` command to define scan ports, scan chains, and the global scan configuration.

To specify scan constraints in your design, use the following command:

```
set_scan_configuration -chain_count <#chains>...
```

## Sample Script

[Example 9-1](#) shows DFT Compiler performing prescan DRC, scan chain stitching, and post-scan DRC. The STIL protocol file generated at the end of the DFT insertion process contains PLL clock details suitable for TetraMAX.

### *Example 9-1 OCC Controller and Clock Chain Synthesis and Insertion*

```

read_verilog mydesign.v
current_design mydesign
link

# Define the PLL reference clock
# top level free running clock

set_dft_signal -view existing -type refclock \
-port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing -type MasterClock \
-port refclk1 -timing [list 45 55]

# Define the ATE clock
# the ATE-provided clock for shift of scan elements

set_dft_signal -view existing -type ScanClock \
-port ATEclk -timing [list 45 55]

set_dft_signal -view existing -type Oscillator \
-port ATEclk

# Define the PLL generated clocks --
# these are the launch/capture clocks for internal scannable
# elements and are controlled by occ controller

set_dft_signal -view existing -type Oscillator \
-hookup_pin pll/pllclk1

set_dft_signal -view existing -type Oscillator \

```

```

-hookup_pin pll/pllclk2

set_dft_signal -view existing -type Oscillator \
    -hookup_pin pll/pllclk3

# Enable PLL capability
set_dft_configuration -clock_controller enable

# The following command specifies the OCC controller
# design to be instantiated. The DFT Compiler synthesized
# clock controller is named snps_clk_mux

set_dft_clock_controller -cell_name snps_pll_controller \
    -design snps_clk_mux -pllclocks { pll/pllclk1 \
    pll/pllclk2 pll/pllclk3 } -ateclocks { ATEclk } \
    -cycles_per_clock 2 -chain_count 1

set_scan_configuration -chain_count 30 -test_mode all
create_test_protocol -capture_procedure multi_clock
dft_drc
report_dft_clock_controller -view spec
preview_dft -show all
insert_dft
dft_drc

# Run DRC with external clocks enabled during capture
# (PLL bypassed)

set_dft_drc_configuration -pll_bypass enable
dft_drc

change_names -rules verilog -h
write -f verilog -h -o top.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -test_mode Internal_scan -o scan_pll.stil

```

**Note:**

This DFT Compiler implementation is intended only as an example and might not satisfy all requirements.

## User-Defined Instantiated Clock Controller and Clock Chain Insertion Flow

This section has the following subsections:

- [Defining Clocks](#)
- [Defining Global Signals](#)
- [Specifying Clock Chains](#)

- Specifying Scan Configuration
- Sample Script

## Defining Clocks

Use the `set_dft_signal` command to define the following clock signals for [Figure 9-2](#):

- [PLL Clocks](#)
- [Reference Clocks](#)
- [ATE Clocks](#)
- [Clock Controller Outputs and Control-Per-Pattern Information](#)

**PLL Clocks.** PLL clocks are the output of the PLL. This output is a free-running source that also runs at a constant frequency, which might not be the same as the reference clock's. This information is forwarded to TetraMAX through the protocol file to allow the verification of the clock controller logic.

The following commands show how to define a PLL clock for the example design:

```
dc_shell> set_dft_signal
          -type Oscillator \
          -hookup_pin PLL/pllclk1 \
          -view existing

dc_shell> set_dft_signal
          -type Oscillator \
          -hookup_pin PLL/pllclk2 \
          -view existing

dc_shell> set_dft_signal
          -type Oscillator \
          -hookup_pin PLL/pllclk3 \
          -view existing
```

**Reference Clocks.** A reference clock definition is used primarily as an informational device. It does not constrain an insertion. Currently, you need to define reference clocks in order to pass TetraMAX DRC.

The following example shows how to define a PLL reference clock that has the same period as the `test_default_period` variable (assumed to be 100 ns).

```
dc_shell> set_dft_signal -view existing \
          -type MasterClock -port my_clock \
          -timing [list 45 55]
```

```
dc_shell> set_dft_signal -view existing \
           -type refclock -port my_clock \
           -period 100 -timing [list 45 55]
```

Note that the need to define the reference clock with type MasterClock and refclock is required only when the reference clock has the same period as the test\_default\_period variable. Otherwise, it is not needed and not accepted.

To define a reference clock by using a period other than the test\_default\_period, use the following command:

```
dc_shell> set_dft_signal -view existing \
           -type refclock -port my_clock \
           -period 10 -timing [list 3 8]
```

**Note:** Do not define the signal as any signal type other than refclock.

Also note the following caveats when defining a reference clock associated with the test\_default\_period:

- If the reference clock period is an integer divisor of the test\_default\_period, then patterns can be written in a variety of formats, including STIL, STIL99, and WGL.
- If the reference clock is not an integer divisor of the test\_default\_period, the only format that can be written in a completely correct way is STIL. Other formats (including STIL99) cannot include the reference clock pulses. A warning is printed, indicating that these pulses must be added back to the patterns manually.
- Do not define a reference clock period or timings with resolution finer than 1 picosecond. TetraMAX cannot work with finer timing resolutions.

**ATE Clocks.** An ATE clock signal can be pulsed several times before and after scan shift (scan enable inactive) to synchronize the clock controller logic in the capture phase and back into the shift phase.

The following commands show how to define ATE clocks for the example design:

```
dc_shell> set_dft_signal \
           -type ScanClock \
           -port ATEclk \
           -timing [list 45 55] \
           -view existing

dc_shell> set_dft_signal \
           -type Oscillator \
           -port ATEclk \
           -view existing
```

**Clock Controller Outputs and Control-Per-Pattern Information.** You must specify the correlation between the internal clock signals driven from the clock controller outputs, the signals driven from the clock generator (PLL) outputs, and the signals gating the internal clock signals (control bits). This information indicates which internal clock signal is active in a particular pattern. The correspondence between the controlled internal clock signals and clock control bits should be identified in the protocol file for TetraMAX to generate patterns.

You must specify

- All user-defined clock controller outputs referencing the internal clocks
- A corresponding set of control bits, ATE clock, and clock generator output (PLL) for each clock controller output

See the following examples:

```
dc_shell> set_dft_signal -type Oscillator \
    -hookup_pin occ_int/intclk1 \
    -ate_clock ATEclk \
    -pll_clock PLL/pllclk1 \
    -ctrl_bits [list 0 occ_int/FF_1/Q 1 \
    1 occ_int/FF_2/Q 1] \
    -view existing
```

Note:

The `-ctrl_bits` option is used to provide a list of triplets that specify the sequence of bits needed to enable the propagation of the clock generator outputs. The first element of each triplet is the cycle number (integer) indicating when the clock signal will be propagated; the second element is the pin name (a valid design hierarchical pin name) of the control bit; and the third element is the active value (0 or 1) of the control bit. For more information about this option, see the `set_dft_signal` man page.

Note:

The `-view existing` option is used because connections already exist between the referenced port and the clock controller.

## Defining Global Signals

You should identify the following global signals in your design:

- [Global Clock Controller Signals](#)
- [Test Mode Signal](#)

**Global Clock Controller Signals.** You must identify the top-level interface to control the clock controller. This includes the top-level signals, such as OCC Bypass, OCC Reset, and ScanEnable signals.

The following examples show how to define a set of global controller signals for the example design:

```
dc_shell> set_dft_signal \
           -type pll_reset \
           -port pi2 \
           -view existing

dc_shell> set_dft_signal \
           -type pll_bypass \
           -port pix \
           -view existing

dc_shell> set_dft_signal \
           -type ScanEnable \
           -port piE \
           -view existing
```

**Test Mode Signal.** You must specify the top-level TestMode signal that is connected to the clock controller logic. To specify this signal, use the command syntax shown in the following example:

```
dc_shell> set_dft_signal \
           -type TestMode \
           -port pil \
           -view existing
```

## Specifying Clock Chains

Specify clock chains by using the `set_scan_group` command. This ensures that the sequential cells are treated as a group and are logically ordered.

```
dc_shell> set_scan_group clk_chain \
           -include_elements \
           [list occ_int/FF_1 occ_int/FF_2] \
           -access [ list ScanDataIn occ_int/si \
           ScanDataOut occ_int/so \
           ScanEnable occ_int/se ] \
           -serial_routed true
```

If you are using adaptive scan, a clock chain must be assigned to a separate scan chain to prevent it from being included between decompressors and compressors logic. The process of combining adaptive scan with clock controllers results in a multimode architecture; therefore, both modes must be specified.

See the following examples:

```
dc_shell> set_scan_path clk_chain \
           -include clk_chain \
           -complete true \
```

```

        -scan_data_in si -scan_data_out so \
-test_mode Internal_scan

dc_shell> set_scan_path clk_chain \
           -include clk_chain -complete true \
           -scan_data_in si -scan_data_out so \
           -test_mode ScanCompression_mode

```

## Specifying Scan Configuration

Specify the `set_scan_configuration` command to define scan ports, scan chains, and the global scan configuration.

To specify scan constraints in your design, use the following command:

```
set_scan_configuration -chain_count <#chains>...
```

## Sample Script

When you run a design that contains an OCC controller and clock chains, a STIL protocol file is generated, as shown in [Example 9-2](#).

### *Example 9-2 Example Flow for Designs With Existing Clock Controller and Clock Chains*

```

read_verilog mydesign.v
current_design mydesign
link

# Define the PLL reference clock
# top level free running clock

set_dft_signal -view existing -type refclock \
    -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing -type MasterClock \
    -port refclk1 -timing [list 45 55]

# Define the ATE clock
# the ATE-provided clock for shift of scan elements

set_dft_signal -view existing -type ScanClock \
    -port ATEclk -timing [list 45 55]

set_dft_signal -view existing -type Oscillator \
    -port ATEclk

# Define the PLL generated clocks
set_dft_signal -view existing -type Oscillator \
    -hookup_pin pll/pllclk1
set_dft_signal -view existing -type Oscillator \
    -hookup_pin pll/pllclk2
set_dft_signal -view existing -type Oscillator \
    -hookup_pin pll/pllclk3

```

```

# Enable PLL capability
set_dft_configuration -clock_controller enable

# Specify clock controller output and control-per-pattern information
set_dft_signal -type Oscillator -hookup_pin occ_int/intclk1 \
-ate_clock ATEclk -pll_clock PLL/pllclk1 \
-ctrl_bits [list 0 occ_int/FF_1/Q 1 1 occ_int/FF_2/Q 1] \
-view existing

# Define the existing clock chain segments
set_scan_group clk_chain \
    -include_elements [list occ_int/FF_1 \
    occ_int/FF_2] \
        -access [ list ScanDataIn occ_int/si \
    ScanDataOut occ_int/so ScanEnable occ_int/se ] \
    -serial_routed true

set_dft_signal -type ScanDataIn -port si -view spec
set_dft_signal -type ScanDataOut -port so -view spec
set_scan_path clk_chain_path -include clk_chain \
    -complete true -scan_data_in si \
    -scan_data_out so -test_mode all_dft

# Specify global controller signals
set_dft_signal -type pll_reset -port pi2 -view existing
set_dft_signal -type pll_bypass -port piX -view existing
set_dft_signal -type ScanEnable -port piE -view existing

# Define the TestMode signals
set_dft_signal -type TestMode -port pil -view existing

# Registers inside the OCC controller must be non-scan or else the
# internal clock will not be controlled correctly. However, the
# clock chain must be scanned. Use set_scan_element false on the
# former but not on the latter.
set_scan_element false occ_int/clock_controller

create_test_protocol -capture_procedure multi_clock
dft_drc
report_dft_clock_controller -view existing
preview_dft -show all
insert_dft
dft_drc

# Run DRC with external clocks enabled during capture
# (PLL bypassed)

set_dft_drc_configuration -pll_bypass enable
dft_drc

change_names -rules verilog -h
write -f verilog -h -o top.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -test_mode Internal_scan -o scan_pll.stil

```

---

## Reporting Clock Controller Information

Use the `report_dft_clock_controller` command to generate reports.

---

## OCC and Clock Chain Synthesis Insertion Flow

Use the `report_dft_clock_controller -view spec` command to output a report. This report will display the options that you set for the `set_dft_clock_controller` command.

[Example 9-3](#) shows how the test protocol will read for the OCC and clock chain synthesis insertion flow.

