

Synopsys® TestMAX™ DFT User Guide

Version R-2020.09, September 2020

SYNOPSYS®

Copyright and Proprietary Information Notice

© 2020 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>.

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

Free and Open-Source Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

New in This Release	35
Related Products, Publications, and Trademarks	36
Conventions	37
Customer Support	38

Part 1: DFT Overview

1. Introduction to Synopsys DFT Tools	40
Key Features	40
Key Benefits	41
DFT Compiler and the Synopsys TestMAX Product Platform	41
DFTMAX Scan Compression	43
DFTMAX Ultra Scan Compression	43
DFTMAX LogicBIST Self-Test	44
Other Tools in the Synopsys Test and Yield Solution	44
 2. Designing for Manufacturing Test	 46
Functional Testing Versus Manufacturing Testing	46
Modeling Manufacturing Defects	47
Understanding Stuck-At Fault Models	47
Controllable and Observable Faults	48
Detecting Stuck-At Faults	48
Determining Coverage	50
Understanding Fault Simulation	50
Automatically Generating Test Patterns	51
Formatting Test Patterns	52
Achieving Maximum Fault Coverage for Sequential Cells	52
Controllability of Sequential Cells	52
Observability of Sequential Cells	52

Understanding the Full-Scan Test Methodology	53
Scan Styles Supported by DFT Compiler	54
Multiplexed Flip-Flop Scan Style	54
Clocked-Scan Scan Style	55
Level-Sensitive Scan Design (LSSD) Style	55
Scan-Enabled Level-Sensitive Scan Design (LSSD) Style	56
Summary of Supported Scan Cells	57
Logic Library Considerations	57
Describing the Test Environment	58
Test Design Rule Checking Functions	58
Getting the Best Results With Scan Design	58
3. Scan Design Techniques	61
Internal Scan Design	61
Scan Cells	61
Scan Chains	62
Scan Cells in Semiconductor Vendor Libraries	62
The Effect of Adding Scan Circuitry to a Design	62
ATPG and Internal Scan	63
Applying Scan Patterns	63
Full-Scan Design	65
Test for System-On-A-Chip Designs	66
Boundary Scan Design	66
4. Scan Styles	68
Multiplexed Flip-Flop Scan Style	68
Flip-Flop Equivalents	68
Master-Slave Latch Equivalents	69
Multiplexed Flip-Flop Scan Style Characteristics	71
Clocked-Scan Scan Style	72
Flip-Flop Equivalents	72
Latch Equivalents	74
Clocked-Scan Scan Style Characteristics	76
LSSD Scan Style	76
Single-Latch LSSD	77

Single-Latch LSSD Scan Style Characteristics	79
Double-Latch LSSD	80
Double-Latch LSSD Scan Style Characteristics	82
Clocked LSSD	82
Clocked LSSD Scan Style Characteristics	84
Scan-Enabled LSSD Style	85
Scan-Enabled LSSD Scan Style Characteristics	87
5. Scan Design Requirements	88
Test Port Requirements	88
Test Timing Requirements	90
Test Clock Requirements	91
Clock Requirements in Edge-Sensitive Scan Shift Styles	91
Skew Issues	91
Mixed Edges	92
Multiple Clocks	93
Clock Requirements in LSSD Scan Styles	94
Master Scan Clock and Slave Clock	94
Synchronized Clocks	94
Skew Control	94
Test Protocol Requirements	94
Valid and Invalid Test Protocols	95
Methods of Generating Test Protocols	95
Reading In an Existing Test Protocol	95
Creating a Fully User-Specified Test Protocol	96
Inferring a Test Protocol Based on Partial Specification	96
Inferring a Test Protocol	96
Initialization Protocol	96
Protocol Types	96
Strobe-Before-Clock Protocol	97
A Strobe-Before-Clock Example	97
Strobe-After-Clock Protocol	99
A Strobe-After-Clock Example	99
Part 2: DFT Compiler Scan	
6. Getting Started	103

Preparing to Run DFT Compiler	105
Invoking the Synthesis Tool	105
Setting Up Your Design Environment	106
Reading In Your Design	107
Setting the Scan Style	107
Configuring the Test Cycle Timing	108
Defining the DFT Signals	109
Performing Scan Synthesis	110
Performing One-Pass Scan Synthesis	112
Performing Scan Insertion	112
Configuring Scan Insertion	113
Performing Pre-DFT Test DRC	114
Previewing Scan Insertion	116
Inserting the DFT Logic	117
Performing Post-DFT Optimization	117
Analyzing Your Post-DFT Design	118
Reporting	119
Designing Block by Block	122
Performing Scan Extraction	123
Hierarchical Scan Synthesis	124
Top-Down Flat Versus Bottom-Up Hierarchical	124
Introduction to Test Models	126
Writing Out a CTL Model at the Core Level	127
Reading In and Using CTL Models at the Top Level	129
Checking Connectivity to Cores at the Top Level	131
Using Advanced Clock Feedthrough Analysis	132
Connecting the Scan-Enable Pins of Cores	132
Hierarchical Synthesis, DFT Insertion, and Layout Flows	133
Linking Test Models to Library Cells	135
Checking Library Cells for CTL Model Information	136
Physical DFT Features in Design Compiler	137
DFT Flows in DC Explorer	138
<hr/>	
7. Running the Test DRC Debugger	140
Starting and Exiting the Graphical User Interface	140
Exploring the Graphical User Interface	141