*Example 9-3 Sample Report for report\_dft\_clock\_controller -view spec*

```
*****
Report : Clock controller
Design : des_chip
Version: Z-2007.03
Date   : Fri Jan 12 05:42:06 2007
*****  
  
=====
TEST MODE: all_dft
VIEW     : Specification
=====  
Cell name:          pll_controller_0
Design:            snps_clk_mux
Chain count:       1
Cycle count:       2
PLL clock:         u_pll/clkgenx2 u_pll/clkgenx3
ATE clock:         ateclk
=====
```

---

## User-Defined Clock Controller and Clock Chain Insertion Flow

For user-defined clock controllers and clock chains, after you define the internal clock signals and the corresponding control-per-pattern information, use the `report_dft_clock_controller -view existing` command to report what you have specified. In [Example 9-4](#), the report shows the relationship among

- The clock controller output
- The ATE clock
- The clock control bits

*Example 9-4 Sample report\_dft\_clock\_controller Output*

```
*****
Report : Clock controller
Design  : des_chip
Version: Z-2007.03
Date   : Fri Jan 12 05:18:55 2007
*****  
=====
TEST MODE: all_dft_user
VIEW      : Existing DFT
=====  
Clock controller: ctrl_0
=====  
    Number of bits per clock: 4
    Controlled clock output pin: dutm/clk
=====  
    Clock generator signal: u_pll/clkgennx2
    ATE clock signal: ateclk
    Control pins:  
        cycle 0  dutc/FF_0_reg/Q  1
        cycle 1  dutc/FF_1_reg/Q  1
        cycle 2  dutc/FF_2_reg/Q  1
        cycle 3  dutc/FF_3_reg/Q  1
=====  
=====
```

---

## DRC Support

D-rules (Category D – DRC Rules) support PLL-related design rule checks. The checked rule and message text correspond by number to TetraMAX PLL-related C-rules (Category C – Clock Rules). The rules are as follows:

- D28 - Invalid PLL source for internal clock
- D29 - Undefined PLL source for internal clock
- D30 - Scan PLL conditioning affected by nonscan cells
- D31 - Scan PLL conditioning not stable during capture
- D34 - Unsensitized path between PLL source and internal clock
- D35 - Multiple sensitizations between PLL source and internal clock
- D36 - Mistimed sensitizations between PLL source and internal clock
- D37 - Cannot satisfy all internal clocks off for all cycles
- D38 - Bad off conditioning between PLL source and internal clock.

---

## Enabling the OCC Controller Bypass Configuration

Use the `set_dft_drc_configuration` and `write_test_protocol` commands to enable the OCC controller bypass configuration for design rule checking. The `set_dft_drc_configuration` command enables post-scan insertion DRC with constraints that put the OCC clock controller in bypass configuration.

The syntax is as follows:

```
set_dft_drc_configuration -pll_bypass enable | disable
```

The default setting is `disable`.

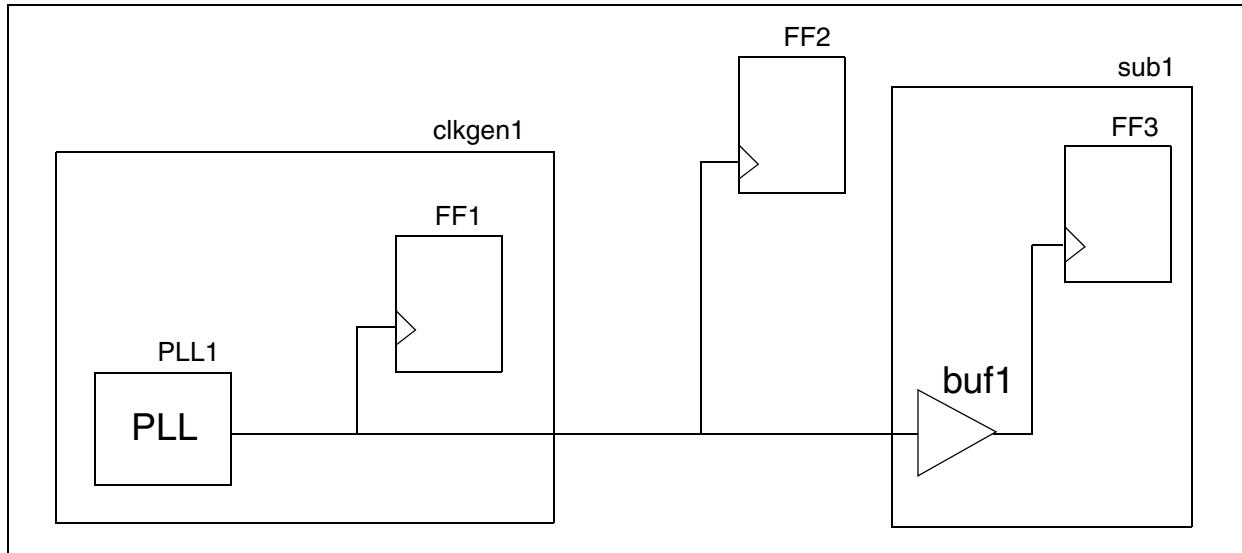
The test protocol written by the `write_test_protocol` command contains information for PLL bypass as well as for PLL enabled. In TetraMAX, use `run_drc -patternexec` to select the operating mode to use.

---

## Example Configurations on a Design

This section shows the results of using various configurations of the `set_dft_clock_controller` command on the example design shown in [Figure 9-4](#).

*Figure 9-4 Example Design*



The following example configurations are applied to this design:

- [Example 1](#) — Controller inserted at the output of PLL, within clkgen1 block
- [Example 2](#) — Controller inserted at the output of clkgen1 block
- [Example 3](#) — Controller inserted at the output of the buffer

---

## Example 1

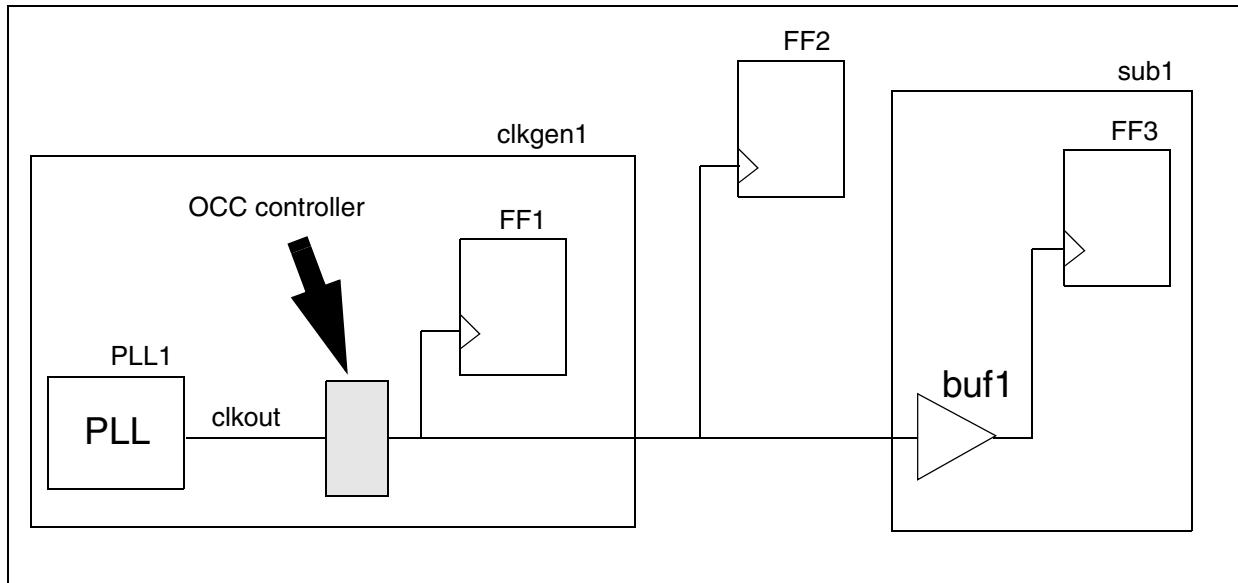
The first example, shown in [Figure 9-5](#), uses the following configuration:

```
dc_shell> set_dft_clock_controller \
           -pllclocks {clkgen1/PLL1/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of PLL, within the clkgen1 block.
- The clocks of all flip-flops are controllable.

*Figure 9-5 Controller Inserted at Output of PLL, Within clkgen1 Block*



---

## Example 2

The second example, shown in [Figure 9-6](#), uses the following configuration:

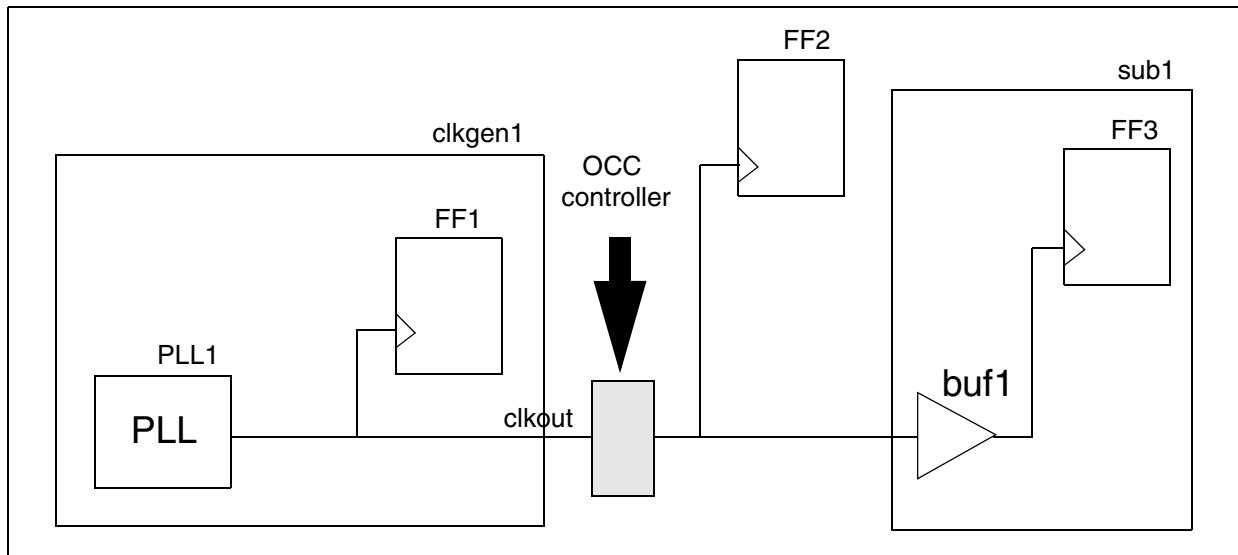
```
dc_shell> set_dft_clock_controller \
```

```
-pllclocks {clkgen1/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of the clkgen1 block.
- The FF1 clock remains uncontrollable.

*Figure 9-6 Controller Inserted at Output of clkgen1 Block*



---

### Example 3

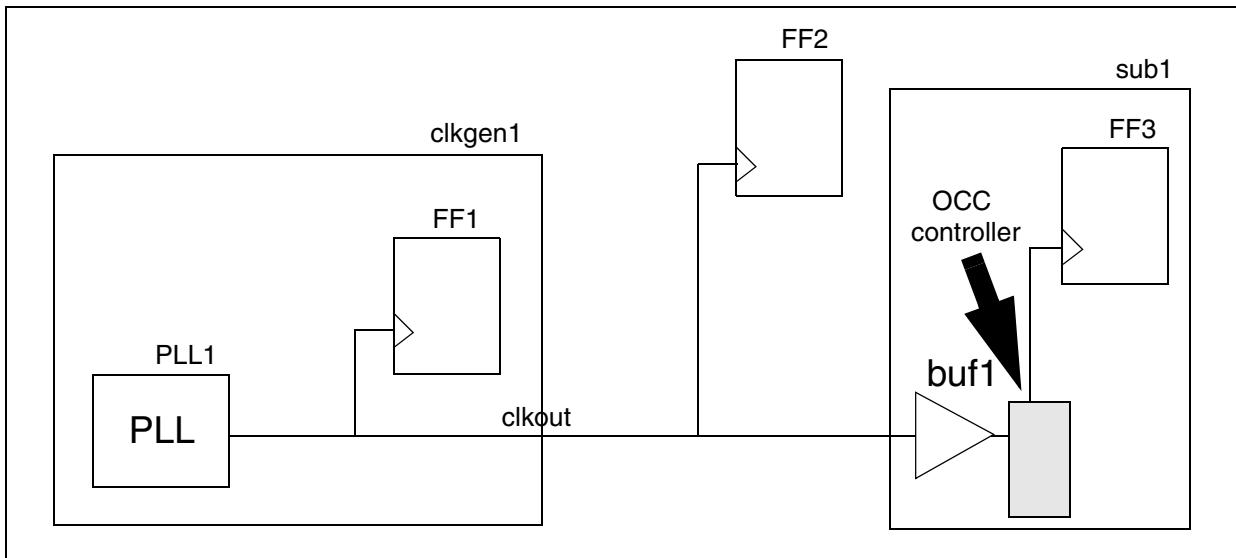
The third example, shown in [Figure 9-7](#), uses the following configuration:

```
dc_shell> set_dft_clock_controller \
           -pllclocks {sub1/buf1/z}
```

In this case, the following occurs:

- The controller is inserted at the output of the buffer.
- The FF1 and FF2 clocks remain uncontrollable.

Figure 9-7 Controller Inserted at Output of the Buffer

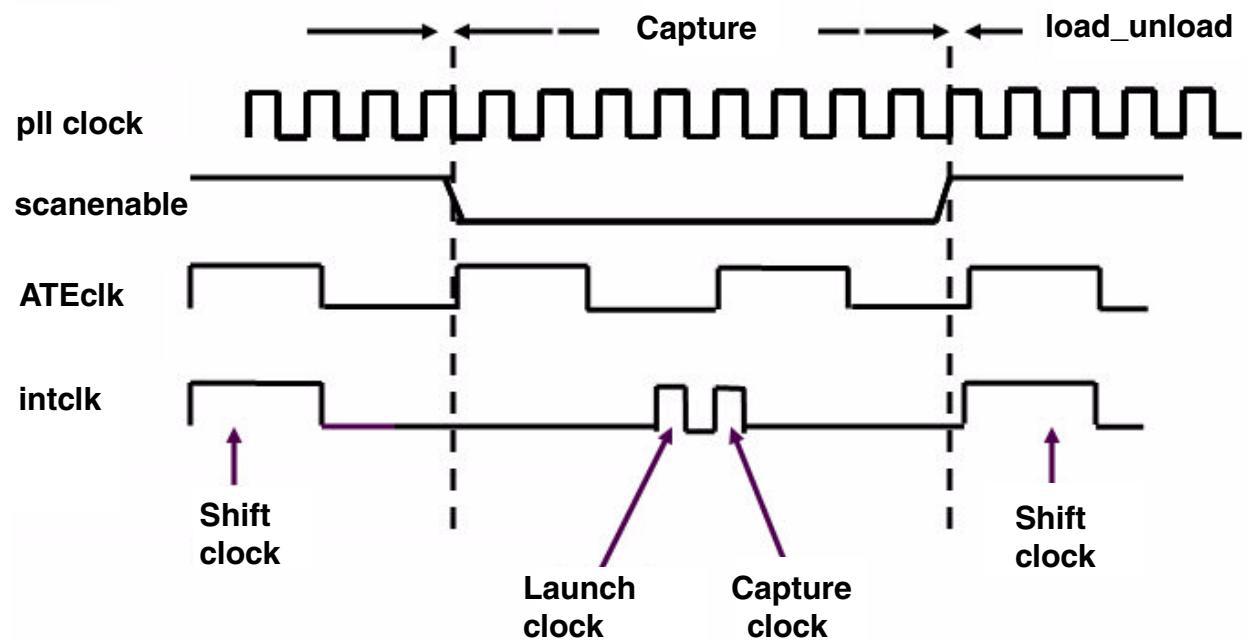


---

## Waveform and Capture Cycle Example

[Figure 9-8](#) shows an example of the relationship between various clocks when the design contains an OCC generator and an OCC controller.

Figure 9-8 Desired Clock Launch Waveform Example



For information about pll clock, ATEclk, and intclk, see “[Definitions](#)” on page 9-4



# 10

## Exporting to Other Tools

---

This chapter describes how to export output from DFT Compiler to other tools, such as TetraMAX ATPG.

This chapter includes the following sections:

- [Verifying DFT Inserted Designs for Functionality](#)
- [Exporting a Design to TetraMAX ATPG](#)
- [SCANDEF-Based Reordering Flow](#)

---

## Verifying DFT Inserted Designs for Functionality

After DFT insertion, the resulting scan-inserted design is verified for functional equivalence with respect to the nonscan design. This is done to ensure that DFT insertion did not introduce any logic errors. Verification is accomplished by using the Synopsys Formality tool.

The following subsections describe the verification process.

- [SVF File Generation](#)
  - [Test Information Passed to the SVF File](#)
  - [Sample Script](#)
  - [Formality Tool Limitations](#)
- 

### SVF File Generation

By default, Design Compiler automatically creates a setup file in your working directory. The automated setup file has the extension .svf (setup verification file) and is named default.svf. This file tracks any design changes that are required for the verification process and assists the Formality tool in compare-point matching and verification.

The automated setup file is stored in binary format.

Use the `set_svf` command to generate a Formality setup information file for efficient compare-point matching in Formality.

The syntax is as follows:

```
set_svf
    [-filename]
    [-append]
    [-off]
```

## Argument Definitions

-filename

Specifies the file into which Formality setup information is recorded. You must specify a file name unless the -off option is specified.

-append

Appends to the specified file. If another Formality setup verification file is already open, then it will be closed before opening the specified file. If -append is not used, then set\_svf overwrites the named file, if it exists.

-off

Stops recording Formality setup information to the currently open file. To resume recording into the same file, you must re-issue the set\_svf command with the -append option.

---

## Test Information Passed to the SVF File

When you run the insert\_dft command, the following DFT specific information is recorded in the SVF file:

- Scan enables are disabled.
- Test modes are disabled wherever they are used (for example, AutoFix/ adaptive scan).
- Constants are passed to the file.
- Core wrapper shift (wrf\_shift) is disabled.
- The TCK, TMS, and TRST ports of BSD Compiler are held at 0 and the TDO port is not verified.

The setup information is reported in the assumptions summary report.

For more information, see the *Formality User Guide*.

---

## Sample Script

[Example 10-1](#) shows you how to use an SVF file for functionality checking in Formality.

### *Example 10-1 Sample Formality Script For Equivalence Checking*

```
#Enable Automatic Setup to Disable Scan/Test
set synopsys_auto_setup true

#Set your SVF file
set_svf ./my_svf_file
```

```

# READ LIBRARIES
foreach file $link_library {read_db $lib}

# Read Reference Design
create_container pre_dft
read_ddc ./outputs/des_unit.pre_dft.ddc
set_top des_unitset_reference_design pre_dft:/WORK/des_unit

# Read Implementation Design
create_container post_dft
read_ddc ./outputs/des_unit.post_dft.ddc
set_top des_unit
set_implementation_design post_dft:/WORK/des_unit
# Match compare points and Verify
match
verify

```

---

## Formality Tool Limitations

The following features are not supported.

- Internal pins support
- DBIST

---

## Exporting a Design to TetraMAX ATPG

This section describes the steps that take a design from DFT Compiler to TetraMAX ATPG, in which you use the automatic test pattern generation (ATPG) capability to generate test vectors. It has the following subsections:

- [Before Exporting Your Design](#)
- [Exporting Your Design to TetraMAX ATPG](#)

---

## Before Exporting Your Design

Before exporting your design to TetraMAX ATPG, be aware of the following:

- You need to make sure all design rule violations have been corrected. Use the `dft_drc` command to detect and correct scan design rule violations.
- Designs must have valid scan chains in order to be recognized by TetraMAX ATPG. Any nonscan sequential cell or capture violation has the potential to lower fault coverage. Use the `report_scan_path -chain all` command to verify that all scan chains are intact before exporting a design.

- All cells on the scan chain must be controllable and observable. This is because controllable-only or observable-only chains identified by DFT Compiler are not written to the STIL protocol file, and TetraMAX ATPG does not recognize them. If you have problems with the scan chain, you must fix them by using DFT Compiler.
- TetraMAX ATPG does not accept designs in which the original source was VHDL and arrays of arrays are used in top-level buses.
- Be aware of dependent slave operation. TetraMAX ATPG reports an S29 warning for circuits and protocols generated by DFT Compiler if the circuit has scan-out lock-up latches inserted by DFT Compiler, and these latches are closed at the end of a cycle. The `dft_drc` command, which runs DFT Compiler design rule checking, similarly reports diverging scan chains for such lock-up latches

Before exporting your design to TetraMAX ATPG, you should also be aware of the differences in which DFT Compiler and TetraMAX ATPG handle your design. These differences are explained in the following sections:

- [Support for DFT Compiler Commands in TetraMAX ATPG](#)
- [Creating Generic Capture Procedures](#)

## **Support for DFT Compiler Commands in TetraMAX ATPG**

To export a DFT Compiler flow to the TetraMAX ATPG flow, TetraMAX ATPG must translate some DFT Compiler commands into TetraMAX ATPG commands. The TetraMAX ATPG flow supports the following DFT Compiler commands:

- `set_dft_signal`
- `read_test_protocol`

The following DFT Compiler commands have no impact on the STIL protocol and are therefore ignored in the flow from DFT Compiler to TetraMAX:

- `set_test_assume`

## **Creating Generic Capture Procedures**

DFT Compiler allows you to write out a protocol file with generic capture procedures to be used in TetraMAX ATPG.

The generic capture procedures consist of the following procedures:

- `multiclock_capture()`
- `allclock_capture()`
- `allclock_launch()`

- `allclock_launch_capture()`

**Advantages of Generic Capture Procedures.** This is the preferred format for the protocol file to be given to TetraMAX ATPG due to the following advantages:

- The single cycle capture procedure is efficient.
- It matches the event ordering (force PI, measure PO, pulse clock) in TetraMAX without any manual modifications.
- Stuck-at and at-speed ATPG can now use a single common protocol file
- The `stuck-at _default_WFT_ WaveformTable` is used as a template for modifying the timing of the at-speed WaveformTables.

**Writing a Protocol File With Generic Capture Procedures.** The criteria for writing out a protocol file with generic capture procedures are as follows:

- To write out protocol files, use the `create_test_protocol` command.

```
create_test_protocol -capture_procedure
    [single_clock | multi_clock]
```

Choose `multi_clock` to create a protocol file that uses one generic capture procedure for all capture clocks. The default value is `multi_clock`.

- You must use preclock strobe timing to write out a single vector generic capture procedure. Otherwise, the capture procedures will have three-vectors.

The following variables are set to their default values to define the timing in the protocol file.

- `$test_default_delay 0`
- `$test_default_bidir_delay 0`
- `$test_default_strobe 40`
- `$test_default_period 100`

To use preclock strobe timing, the value of `$test_default_strobe` must be set to be before the time of the leading edge of any of the test clocks that you have defined.

**WaveformTables.** The STIL protocol file, written out, using the generic capture procedures, has several different WaveformTables:

- `_default_WFT_`
- `_multiclock_capture_WFT_`

- `_allclock_capture_WFT_`
- `_allclock_launch_WFT_`
- `_allclock_launch_capture_WFT_`

The timings of these different WaveformTables are identical when they are written out by DFT Compiler.

The WaveformTables are suitable for the following procedures:

- The `test_setup` macro and `load_unload` procedures, which use `_default_WFT_`
- The capture procedures that neither launch nor capture transition fault effects and use `_multiclock_capture_WFT_`. Generic capture procedures, using the internal clocks also use `_multiclock_capture_WFT_` because the PLL pulse trains are generated internally and independently of the external timing.

Observe the following criteria when using the protocol file for at-speed testing:

- The timings in the `_allclock_` WaveformTables should be changed to get at-speed transition fault testing on the external clocks. Ensure that you do not change the period or the timings of the reference clocks, or else the PLLs may lose lock. You should change only the rise and fall times of the external clocks.
- Each two-clock transition fault test consists of a launch cycle using `_allclock_launch_WFT_` followed by a capture cycle using `_allclock_capture_WFT_`. The active clock edges of these two cycles should be close to each other. Make sure that the clock leading-edge comes after the `all_outputs` strobe time, and adjust the time for all values (L, H, T and X) in the `_allclock_capture_WFT_` if necessary.
- The `_allclock_launch_capture_WFT` is used only when launch and capture are caused by opposite edges of the same clock. Note the timing is from the leading edge of the clock to the same clock's trailing edge. However, this timing only occurs in full sequential ATPG and can be ignored in most cases.

**Writing a Protocol File With a Multiclock Capture Procedure.** You can control multiple clock capture by specifying a single generic capture procedure, called `multiclock_capture`, in an SPF file. This procedure, which enables you to map all capture behaviors irrespective of the number of clocks present, is the default procedure.

In addition to supporting capture operations that contain multiple clocks, this procedure also eliminates the need to manually define a full set of clock-specific capture procedures. The `multiclock_capture` is used for stuck-at ATPG but can be used for transition delay and path delay ATPG if the `allclock` procedures are not present in the protocol file.

[Example 10-2](#) shows a `multiclock_capture` procedure.

*Example 10-2 Example of a multiclock Capture Procedure*

```
Procedures {
    "multiclock_capture" {
        W "_multiclock_capture_WFT_";
        C {
            "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
            "all_outputs" = \r165 X;
            "all_bidirectionals" = ZZZ;
        }
        F {
            "i_scan_block_sel[0]" = 1;
            "i_scan_block_sel[1]" = 1;
            "i_scan_compress_mode" = 0;
            "i_scan_testmode" = 1;
        }
        V {
            "_po" = \r168 #;
            "_pi" = \r179 #;
        }
    }
}
```

As is the case with all capture procedures, the single-vector form of `multiclock_capture` requires the timing in the WaveformTable to follow the TetraMAX event order for captures (preclock strobe timing). This means that all input transitions must occur first, all output measures must occur next, and all clock pulses must be defined as the last event.

[Example 10-3](#) shows a `multiclock_capture` WaveformTable.

**Example 10-3 Example of multiclock\_capture WaveformTable**

```
WaveformTable "_multiclock_capture_WFT_" {
    Period '100ns';
    Waveforms {
        "all_inputs" {
            0 {
                '0ns' D;
            }
        }
        "all_inputs" {
            1 {
                '0ns' U;
            }
        }
        "all_inputs" {
            Z {
                '0ns' Z;
            }
        }
        "all_bidirectionals" {
            T {
                '0ns' Z;
                '40ns' T;
            }
        }
        "all_bidirectionals" {
            L {
                '0ns' Z;
                '40ns' L;
            }
        }
        "all_outputs" {
            X {
                '0ns' X;
                '40ns' X;
            }
        }
        "all_outputs" {
            H {
                '0ns' X;
                '40ns' H;
            }
        }
        "all_outputs" {
            T {
                '0ns' X;
                '40ns' T;
            }
        }
        "all_outputs" {
            L {
                '0ns' X;
                '40ns' L;
            }
        }
        "i_scan_clk_sclk2_in" {
            P {
                '0ns' D;
            }
        }
    }
}
```

```

        '45ns' U;
        '55ns' D;
    }
}
"i_scan_clk_sclkd" {
    P {
        '0ns' D;
        '45ns' U;
        '55ns' D;
    }
}
"i_resetin" {
    P {
        '0ns' U;
        '45ns' D;
        '55ns' U;
    }
}
}
}

```

**Writing a Protocol File With a Single Clock Capture Procedure.** You can write out a protocol file that uses a single clock capture procedure for all clocks (also known as the legacy three-vector capture procedure) by specifying the `-capture_procedure single_clock` option-argument of the `create_test_protocol` command.

[Example 10-4](#) shows a `single_clock` procedure.

*Example 10-4 Example of a single\_clock Capture Procedure*

```
Procedures {
    "capture_clk_st"{
        W "_default_WFT_";
        C {"all_inputs" = 0\r59 N 0;
            "all_outputs" = \r46 X;};
        F {"test_mode" = 0;};
        "forcePI": V {"_pi" = \r61 #;};
        "measurePO": V {"_po" = \r46 #;};
        "pulse": V {"clk_st" = P;};
    }
    "capture_pclk"{
        W "_default_WFT_";
        C {"all_inputs" = 0\r59 N 0;
            "all_outputs" = \r46 X;};
        F {"test_mode" = 0;};
        "forcePI": V {"_pi" = \r61 #;};
        "measurePO": V {"_po" = \r46 #;};
        "pulse": V {"pclk" = P;};
    }
    "capture_rstn"{
        W "_default_WFT_";
        C {"all_inputs" = 0\r59 N 0;
            "all_outputs" = \r46 X;};
        F {"test_mode" = 0;};
        "forcePI": V {"_pi" = \r61 #;};
        "measurePO": V {"_po" = \r46 #;};
        "pulse": V {"rstn" = P;};
    }
}
```

**Using Allclock Capture Procedures.** The allclock procedures are used for transition delay and path delay ATPG. One set of generic allclock procedures can replace clock specific at-speed capture-launch procedures. DFT Compiler copies the \_default\_WFT\_ WaveformTable to each of the WaveformTables of the allclock procedure. You need to modify the timing information in the allclock WaveformTables to synchronize with the timing that will be used for the at-speed testing in TetraMAX ATPG.

By default, an allclock procedure applies to a single vector, although it doesn't have to carry the redundant clock parameter. An allclock procedure may reference any WaveformTable for each operation.

For examples of allclock procedures, see [Example 10-5](#), [Example 10-6](#), and [Example 10-7](#).

**Example 10-5 Example of Allclock Procedures**

```
"allclock_capture" {
    W "_allclock_capture_WFT_";
    C {
        "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
        "all_outputs" = \r165 X;
        "all_bidirectionals" = ZZZ;
    }
    F {
        "i_scan_block_sel[0]" = 1;
        "i_scan_block_sel[1]" = 1;
        "i_scan_compress_mode" = 0;
        "i_scan_testmode" = 1;
    }
    V {
        "_po" = \r168 #;
        "_pi" = \r179 #;
    }
}
"allclock_launch" {
    W "_allclock_launch_WFT_";
    C {
        "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
        "all_outputs" = \r165 X;
        "all_bidirectionals" = ZZZ;
    }
}
.....
F {
}
..
V {
    "_po" = \r168 #;
    "_pi" = \r179 #;
}
}
"allclock_launch_capture" {
    W "_allclock_launch_capture_WFT_";
    C {
        "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
        "all_outputs" = \r165 X;
        "all_bidirectionals" = ZZZ;
    }
    F {
        "i_scan_block_sel[0]" = 1;
        "i_scan_block_sel[1]" = 1;
        "i_scan_compress_mode" = 0;
        "i_scan_testmode" = 1;
    }
    V {
        "_po" = \r168 #;
        "_pi" = \r179 #;
    }
}
```

In [Example 10-6](#), a `_default_WFT` timing is copied to an `allclock _capture` procedure.

*Example 10-6 allclock\_capture WaveformTable With Default Timing*

```
WaveformTable "_allclock_capture_WFT_" {
    Period '100ns';
    Waveforms {
        "all_inputs" {
            0 {
                '0ns' D;
            }
        }
        ...
        "all_bidirectionals" {
            L {
                '0ns' Z;
                '40ns' L;
            }
        }
        ...
        "all_outputs" {
            L {
                '0ns' X;
                '40ns' L;
            }
        }
        "i_HEAD_PIPE_REG_CLK" {
            P {
                '0ns' D;
                '45ns' U;
                '55ns' D;
            }
        }
    }
    ....
```

[Example 10-7](#) shows an updated WaveformTable timing copied to an `allclock_capture` procedure.

*Example 10-7 allclock\_capture Procedure WaveformTable With Updated Timing*

```
WaveformTable "_allclock_capture_WFT_" {
    Period '10ns';
    Waveforms {
        "all_inputs" {
            0 {
                '0ns' D;
            }
        }
    ...
    "all_bidirectionals" {
        L {
            '0ns' Z;
            '4ns' L;
        }
    }
    ...
    "all_outputs" {
        L {
            '0ns' X;
            '4ns' L;
        }
    }
    "i_HEAD_PIPE_REG_CLK" {
        P {
            '0ns' D;
            '5ns' U;
            '6ns' D;
        }
    }
}
....
```

**Limitation in the Generic Capture Procedures.** A limitation associated with the capture procedures is the following:

- The `_default_WFT` timing is copied to the `allclock` WaveformTables (`launch_WFT`, `capture_WFT`, `launch_capture_WFT`). You need to modify these WaveformTables with the correct timing before running at-speed ATPG (transition delay, path delay).

For more information on generic capture procedures, see the chapter on STIL procedure files in the *TetraMAX User Guide*.

---

## Exporting Your Design to TetraMAX ATPG

The following steps work only if your design is completely chain-routed and you have successfully validated the scan chains by inspecting the outputs generated by the `dft_drc` and `report_scan_path -chain all` commands.

To export your design to TetraMAX ATPG, do the following:

1. Before starting any work with DFT Compiler, including scan insertion, set the test timing variables to the values specified by your ASIC vendor. If your ASIC vendor does not have specific requirements, use the following values to achieve the best results from TetraMAX ATPG:

```
dc_shell> set test_default_delay 0  
dc_shell> set test_default_bidir_delay 0  
dc_shell> set test_default_strobe 40  
dc_shell> set test_default_period 100
```

2. Identify the netlist format that you are exporting to TetraMAX ATPG, using the `test_stil_netlist_format` environment variable. The syntax is

```
set test_stil_netlist_format db |verilog | vhdl
```

It is important that you identify the netlist format because TetraMAX ATPG handles difficult names, such as those for buses and escaped characters in Verilog, according to the language used in the netlist.

3. To guide netlist formatting, set the environment variables that affect how designs are written out.

Note:

Set the environment variables before you write out the netlist or STIL protocol file.

For example, if you want vectored ports in your Verilog design to be bit-blasted, set the `verilogout_single_bit` variable to true. For more information about environment variables that affect how designs are written out, see the *HDL Compiler (Presto Verilog) Reference Manual* or the *HDL Compiler (Presto VHDL) Reference Manual*.

4. Check for design rule violations by entering the `dft_drc` command:

```
dc_shell> dft_drc
```

For information on these commands, see [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

Fix any design rule violations. Repeat the `dft_drc` command until no design rule violations are found.

5. Write out the netlist. For example, to write out a Verilog netlist, use the following command:

```
dc_shell> write -format verilog -hier \  
          -out filename.v
```

6. Write out the test protocol file, using the STIL protocol. Enter the following command:

```
dc_shell> write_test_protocol -out protocol.spf
```

All of the information that TetraMAX ATPG requires to create ATPG vectors, such as scan pins and constrained signals, is found in the STIL protocol file.

---

## SCANDEF-Based Reordering Flow

You can use DFT Compiler in the IC Compiler environment to perform scan chain ordering and to fix timing violations based on physical information, or you can perform scan chain reordering by using other place and route tools.

This section has the following subsections:

- [Overview](#)
  - [Generation of a SCANDEF File](#)
  - [Impact of DFT Configuration Specification on SCANDEF File Generation](#)
  - [Generating a SCANDEF File for Typical Flows](#)
  - [Generating SCANDEF Files for Hierarchical Flows](#)
  - [Support for Other DFT Features](#)
  - [Limitations With SCANDEF Generation](#)
- 

### Overview

Additional routing is required when you add scan chains to designs. To meet die size and timing requirements, you need to reduce the amount of routing as much as possible. One way to do this is to reorder scan chains.

The physical scan chain implementation in IC Compiler consists of the ability, first, to read and check the integrity of the SCANDEF file and, second, to perform physical scan chain optimization.

You can also use DFT Compiler in the physical environment to perform placement-based scan chain reordering, or you can choose to use other place and route tools.

If you choose one of these other place and route tools, DFT Compiler must assemble scan chains before the physical placement of the scan elements. In this case, DFT Compiler determines the scan chain order based on hierarchy or alphabetical order instead of proximity.

Reordering of scan chains in the place and route tools requires information about the scan chains that exist in the design. This information can be generated from DFT Compiler in a data file called SCANDEF, an ASCII file that specifies a list of “stub” chains that can be reordered by another tool. The boundaries of these stub chains can be an I/O port, a lock-up latch, or a multiplexer. Hence, the number of scan chains defined in the SCANDEF file does not necessarily match the chain count specified during scan chain architecting.

---

## Generation of a SCANDEF File

The tasks you must perform to generate a SCANDEF file are described in the following sections:

- [Reading and Compiling the Design](#)
- [Specifying the Scan Configuration](#)
- [Writing Out the SCANDEF File](#)

## Reading and Compiling the Design

You must first read your design into dc\_shell. If your design is an HDL source file, you must compile it, using either the `compile` command or the `compile -scan` command. For more information, see the Design Compiler documentation and [Chapter 5, “Pre-Scan Test Design Rule Checking.”](#)

## Specifying the Scan Configuration

Make sure you properly set up the intended scan design by using the `set_scan_configuration` command. With this command, you describe such items as clock mixing, scan style, and the number of scan chains.

For more information on the `set_scan_configuration` command, see [Chapter 6, “Architecting Your Test Design.”](#)

## Writing Out the SCANDEF File

In the step before place and route, the `insert_dft` command generates appropriate files for the place and route tools. To generate the .def file,

1. Specify the `insert_dft` command.
2. Specify the `write_scan_def` command:

```
write_scan_def -output filename.def
```

Next, you need to run your place and route tools to generate a new scan chain routing order.

Generate the .ddc file for the step after place and route. Write out an ASCII netlist for the place and route tool. For example,

```
dc_shell> write -f db -hier cpu -out mydesign.ddc  
dc_shell> write -f verilog -hier cpu -out mydesign.v
```

---

## Generating a SCANDEF File for Typical Flows

The complete flow capturing the generation of a SCANDEF file is as follows:

```
read_file -f ddc top.ddc  
current_design top  
set_scan_configuration -style multiplexed_flip_flop  
set_dft_signal -view existing_dft -type ScanClock \  
-port clock -timing [list 45 55]  
create_test_protocol  
dft_drc  
preview_dft  
insert_dft  
change_name  
write_scan_def -o my_def.def  
write_test_protocol -o test_mode.spf  
write -f verilog -hier -o top.out
```

---

## Generating SCANDEF Files for Hierarchical Flows

The following flow is used at the top level to generate the SCANDEF file in hierarchical flows:

```

## Reading top-level design
read_verilog top.v

## Reading test-models
read_test_model -format ddcblock.ctlddc

current_design top
link
use_test_model -true top
set_scan_configuration -style multiplexed_flip_flop
set_dft_signal -view existing_dft -type ScanClock \
-port clock -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
change_name
write_scan_def -o my_def.def
write_test_protocol -o test_mode.spf

## Removing the test-model blocks so that Design Compiler
does not write empty modules for the blocks
remove_design block
write -f verilog -hier -o top.out

```

## Hierarchical SCANDEF Flow Support

Hierarchical Scan Synthesis flows abstract the scan information in DFT-inserted blocks in the form of a core test language model (CTL model). You can use CTL models instead of the netlist representation of the subblocks during chip-level scan integration. In such cases, the SCANDEF file represents the scan segments within these subblocks as black-box segments. This representation permits repartitioning of the segment as a whole but does not permit the reordering cells within the segments. This feature allows DFT Compiler to write the scan cells of the subblocks, so the tool can reorder the subblock scan cells along with the chip-level scan cells.

**Note:**

Hierarchical SCANDEF flows are not supported on IBM AIX platforms

## DFT Commands

Use the following DFT Compiler command with the -expand option at the chip-level to write the SCANDEF file with the scan cells of the subblocks.

```
write_scan_def -expand [list_of_instances]
```

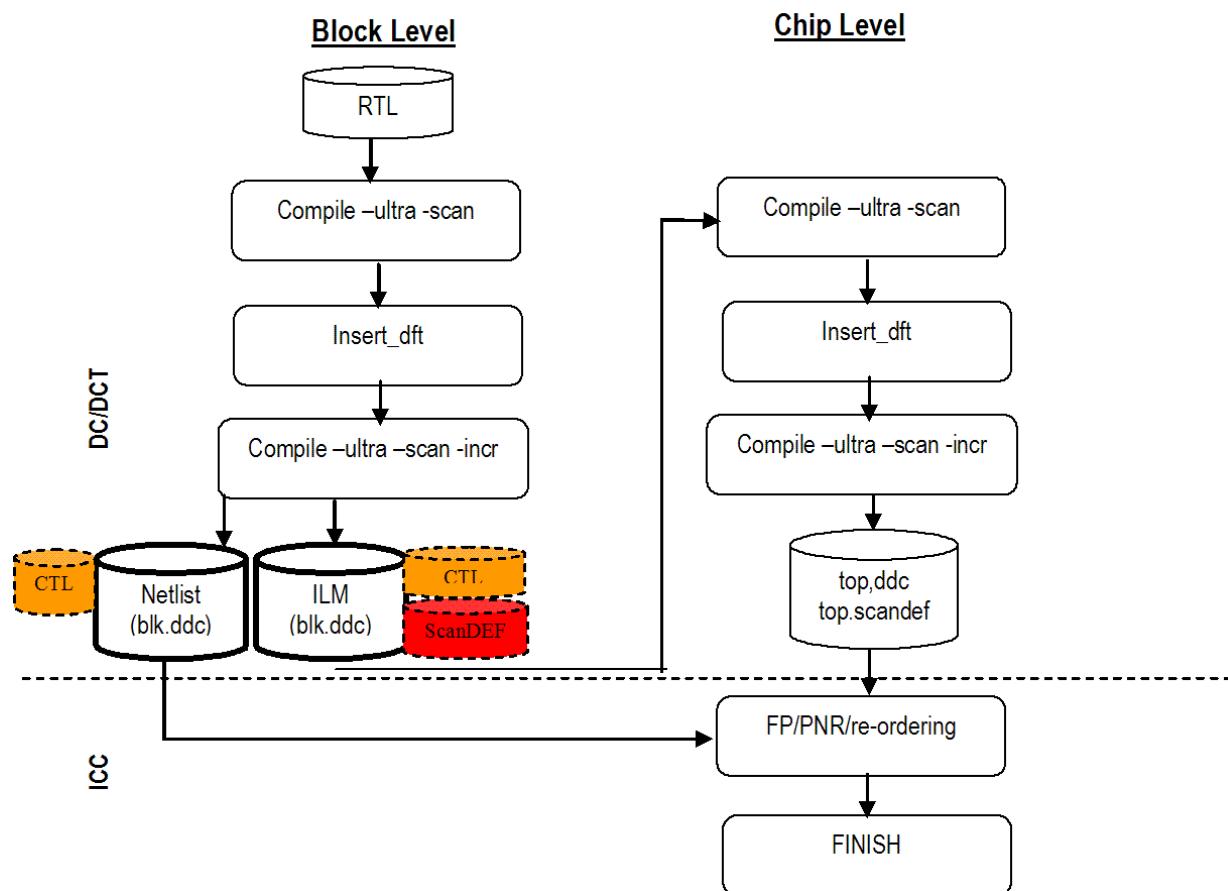
The `-expand` option is used to specify a list of instances abstracted into CTL models. The scan segments of these models must be treated as “flat,” fully visible segments when generating the chip-level SCANDEF file.

The default behavior is to treat all instances abstracted into CTL models as black boxes. This means that they are represented in the SCANDEF file by the “BITS” construct.

## Hierarchical SCANDEF Flows

[Figure 10-1](#) shows the hierarchical SCANDEF flow.

*Figure 10-1 Hierarchical SCANDEF Flow*



In this flow, the block is logically optimized and DFT-complete before chip level integration. Test information of the block is abstracted in CTL models. (You can also choose to abstract the block into an ILM representation.) During chip integration, you can choose to use the full netlist or the ILM representation of the block to perform optimizations.

The block CTL model can be used to insert chip-level DFT without disturbing the block-level DFT structures in a hierarchical scan synthesis flow. Using the block-level CTL model or the complete netlist, the chip-level SCANDEF file can be generated by treating the block-level scan segments either as black-box segments or as flat, fully visible segments. IC Compiler uses the full netlist representation of the block to perform global physical optimizations. You can choose which block are flattened at the chip-level SCANDEF generation by using the `-expand` option of the `write_scan_def` command.

The following hierarchical SCANDEF flows are supported:

- Flow 1 – The block-level CTL models are used at the chip level for DFT insertion.
- Flow 2 – The complete block-level netlists (.ddc files) along with SCANDEF information at the block-level are used at the chip level for DFT insertion.
- Flow 3 – The complete block-level netlists (.ddc files) are used at the chip level for DFT insertion.

## Flow 1

### *Block level*

```
read_ddc sub1_test_ready.ddc
current_design sub1
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
compile_ultra -incremental -scan
dft_drc
change_names -rules verilog -hier
write_scan_def -o sub1.def
write_test_model -o sub1.ctlddb -format ddb
write -f verilog -hier -o sub1.v
write_test_protocol -out sub1_scan.spf
```

### *Chip level*

```
read_verilog ./top.v
read_ddc sub1.ctlddb
read_ddc sub2.ctlddb
use_test_model -true [list sub1 sub2]
current_design TOP
link
set_scan_configuration -chain 10
set_dft_signal -port clock -type Scanclock -view existing \
    -timing [list 45 55]
create_test_protocol
```

```

dft_drc
preview_dft
insert_dft
write_scan_def -o top.def -expand [list inst_sub1 inst_sub2]

```

Note: inst\_sub1 and inst\_sub2 are instance names of modules sub1 and sub2, respectively.

## Flow 2

### *Block level*

```

read_ddc sub1_test_ready.ddc
current_design sub1
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
compile_ultra -incremental -scan
dft_drc
change_names -rules verilog -hier
write_scan_def -o ./sub1.def
write -f ddc -o ./sub1.ddc -hier

```

### *Chip level*

```

read_verilog ./top.v
read_ddc sub1.ddc
read_ddc sub2.ddc
use_test_model -true [list sub1 sub2]
current_design TOP
link
set_scan_configuration -chain 10
set_dft_signal -port clock -type Scanclock -view existing \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
write_scan_def -o top.def -expand [list inst_sub1 inst_sub2]

```

Note: inst\_sub1 and inst\_sub2 are instance names of modules sub1 and sub2, respectively.

## Flow 3

### *Block level*

```
read_ddc sub1_test_ready.ddc
current_design sub1
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
compile_ultra -incremental -scan
dft_drc
change_names -rules verilog -hier
write -f ddc -o ./sub1.ddc -hier
```

### *Chip level*

```
read_verilog ./top.v
read_ddc sub1.ddc
read_ddc sub2.ddc
use_test_model -true [list sub1 sub2]
current_design TOP
link
set_scan_configuration -chain 10
set_dft_signal -port clock -type Scanclock -view existing \
create_test_protocol
    -timing [list 45 55]
dft_drc
preview_dft
insert_dft
write_scan_def -o top.def -expand [list inst_sub1 inst_sub2]
```

Note: inst\_sub1 and inst\_sub2 are instance names of modules sub1 and sub2, respectively.

## Limitations

- The expanded hierarchical SCANDEF is based solely on the nonexpanded hierarchical SCANDEF; that is, the expanded SCANDEF can only include cells that are represented by the “BITS” construct in the nonexpanded version.
- In Flow 1, where you use block-level test models for chip-level DFT insertion, you need to write a SCANDEF file at the block level before writing out a CTL model.
- In Flow 3, where you use the block-level netlists for chip-level DFT insertion and don’t write a SCANDEF file before writing the netlist, the block-level scan constraints, such as scan\_path or scan\_group, are not be considered when expanding.

---

## **Impact of DFT Configuration Specification on SCANDEF File Generation**

The impact of the specified DFT configuration on the final SCANDEF file is shown in the following test cases.

Consider a design with a set of scan flip-flops, f1, f2, f3, f4.

### **Case 1:**

```
set_scan_path chain1 -exact_length 2
```

The scan elements in this scan chain can be swapped with those of other scan chains. The SCANDEF file is as follows:

```
- CHAIN1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
    f2 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so ;
+ PARTITION CLOCK1_45_55
```

This case is similar to `set_scan_path chain1 -chain_length 2`.

### **Case 2:**

```
set_scan_path chain1 -head {f1}
    -tail {f3} -include {f4}
```

The SCANDEF file is as follows:

```
- CHAIN1
+ START f1 Q
+ FLOATING f4 ( IN SDI ) ( OUT Q )
+ STOP f3 SD1 ;
```

In this case, because we have some include elements, the “def” scan chain does not have the PARTITION attribute.

### **Case 3:**

```
set_scan_path chain1 -include {f4} -exact_length 2
```

This case is a combination of case 1 and case 2. The element f4 must belong to the scan chain and cannot be used for partitioning. Elements that are added are swapped with other scan chains. However, due to the .def format limitation, consider all the elements as “INCLUDE” constructs. Therefore, if f2 is the added element, the SCANDEF is as follows:

```

- CHAIN1
+ START PIN test_si
+ FLOATING f4 ( IN SDI ) ( OUT Q )
    f2 (IN SDI ) ( OUT Q )
+ STOP PIN test_so ;

```

### **Case 4:**

```
set_scan_path chain1 -include {f1 f2} -complete
```

In this case, the scan chain is not modified. The SCANDEF file is as follows:

```

- CHAIN1
+ START PIN test_si
+ ORDERED f1 ( IN SDI ) ( OUT Q )
    f2 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so ;

```

### **Case 5:**

```
set_scan_group group1 -include_elements \
[list f1 f2] -serial_routed true ;
set_scan_path chain1 -include [list group1 f3 f4] ;
```

The SCANDEF is as follows:

```

- CHAIN1
+ START PIN test_si
+ ORDERED f1 ( IN SDI ) ( OUT Q )
    f2 ( IN SDI ) ( OUT Q )
+ FLOATING f3 ( IN SDI ) ( OUT Q )
    f4 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so

```

### **Case 6:**

```
set_scan_group group1 -include_elements [list f1 f2] -serial_routed
false ;

set_scan_path chain1 -include [list group1 f3 f4 f5 f6] ;
```

Consider that `test_si` and `test_so` signify the scan-in and the scan-out ports of the scan chain, `chain1`. Then, two subchains are considered and the SCANDEF file is as follows:

```

- SUB_GP1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
f2 ( IN SDI ) ( OUT Q )
+ STOP f3 SDI ;

-SUB_GP2
+ START f3 q ;
+ FLOATING f4 ( IN SDI ) ( OUT Q )
f5 ( IN SDI ) ( OUT Q )
f6 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so;

```

### **Case 7:**

Consider a top-level design with some flip-flops and a test model for a block—block1 with scan ports `test_si1` and `test_so1` and a scan chain of length 20. The SCANDEF is as follows:

```

- SUB_GP1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
    f2 ( IN SDI ) ( OUT Q )
    block1 (IN test_si1) (OUT test_so1 ) ( BITS 20 )
+ PARTITION pll_clk3_clk_45_45
+ STOP f3 SDI ;

```

The value of the BITS parameter indicates the length of the scan chain in the CTL model. This length is maintained during partitioning.

### **Case 8:**

Consider a design in which the scan chain begins with a scan port `test_si` and ends at scan port `test_so`. When you run `insert_level_shifters`, consider the level shifters to be inserted at the test ports. The SCANDEF file is as follows:

```

- SUB_GP1
+ START LS_test_si Y
+ FLOATING f1 ( IN SDI ) ( OUT Q )
    f2 ( IN SDI ) ( OUT Q )
+ PARTITION pll_clk3_clk_45_45
+ STOP LS_test_so A;

```

Here, `LS_test_si` and `LS_test_so` are the inserted level shifters. So, the chain begins at the output of the `LS_test_si` level shifter and ends at the input of the `LS_test_so` level shifter.

---

## **Support for Other DFT Features**

The following DFT features are supported:

- Basic scan and adaptive scan
- Multimode scan
- Internal pins
- Memories with test models
- Multivoltage support
- Hierarchical flows (with test models)
- PLL flows
- Core wrapping

---

## **Limitations With SCANDEF Generation**

SCANDEF is not supported for the following:

- BSD Compiler
- Scan extraction flows that have combinational logic between two adjacent scan flip-flops

---

## **Exporting of SCANDEF Files to Third-Party Tools**

DFT Compiler inserts scan into the design, creating files that contain scan instance information required for placement-based scan chain reordering. The Astro tool or third party tools can be used to read these files and determine the scan order based on placement.



# Index

---

## Numerics

0-input connection 7-27  
1-input connection 7-27  
2-input AND gate 7-28

## A

-add\_lockup option, set\_scan\_configuration command 6-45  
analog circuitry 1-48  
AND gate 7-28  
asynchronous behavior 1-48  
asynchronous control pins violations 2-14  
asynchronous inputs, AutoFix and 7-20  
asynchronous pin  
  uncontrollable  
    unreliable data capture 6-72  
asynchronous pins  
  uncontrollable 4-11  
    correcting 4-12  
    fault coverage impact 4-11  
asynchronous signals 2-4  
asynchs, defining 7-33  
ATE clock 9-1  
ATE Clocks  
  OCC definition 9-4  
ATPG 1-48

At-speed testing 9-3  
attributes  
  clock\_gate\_test\_pin 1-54  
  dont\_touch 4-8  
  locating 4-9  
  removing 4-9  
  scan\_element 4-8  
  test\_asynch 2-14  
  test\_asynch\_inverted 2-14  
AutoFix 7-28  
  asynchronous inputs 7-20  
  clock configuration 7-20  
  finding controllable clocks 7-20  
  running with preview\_dft 4-39  
  sample scripts 7-20  
  set\_dft\_signal commands for 7-19  
  summary of 7-16  
  top-down example 7-22  
  using 7-14  
available STIL formats 5-26

**B**

behavioral modeling 5-13  
bidirectional delay  
  (*see also* test\_default\_bidir\_delay variable)  
    5-21  
default 5-21

incorrect, risks of 5-21  
requirements 5-21  
specifying 5-21  
usage 5-21  
bidirectional ports  
    adding disabling logic to 6-42  
    degenerated 6-43  
    design characteristics 5-26  
    handling 6-42  
    meeting vendor requirements 6-42  
    using as  
        scan input 6-32  
        scan output 6-32  
bidirectional ports, wrapping 8-17, 8-20  
bidirectional timing 5-21  
black box  
    violations 2-16, 2-22  
black box cell  
    causes 5-13  
    set\_test\_assume 5-13  
    structural model 5-14  
black box cell, scan specification 6-16  
block-by-block design strategy 1-17  
bottom-up compile 4-30  
bottom-up design flow  
    comparing results with top-down 6-17  
    inserting three-state disabling logic 6-40  
    overview 1-41  
    support for mixed scan states 4-32  
boundary I/O registers 8-10  
bus contention, preventing during scan shift  
    6-38

clock chain 9-9  
clock fixing 7-28  
clock gating  
    improving testability 1-48  
    inserting control points 1-49  
    inserting observation points 1-52  
    latch-based configurations 1-59  
    latch-free configurations 1-60  
observability  
    logic depth, choosing 1-54  
    observability logic 1-54  
    scan mode versus test mode 1-50  
clock mixing 8-13  
clock MUXing logic 9-9  
clock shapers 9-1  
clock signal, specifying 7-33  
clock signals, internally generated 1-47  
clock timing  
    default 5-23  
    effect of incorrect 6-67  
    meeting vendor requirements 5-22  
clock\_gate\_test\_pin attribute 1-54  
-clock\_mixing option, set\_scan\_configuration  
    command 6-11, 6-49  
clocked scan 4-13

## C

capturing scan data, requirements 6-70  
-chain\_count option  
    set\_scan\_configuration command 6-10  
-chain\_count option, set\_scan\_configuration  
    command 6-10  
check\_design command 4-32

clocking constraints  
    requirements for valid clocks 4-9

clocks  
    AutoFix and 7-20  
    feeding multiple register inputs 2-18  
    gating violations 2-17  
    identifying test clocks 2-3  
    interaction violations 2-20  
    internally generated 7-25  
    multiple 2-21  
    test point 7-19  
    uncontrollable 2-13  
        used as data violation 2-15

clocks, defining 7-33

combinational feedback loops, effect on scan design 1-47

combinational feedback, violations 2-19

commands 2-13

- check\_design 4-32
- compile 4-26, 7-15
- create\_multibit 4-21
- create\_test\_protocol 6-47
- current\_test\_mode 8-26, 8-29
- dft\_drc 4-16, 7-15, 7-21
  - violation reporting 5-9
- filter 4-16
- help 5-9
- hookup\_testports 1-54
- insert\_dft 4-32, 7-15, 7-21
  - default three-state net behavior 6-39
  - invoking synthesis process 4-39
  - pad cells 1-46
- link 4-31, 5-13
- list\_test\_models 1-26
- preview\_dft 4-39, 6-6, 6-32, 7-15, 7-21
  - creating a specification dc\_shell script 4-39
- preview\_dftcommand 6-15
- read\_test\_model 1-26, 1-29
- remove\_dft\_configuration 7-19
- remove\_multibit 4-21
- remove\_scan\_configuration 6-59
- remove\_scan\_specification

set\_scan\_state 6-35  
 set\_signal\_type 6-22  
     establishing signal type attributes 6-61  
 set\_test\_hold 2-4  
 write\_test\_model 1-25, 1-28  
 write\_test\_protocol 5-34  
 compile command 7-15  
 compile command, test-ready 1-5  
 compile -scan command 4-26  
     specifying scan cells 4-18  
 compiled cells 1-48  
 -complete option, set\_scan\_path command 6-16  
 configuration, wrapper 8-24  
 constant logic values, defining for test mode 2-4  
 constraint-optimized scan insertion  
     degeneration support 4-33  
     process flow, scan replacement 4-34  
     scan replacement 4-32  
     support for mixed scan states 4-32  
 constraint-optimized scan insertion, when to use 4-4  
 constraints, design 1-4, 1-10, 1-15  
 Control Clock, specifying 7-31  
 control point 1-49  
 control signal 7-31  
 Control Signal, specifying 7-31  
 control signal, specifying 7-33  
 control signals 2-4  
 control wrapper chains, wrapper chains, control 8-22  
 Control\_0 test point 7-28  
 Control\_01 test point 7-29  
 Control\_1 test point 7-28  
 Control\_Z0 test point 7-29  
 Control\_z01 test point 7-29  
 Control\_Z1 test point 7-29  
 controllability 1-47  
 correcting  
     black box library cells 5-14  
     clock driving data 6-71  
     hold-time violations 6-53  
     invalid clock gating 4-11  
     uncontrollable asynchronous pins 4-12  
     uncontrollable clocks 4-11  
     unresolved references 4-31  
 correcting invalid clock gating violations (see test mode) 6-67  
 coverage, increasing 7-30  
 create\_multibit command 4-21  
 create\_test\_protocol command 7-33  
 C-rules 9-26  
 current\_test\_mode command 8-26, 8-29

## D

data feedthrough 2-16  
 -dedicated\_scan\_ports option, set\_scan\_configuration command 6-34  
 default test protocol  
     specifying timing information 5-20  
 definitions  
     dummy scan port 6-31  
     nonscan design 4-32  
     scan configuration 6-3  
     scan design 4-33  
     scan insertion 4-32  
     scan lock-up latch 6-43  
     scan replacement 4-1  
     scan specification 6-5  
     system clock 4-9  
     test clock 4-9  
     test protocol 1-43  
     test-ready compile 4-23  
     unresolved reference 4-31  
     unrouted scan design 4-33  
     untestable functional path 6-70  
 dependent slave operation. 10-5  
 depth\_value 1-53  
 design characteristics and test protocol 5-25

design constraints 1-4, 1-10, 1-15  
design flow, test point insertion 7-14  
design state, unrouted scan 4-25  
design strategy, three-state nets 6-39  
DesignWare IP 9-12  
DFT Compiler  
    interoperability issues 1-57  
dft\_drc  
    after building scan chains 6-60  
    and insert\_dft 5-10  
    clocks inferred by set\_dft\_signal 6-67  
    directing output to a file 5-9  
    dont\_touch attribute 5-17  
    extracting scan chains 6-62, 7-2  
    functional cell models 5-13  
    illegal driver 6-62  
    nonscan latches 5-18  
    non-three-state driver 6-61  
    set\_dft\_signal 6-66  
    set\_scan\_element command 5-17  
    summarizing violations 5-6  
    topological checks 6-61  
dft\_drc command 4-16, 5-18, 7-15, 7-21, 7-33,  
    10-4, 10-15  
    fixing violations with AutoFix 7-16  
    identifying  
        scan cells 4-16  
        scan equivalents 4-17  
    identifying invalid clock nets 4-11  
messages  
    error (*see also* messages, error) 5-9  
    warning (*see also* messages, warning) 5-8  
messages, information 4-8  
    (<seeitalic>see also messages,  
        information)  
-verbose option 5-9, 5-10  
    violation reporting 5-9  
-disable option, set\_scan\_configuration  
    command 6-39  
disabling logic  
    adding to bidirectional ports 6-42  
dont\_touch attribute 4-8

dont\_touch attribute, and dft\_drc 5-17  
D-rules 9-26

## E

environment variables  
    test\_clock\_in\_port\_naming\_style 6-4  
    test\_clock\_out\_port\_naming\_style 6-4  
    test\_clock\_port\_naming\_style 6-4, 6-28  
    test\_default\_scan\_style 5-32  
    test\_scan\_clock\_a\_port\_naming\_style 6-4,  
        6-28  
    test\_scan\_clock\_b\_port\_naming\_style 6-4  
    test\_scan\_clock\_port\_naming\_style 6-4,  
        6-28  
    test\_scan\_enable\_inverted\_port\_naming\_st  
        yle 6-4, 6-28  
    test\_scan\_enable\_port\_naming\_style 6-4,  
        6-28  
    test\_scan\_in\_port\_naming\_style 6-28  
    test\_scan\_out\_port\_naming\_style 6-28

excluding cells from AutoFix 7-20  
excluding ports from wrapping 8-18  
existing scan chains 6-22  
External Clocks  
    OCC definition 9-4  
EXTEST mode 8-5

## F

fault coverage  
    and controllability 1-47  
    evaluating  
        sequential cell summary 5-10  
impact  
    clock driving data 6-71  
    invalid clock nets 4-9  
    uncontrollable asynchronous pin 4-11  
fault coverage, impact  
    scan equivalents 4-14  
fault models 9-3  
feedback violations 2-19

feedthrough 2-16  
filter command 4-16  
finding controllable clocks 7-20  
flows  
    bottom-up scan insertion 1-42  
    Physical Flow 9-19  
    scan replacement 4-3  
    top-down scan insertion 1-44  
Force\_0 test point 7-27  
Force\_01 test point 7-27  
Force\_1 test point 7-27  
Force\_z0 test point 7-27  
Force\_z01 test point 7-27  
Force\_z1 test point 7-27  
forfixing  
    clock-as-data 7-28  
    X propagation 7-28  
format  
    STIL 2-4  
functional mode 8-6  
functional model 5-13

## G

gated clocks 1-47

## H

hard-to-observe nodes 7-30  
HDL-level circuit description 1-13  
help command 5-9  
hierarchical isolation 6-34  
hierarchical scan insertion 4-35  
hierarchical scan synthesis, benefits of 1-41  
hold-time violations  
    avoiding 6-50  
    correcting 6-53  
hookup\_testports command 1-54

|  
identifying  
    black box cells 5-13  
    scan cells 4-16  
        uncontrollable 6-63  
    scan equivalents 4-17  
    scan ports, existing 6-22  
identifying invalid clock nets 4-11  
identifying test clocks 2-3  
If 5-14  
increasing coverage 7-30  
incremental compiles 4-30  
inferred test protocol  
    bidirectional delay 5-21  
    customizing  
        set\_dft\_signal command 6-47  
    default delay 5-20  
    examining 5-34  
    scan shift and parallel cycles 5-32  
information message, dft\_drc 5-8  
initialization protocol  
    STIL format 2-4  
initialization requirements, defining 2-4  
input  
    delay  
        default 5-20  
        specifying 5-20  
    timing 5-20  
insert\_dft command 4-32, 6-61, 7-15, 7-21,  
    7-34  
    and dft\_drc 5-10  
    dedicated test clocks 6-55  
    handling internal clock signals 6-49  
    invoking synthesis process 4-39  
    pad cells 1-46  
    specifying scan cells 4-18  
    three-state net behavior 6-39  
-insert\_terminal\_lockup option 6-44  
insertion on tri-states 7-29  
interface, wrapper cell 8-9

Internal Clocks  
    OCC definition 9-4  
internal clocks 9-9  
internally generated clocks 7-25  
INTEST mode 8-4  
inverted test mode  
    creating a 7-25

## L

latch, scan lock-up 6-43  
latches  
    requiring multiple clocks violations 2-21  
launch before capture violations 2-16  
launch clock 9-3  
level-sensitive scan design (<seeitalic>see  
    LSSD)  
limitations  
    inserting three-state disabling logic 6-38  
link command 4-31, 5-13  
link\_library variable 4-31  
list\_test\_models command 1-26  
listing scan equivalents 4-17  
load\_unload 9-1  
load\_unload procedure  
    STIL format 5-29  
loading scan data, requirements 6-63  
location of test point, specifying 7-31  
-lockup\_type option 6-45  
-longest\_chain\_length option,  
    set\_scan\_configuration command 6-9  
LSSD  
    advantages 4-13  
    controlling slave clock routing 6-56  
    disadvantages 4-13  
    multiple master clocks 6-55

## M

macro functions 1-48

mapping your circuit description 1-5  
meeting vendor requirements  
    bidirectional ports 6-42  
    scan chain count 6-10  
    test timing  
        clock waveforms 5-22  
memory, reducing use of 1-18  
messages  
    error  
        TESTDB-256 6-16  
    information 5-8  
        TEST-121 4-9  
        TEST-202 4-8  
    warning  
        TEST-116 4-12  
        TEST-120 4-6, 4-17  
        TEST-169 4-11  
        TEST-186 4-11  
        TEST-224 4-6, 4-16  
        TEST-342 6-16  
        TEST-353 6-16  
        TEST-376 6-17  
        TEST-451 5-13  
        TEST-462 4-7  
        TEST-463 4-7  
        TEST-464 4-7  
        TEST-465 4-7  
        TEST-466 4-8  
        TEST-467 4-8  
        TEST-468 4-7  
        UIT-227 6-45  
        UIT-229 6-49  
mixed clocks 6-51  
modes  
    activating 8-15  
    creating additional 8-22  
    EXTEST 8-5  
    functional 8-6  
    INTEST 8-4  
    NORMAL 8-6  
    SAFE 8-7  
    specifying multiple modes 8-18

- test wrapper 8-4
- multibit
  - component scan replacement 4-22
  - components 4-21
  - components treated as synthesizable
    - segments 6-14
  - controlling test synthesis 4-22
  - specifying components as synthesizable
    - segments 6-14
  - supported library cells 5-15
- multiple clocks
  - LSSD designs 6-55
  - required by latch 2-21
  - scan assembly 6-49
    - correcting hold-time violations 6-53
    - scan lock-up latches 6-44
  - scan chain ordering 6-16
  - scan insertion
    - balancing scan chains 6-11
    - default behavior 6-11
    - mixing clocks 6-13
    - mixing edges 6-12
  - scan specification 6-16
  - violations 2-21
- multiplexed flip-flop, advantages and disadvantages 4-13
- multiplexer 7-28
- multiplexer connection 7-27
- multipliers 9-1

## N

- netlist, writing 7-34
- nonscan latch modeling 5-18
- NORMAL mode 8-6
- Number of Test Points Per Test Point Enable, specifying 7-32

## O

- observability logic, logic depth 1-54

- Observe test points 7-30
- OCC
  - capabilities 9-5
  - defining clocks 9-13
  - definitions 9-4
  - Design Flow 9-6
  - limitations 9-5
  - setting up the environment 9-12
  - supported flows 9-4
- OCC and Clock Chain Synthesis Insertion Flow 9-12
- OCC controller
  - configuring 9-16
  - logical representation 9-10
- OCC support
  - enabling 9-12
  - example configurations on a design 9-27
- off-chip clock signals 1-47
- On-chip clock
  - controller 9-9
- online help, messages 5-9
- optimization
  - default compile
    - Verilog example 4-27
    - VHDL example 4-28
  - preparing for 4-25
  - test-ready compile 4-23, 4-29
- output
  - strobe
    - default 5-22
    - specifying 5-22

## P

- pad cells
  - insert\_dft command 1-46
- phased-locked loops 9-1
- PLL
  - clock 9-9
  - example design flow 9-6
- PLL Clocks

OCC definition 9-4  
 PLL clocks, defining 9-13  
 PLL support, initiating 9-12  
 port name
 

- naming style variables 6-28
- `test_clock_in_port_naming_style` 6-4
- `test_clock_out_port_naming_style` 6-4
- `test_clock_port_naming_style` 6-4, 6-28
- `test_scan_clock_a_port_naming_style` 6-4, 6-28
- `test_scan_clock_b_port_naming_style` 6-4
- `test_scan_clock_port_naming_style` 6-4, 6-28
- `test_scan_enable_inverted_port_naming_style` 6-4, 6-28
- `test_scan_enable_port_naming_style` 6-4, 6-28
- `test_scan_in_port_naming_style` 6-28
- `test_scan_out_naming_style` 6-28

 post-DRC, running 7-34  
 Power Compiler
 

- interoperability issues 1-57
- using scan enables 1-58
- using `test_mode` 1-57

 Power Saving option 7-33  
 pre-DRC, running 7-33  
`preview_dft` command 4-39, 6-6, 6-32, 7-15, 7-21, 7-34
 

- creating a specification dc\_shell script 4-39
- versus the `report_test` command 6-7
- <seeitalic>see also previewing

`preview_dftcommand` 6-15  
 previewing
 

- scan lock-up latches 6-46

 protocol file
 

- design examples 2-8

  
**R**  
 RAM cells 1-48  
 rapid scan synthesis 6-18

read\_test\_model command 1-26, 1-29  
 reducing test data volume 7-30  
 Reference Clocks
 

- OCC definition 9-4
- reference clocks, defining 9-13
- reference, unresolved 4-31
- registered clock gating circuitry violations 2-17
- `remove_dft_configuration` command 7-19
- `remove_multibit` command 4-21
- `remove_scan_specification` command
  - syntax 6-36
- removing attributes 4-9

`report_scan_path -chain` command 10-4  
`report_test` command 6-25, 6-47, 7-34
 

- port option 6-61
- scan\_path option 6-62
- versus the `preview_dft` command 6-7

 requirements
 

- bidirectional delay 5-21
- capturing scan data 6-70
- loading scan data 6-63

 requirements, clocks 4-9  
 rerouting
 

- scan path 6-17

`reset_design` command 6-59  
`reset_dft_configuration` command 4-38  
`reset_scan_configuration` command 6-59  
`reset_scan_specification` command 4-38

## S

SAFE mode 8-7  
 scan architecture
 

- modifying 6-59
- specifying 6-3

 scan assembly
 

- basic command flow 6-3
- correcting hold-time violations 6-53
- design optimization 6-17
- determining scan chain count 6-10

determining scan chain length 6-8  
multiple clock designs 6-49  
scan lock-up latches 6-44  
selecting test ports 6-31  
Scan ATPG clock chain 9-9  
scan cell 4-16  
    capturing data, requirements 6-70  
    determining scan functionality 4-14  
    exclusion conditions 6-9, 6-20  
    library description 4-15  
    loading data, requirements 6-63  
    specifying 4-18  
    uncontrollable  
        causes 6-63  
        identifying 6-63  
        incorrect clock timing 6-67  
        invalid clock logic 6-64  
        missing scan-input port 6-64  
        nonscan cell 6-69  
    unreliable data capture  
        causes 6-70  
        clock driving data 6-71  
        uncontrollable asynchronous pin 6-72  
        untestable functional path 6-72  
    unrouted, causes 6-18  
scan cell replacement 6-19, 6-20  
scan chain 6-62  
    associate scan enable ports 6-33  
    avoiding hold-time violations 6-50  
    balancing  
        multiple clock designs 6-11  
    definition 6-62  
    determining count 6-10  
    determining length 6-8  
    existing 6-22  
    extraction 6-62, 7-2  
    internal clock signals 6-49  
    protocol dependence 6-62  
    rerouting 6-17  
    scan-in pin sharing 6-31, 6-32  
    specifying 6-5  
    specifying for the current design 6-8  
    stitching 6-18  
    with asserted scan\_enable 1-58  
scan chains, previewing 7-34  
scan configuration 6-3  
    modifying 6-59  
    specifying 6-3  
scan connectivity violations 6-62  
scan control register 7-29  
scan design  
    previewing 6-6  
scan design technique 1-47  
scan element, diagram 6-6  
scan enable 1-50  
scan enable signal 9-9  
scan enable signals 2-4  
scan equivalents  
    fault coverage impact 4-14  
    identifying 4-17  
    listing 4-17  
scan implementation, controlling 4-26  
scan input  
    bidirectional 6-32  
    sharing 6-32  
scan input port, sharing 6-32  
scan insertion 7-34  
    bottom-up 1-41  
    definition 4-32  
    hierarchical 4-35  
    mixed clocks 6-51  
preparation 6-3  
    defining a test mode 6-4  
    design constraints 6-4  
    specifying test ports 6-4  
previewing (*see also* preview\_dft command)  
    4-39  
process  
    internal clocks 6-49  
specifying scan chain configuration 4-37  
    reset\_scan\_specification command 4-38  
    set\_dft\_signal command 4-38  
    set\_scan\_configuration command 4-38

set\_scan\_element command 4-38  
 set\_scan\_path command 4-38  
 synthesis (*see also* insert\_dft command)  
     4-39  
 top-down 1-43  
 scan link  
     diagram 6-6  
 scan lock-up element 6-43  
 scan lock-up elements  
     disabling 6-45  
 scan lock-up latch  
     inserting 6-44  
     previewing 6-46  
 -scan option, compile command 4-26  
 scan ordering  
     controlling 6-15  
     rerouting 6-17  
     validating 6-16  
 scan output  
     bidirectional 6-32  
     dedicating 6-34  
     selecting 6-17  
     sharing 6-32  
     sharing with constant signal 6-32  
     three-state 6-32  
 scan port 6-22  
     dummy 6-31  
 scan register 7-30  
 scan register connection 7-27  
 scan replacement  
     definition 4-1  
     effects of violations 5-9  
     flow 4-3  
     method 4-33  
     preventing 4-8  
     selecting a strategy 4-5  
     timing impact 4-33  
     using constraint-optimized scan insertion  
         4-32  
     using test-ready compile 4-23  
 scan routing 6-15  
 scan segment  
     diagram 6-6  
 scan segments 6-6  
 scan shift  
     auxiliary-clock LSSD scan style 5-33  
     clocked-scan scan style 5-33  
     LSSD scan style 5-33  
     multiplexed flip-flop scan style 5-32  
 Scan shift clocks 9-3  
 scan signals, specifying for the current design  
     6-26  
 scan specifications 6-5  
     consistency 6-37  
     removing 6-36  
 scan state 6-35  
 scan states  
     nonscan 4-32  
     scan 4-33  
     supported by constraint-optimized scan  
         insertion 4-32  
         unrouted scan 4-33  
 scan style  
     selecting 4-14  
     selecting, in design flow 1-5, 1-10, 1-15  
     supported options 4-12  
 scan\_element attribute 4-8  
 scan\_enable 1-58  
 scan\_enable port 1-54  
 scan\_enable signal 1-50  
 scan\_source\_or\_sink command 7-31  
 scan\_style attribute 5-25  
 ScanEnable signal type 7-31  
 scan-in pin sharing 6-31, 6-32  
 scannable cell equivalents in library 1-47  
 scan-test signals, port naming style variables  
     6-28  
 scripts  
     sample AutoFix 7-20  
 selecting  
     scan replacement strategy 4-5  
     scan style 4-14

- test ports 6-31
- select-input 7-28
- sensitizable feedback loops 2-19
- sequential cell
  - summary report 5-10
- sequential cells
  - exclusion from scan chains 5-10
  - summary
    - with violations 5-11
    - without violations 5-12
- set link\_library 9-12
- set synthetic\_library 9-12
- set\_attribute command 1-54
- set\_automix\_clock command
  - internally generated clocks and 7-25
- set\_automix\_configuration command 4-39, 7-20
- set\_automix\_element command 4-39, 7-20
- set\_clock\_gating\_style command 1-50, 1-53, 1-57
- set\_core\_wrapper configuration command
  - register\_io\_implementation 8-13
- set\_core\_wrapper\_cell command 8-13, 8-18, 8-24, 8-28
- set\_core\_wrapper\_configuration command
  - 8-10, 8-11, 8-18, 8-21, 8-22, 8-24, 8-28
  - chain\_count option 8-19, 8-25
  - dedicated\_wrapper\_cell 8-12
  - longest\_chain\_length option 8-19, 8-25
  - one\_wrapper\_clock 8-13
- set\_core\_wrapper\_path command 8-20, 8-25
- set\_dft\_clock\_controller command 9-16
- set\_dft\_configuration command 4-39, 7-19, 8-24, 8-28, 9-12
- set\_dft\_signal 9-13, 9-19
  - AutoFix options 7-19
- set\_dft\_signal command 1-6, 1-11, 1-50, 1-54, 2-3, 2-13, 4-38, 4-39, 6-5, 6-31, 6-32, 6-33, 6-46, 6-47, 7-33, 8-13, 8-25, 8-28, 9-13
  - commands
    - set\_dft\_signal 7-19
- establishing signal type attributes 6-61
- inverted test mode 7-25
- protocol inference 6-66
- setting test mode 7-25
- syntax 6-26
- test mode signal 7-19
- set\_dont\_touch command 4-9
- set\_multibit\_options command 4-21
- set\_scan\_bidi command 6-5
- set\_scan\_command 6-35
- set\_scan\_configuration command 4-38, 5-32, 6-3, 6-14, 6-32, 7-33, 10-17
  - add\_lockup option 6-45
  - chain\_count option 6-10
  - clock\_mixing 8-13
  - clock\_mixing option 6-11, 6-49
  - configuring three-state buses 6-40
  - dedicated\_scan\_ports option 6-34
  - disable option 6-39
  - external\_tristates option 6-40
  - max\_chain\_length option 6-9
  - style option 4-15, 4-26
- set\_scan\_element command 4-38, 6-5
  - and dft\_drc 5-18
  - syntax 6-34
  - to prevent scan replacement 5-18
  - to suppress TEST-120 warning 5-17
- set\_scan\_path command 4-38, 6-5, 6-15, 6-46
  - complete option 6-16
  - syntax 6-8
- set\_scan\_register\_type command 4-18
- set\_scan\_replacement command 6-19
- set\_scan\_segment command 6-14
- set\_scan\_signal command 8-20
- set\_signal\_type command 6-22
- set\_test\_hold command 2-4
- set\_test\_point\_element command 7-31, 7-33
- setting
  - test mode signals 7-19
  - test point clocks 7-19

shared wrapper cells 8-10  
 sharing wrapper cells 8-10  
 shift data 9-3  
 shift procedure  
     STIL format 5-29  
 signal timing  
     STIL format 5-27  
 signal types  
     for AutoFix 7-19  
 signal\_type attribute 5-25  
 signal\_type attribute, identifying existing scan ports 6-22  
 signals  
     control 2-4  
     scan enable 2-4  
 Source signal, specifying 7-31  
 specifying  
     bidirectional delay 5-21  
     input delay 5-20  
     output strobe 5-22  
     scan architecture 6-3  
     scan configuration 6-3  
     test period 5-20  
 STIL (Standard Test Interface Language) 10-5  
 STIL format  
     initialization protocol 2-4  
 STIL formats  
     available 5-26  
     load\_unload procedure 5-29  
     shift procedure 5-29  
     signal timing 5-27  
     test\_setup macro 5-27  
 strobe 5-22  
 structured logic  
     defining 4-21  
     disabling 4-22  
 -style option, set\_scan\_configuration command 4-15  
 subdesign scan chain  
     including 6-23  
     inferring 6-23  
     summarizing violations 5-6  
     synthesis 4-25  
     synthesizable segment  
         defining 4-21  
         reporting 6-15  
     system clock 4-9

## T

target technology library 1-4, 1-10, 1-15  
 test clock 4-9  
 test clocks 2-3  
 test data volume, reducing 7-30  
 test design rules 5-9  
 test information, displaying 6-25  
 test mode 1-50  
     correcting invalid clock gating violations 6-67  
 test mode signal  
     inserted by AutoFix 7-16  
     set\_dft\_signal 7-19  
 test models 1-21  
     managing 1-28  
     saving 1-25  
     scan assembly using 1-24  
     using 1-26  
 test modes  
     inverted 7-25  
 test modes, test points and 7-19  
 test period  
     default 5-20  
     specifying 5-20  
 test point clocks 7-19  
 test point enable scan register (TPE) 7-28  
 Test Point Enable, specifying 7-32  
 test point insertion, design flow 7-14  
 test points  
     inserting 4-39  
     test modes and 7-19  
 test port  
     naming style variables 6-28

- selecting 6-31
- test ports, connecting 1-54
- test protocol 1-59
  - design characteristics 5-25
- test protocol, defined 1-43
- test signal type attributes 6-27
- test variables
  - test\_default\_bidir\_delay 5-21
  - test\_default\_delay 5-20
  - test\_default\_period 5-20
  - test\_default\_strobe 5-22
  - test\_default\_strobe\_width 5-22
- test wrapper 8-2
  - control interface 8-8
  - exceptions 8-18
  - modes 8-4
- test\_allow\_clock\_reconvergence variable 4-10
- test\_asynch attribute 2-14, 6-27
- test\_asynch\_inverted attribute 2-14, 6-27
- test\_bidir\_control attribute 6-27
- test\_bidir\_control\_inverted attribute 6-27
- test\_clock\_in\_port\_naming\_style variable 6-4
- test\_clock\_out\_port\_naming\_style variable 6-4
  - 6-4
- test\_clock\_port\_naming\_style variable 6-4, 6-28
- test\_control port 1-50
- test\_dedicated\_subdesign\_scan\_outs variable 6-34
- test\_default\_bidir\_delay variable 5-21, 6-60
  - definition 5-21
  - setting 5-21
- test\_default\_delay variable 5-20, 6-60
  - definition 5-20
  - setting 5-20
- test\_default\_period variable 5-20
  - definition 5-20
  - setting 5-20
- test\_default\_scan\_style variable 5-32
- test\_default\_strobe variable 5-22, 6-60
- definition 5-22
- setting 5-22
- test\_default\_strobe\_width variable
  - definition 5-22
  - setting 5-22
- test\_disable\_find\_best\_scan\_out variable 6-17
- test\_mode control input 7-25
- test\_mode port 1-54
- test\_mode signal 1-50, 1-52, 1-57
- test\_mux\_constant\_so variable 6-32
- test\_period variable 6-60
- test\_ready, setting scan state to 6-35
- test\_scan\_clock attribute 5-33
- test\_scan\_clock\_a attribute 5-33
- test\_scan\_clock\_a\_port\_naming\_style variable 6-4, 6-28
- test\_scan\_clock\_b attribute 5-33
- test\_scan\_clock\_b\_port\_naming\_style variable 6-4
- test\_scan\_clock\_port\_naming\_style variable 6-4, 6-28
- test\_scan\_enable attribute 6-46
- test\_scan\_enable\_inverted attribute 6-46
- test\_scan\_enable\_inverted\_port\_naming\_style variable 6-4, 6-28
- test\_scan\_enable\_port\_naming\_style variable 6-4, 6-28
- test\_scan\_in attribute 6-27, 6-46
- test\_scan\_in\_port\_naming\_style variable 6-28
- test\_scan\_out attribute 6-27, 6-46
- test\_scan\_out\_inverted attribute 6-46
- test\_scan\_out\_port\_naming\_style variable 6-28
- test\_setup macro 5-27
- TEST-114 6-61
- TEST-115 6-61
- TEST-116 message 4-12
- TEST-120 5-17
- TEST-120 message 4-6, 4-17

TEST-121 5-17  
TEST-121 message 4-9  
TEST-124 5-10  
TEST-169 message 4-11  
TEST-186 message 4-11  
TEST-202 5-17  
TEST-202 message 4-8  
TEST-224 5-17  
TEST-224 message 4-6, 4-16  
TEST-331 6-62  
TEST-342 message 6-16  
TEST-353 message 6-16  
TEST-376 message 6-17  
TEST-451 5-13, 5-16  
TEST-451 message 5-13  
TEST-460 to TEST-469 5-6  
TEST-462 5-15  
TEST-462 message 4-7  
TEST-463 5-15  
TEST-463 message 4-7  
TEST-464 5-15  
TEST-464 message 4-7  
TEST-465 5-15  
TEST-465 message 4-7  
TEST-466 5-16  
TEST-466 message 4-8  
TEST-467 5-16  
TEST-467 message 4-8  
TEST-468 5-16  
TEST-468 message 4-7  
TESTDB-256 message 6-16  
TestMode signal 7-27  
TestMode signal type 7-31  
Test-ready compile 1-5  
test-ready compile 4-23  
    benefits 4-5  
    bottom-up compile 4-30  
complex compiles 4-30  
controlling 4-26  
controlling scan implementation 4-26  
degeneration support 4-30  
example 4-29  
incremental compiles 4-30  
invoking 4-26  
when to use 4-4  
TetraMAX ATPG  
    before exporting your design 10-4  
    differences from DFT Compiler 10-5  
    exporting your design 10-14  
    netlist format 10-15  
    supported DFT Compiler commands 10-5  
three-state bus  
    configuring external 6-40  
    configuring internal 6-41  
    disabling 6-41  
three-state nets  
    default insert\_scan behavior and 6-39  
    design strategy, default 6-39  
    inserting disabling logic  
        limitations 6-38  
        process 6-38  
    preventing  
        bus contention 6-38  
        insertion of disabling logic 6-39  
timing  
    attributes  
        effect on vector formatting 5-24  
top-down  
    AutoFix design example 7-22  
top-down design flow, comparing results with  
    bottom-up 6-17  
top-down design flow, overview 1-43  
tpclk clock port 7-31  
tristate ports, wrapping 8-17, 8-20  
tri-state test points 7-27  
type of control point, specifying 7-31  
types of User-Defined Test Points 7-26

## U

UDL 8-2  
UDTP Example 7-32  
UIT-227 message 6-45  
UIT-229 message 6-49  
uncontrollable asynchronous pin 4-11  
    correcting 4-12  
    fault coverage impact 4-11  
    unreliable data capture 6-72  
uncontrollable clock 4-11  
uncontrollable clocks 2-13  
uncontrollable registers 2-14  
uncontrollable scan cell (*see* scan cell,  
    uncontrollable) 6-63  
unreliable capture 2-16  
unresolved reference 4-31  
unrouted scan design, after test-ready compile  
    4-25  
unsupported cells 5-16  
untestable functional path 6-70  
    example 6-72  
user defined logic. *See* UDL  
user-defined wrapper chains 8-22

## V

validating  
    scan ordering 6-16  
variables  
    environment  
        test\_clock\_in\_port\_naming\_style 6-4  
        test\_clock\_out\_port\_naming\_style 6-4  
        test\_clock\_port\_naming\_style 6-4, 6-28  
        test\_scan\_clock\_a\_port\_naming\_style  
            6-4, 6-28  
        test\_scan\_clock\_b\_port\_naming\_style 6-4  
        test\_scan\_clock\_port\_naming\_style 6-4,  
            6-28  
        test\_scan\_enable\_inverted\_port\_naming\_  
            style 6-4, 6-28

test\_scan\_enable\_port\_naming\_style 6-4,  
    6-28  
test\_scan\_in\_port\_naming\_style 6-28  
test\_scan\_out\_port\_naming\_style 6-28  
link\_library 4-31  
test  
    test\_default\_bidir\_delay 5-21  
    test\_default\_delay 5-20  
    test\_default\_period 5-20  
    test\_default\_strobe 5-22  
    test\_default\_strobe\_width 5-22  
test\_allow\_clock\_reconvergence 4-10  
test\_dedicated\_subdesign\_scan\_outs 6-34  
test\_default\_bidir\_delay 5-21, 6-60  
test\_default\_delay 5-20, 6-60  
test\_default\_period 5-20  
test\_default\_scan\_style 5-32  
test\_default\_strobe 5-22, 6-60  
test\_disable\_find\_best\_scan\_out 6-17  
test\_mux\_constant\_so 6-32  
test\_period 6-60  
vector  
    formatting, timing attributes 5-24  
vendor requirements, meeting 6-42  
-verbose option, dft\_drc command 5-9, 5-10  
Verilog 7-34  
violations  
    asynchronous control pins 2-14  
    black box 2-16  
    black boxes 2-22  
    clock feeding multiple register inputs 2-18  
    clock used as data 2-15  
    clocks interacting with register input 2-20  
    combinational feedback loops 2-19  
    latch requiring multiple clocks 2-21  
    launch before capture 2-16  
modeling  
    black boxes 5-13  
    generic cells 5-16  
    latches 5-18  
    no scan cell equivalents 5-17

scan cell equivalents and dont\_touch  
attribute 5-17  
unsupported cells 5-14  
multiple clocks 2-21  
preventing data capture 2-15  
preventing scan insertion 2-13, 5-36, 5-38,  
5-39  
reducing fault coverage 2-19  
registered clock gating circuitry 2-17  
scan connectivity 6-62  
summarizing 5-6  
topological  
nets and drivers 6-61  
uncontrollable clocks 2-13  
violations, fixing with AutoFix 7-16

## W

WC\_D1 wrapper cell 8-9  
WC\_S1 wrapper cell 8-11  
wrapper cell 8-9  
wrapper cell interface 8-9  
wrapper cell order, specifying 8-20  
wrapper cells 8-2

all-input 8-21  
all-output 8-21  
shared 8-10  
sharing 8-10  
tristate and bidirectional ports 8-20  
WC\_S1 8-11  
wrapper chain ports, specifying 8-20  
wrapper chains, specifying 8-19, 8-21  
wrapper configuration 8-24  
wrapper control signals 8-8  
wrapper exceptions 8-18  
wrapper modes 8-4  
wrapping  
all-input cell chains 8-21  
all-output cell chains 8-21  
bidirectional ports 8-17  
excluding ports 8-18  
tristate ports 8-17  
write\_test\_model command 1-25, 1-28  
write\_test\_protocol command 5-34

## X

XOR tree 7-30