Logic Hierarchy View	142
Console	143
Command Line	143
Viewing Man Pages	143
Menus	144
Checking Scan Test Design Rules	144
Examining DRC Violations	144
Viewing Test Protocols	144
Viewing Design Violations	145
Reporting DRC Violations	145
Inspecting DRC Violations	147
Viewing a Violation	147
Viewing Multiple Violations Together	149
Viewing CTL Model Scan Chain Information	150
Viewing test_setup Pin Data Waveforms	151
Commands Specific to the DFT Tools in the GUI	153
gui_inspectViolations	153
guiWaveAddSignal	154
guiViolationSchematicAddObjects	154
8. Performing Scan Replacement	156
Scan Replacement Flow	156
Preparing for Scan Replacement	158
Selecting a Scan Replacement Strategy	158
Identifying Barriers to Scan Replacement	159
Logic Library Does Not Contain Appropriate Scan Cells	160
Support for Different Types of Sequential Cells and Violations	160
Attributes That Can Prevent Scan Replacement	162
Invalid Clock Nets	163
Invalid Asynchronous Pins	166
Preventing Scan Replacement	166
Specifying a Scan Style	167
Types of Scan Styles	167
Multiplexed Flip-Flop Scan Style	168
Clocked Scan Style	168
LSSD Scan Style	168
Scan-Enabled LSSD Scan Style	169
Scan Style Considerations	169
Setting the Scan Style	170

Verifying Scan Equivalents in the Logic Library	170
Checking the Logic Library for Scan Cells	171
Checking for Scan Equivalents	172
Scan Cell Replacement Strategies	172
Specifying Scan Cells	173
Restricting the List of Available Scan Cells	173
Scan Cell Replacement Strategies	174
Mapping Sequential Gates in Scan Replacement	174
Multibit Components	175
What Are Multibit Components?	176
How DFT Compiler Creates Multibit Components	176
Controlling Multibit Test Synthesis	177
Performing Multibit Component Scan Replacement	177
Disabling Multibit Component Support	178
Test-Ready Compilation	178
What Is Test-Ready Compile?	178
The Test-Ready Compile Flow	179
Preparing for Test-Ready Compile	180
Performing Test-Ready Compile in the Logic Domain	181
Controlling Test-Ready Compile	181
Comparing Default Compile and Test-Ready Compile	182
Complex Compile Strategies	185
Validating Your Netlist	186
Running the link Command	186
Running the check_design Command	187
Performing Constraint-Optimized Scan Insertion	187
Supported Scan States	188
Locating Scan Equivalents	188
Preparing for Constraint-Optimized Scan Insertion	190
Scan Insertion	191
Specification Phase	193
Preview	194
Synthesis	195
<hr/>	
9. Architecting Your Test Design	196
Configuring Your DFT Architecture	197
Defining Your Scan Architecture	197
Setting Design Constraints	198

Defining Constant Input Ports During Scan	199
Specifying Test Ports	199
Specifying Individual Scan Paths	199
Architecting Scan Chains	201
Controlling the Scan Chain Length	202
Specifying the Global Scan Chain Length Limit	202
Specifying the Global Scan Chain Exact Length	202
Determining the Scan Chain Count	203
Defining Individual Scan Chain Characteristics	203
Balancing Scan Chains	204
Concatenating Scan Cells and Segments	205
Multiple Clock Domains	205
Multibit Components and Scan Chains	208
Physical Reordering and Repartitioning	209
Controlling the Routing Order	210
Retiming Scan-Ins and Scan-Outs to the Leading Clock Edge	211
Routing Scan Chains and Global Signals	213
Rerouting Scan Chains	213
Stitching Scan Chains Without Optimization	214
Specifying a Stitch-Only Design	214
Mapping the Replacement of Nonscan Cells to Scan Cells	214
Criteria for Conversion Between Nonscan and Scan Cells	216
Scan Stitching Only Scan-Replaced Cells	217
Using Existing Subdesign Scan Chains	218
Uniquifying Your Design	220
Reporting Scan Path Information on the Current Design	221
Architecting Scan Signals	221
Specifying Scan Signals for the Current Design	222
Selecting Test Ports	227
Defining Existing Unconnected Ports as Scan Ports	227
Sharing a Scan Input With a Functional Port	228
Sharing a Scan Output With a Functional Port	228
Controlling Subdesign Scan Output Ports	229
Controlling Scan-Enable Connections to DFT Logic	230
Associating Scan-Enable Ports With Specific Scan Chains	230
Defining Dedicated Scan-Enable Signals for Scan Cells	230
Connecting the Scan-Enable Signal in Hierarchical Flows	232
Preserving Existing Scan-Enable Pin Connections	234
Controlling Buffering for DFT Signals	235
Suppressing Replacement of Sequential Cells	235

In Logic Scan Synthesis	236
Changing the Scan State of a Design	236
Removing Scan Configurations	237
Keeping Specifications Consistent	237
Synthesizing Three-State Disabling Logic	238
Configuring Three-State Buses	241
Configuring External Three-State Buses	241
Configuring Internal Three-State Buses	241
Overriding Global Three-State Bus Configuration Settings	242
Disabling Three-State Buses and Bidirectional Ports	242
Handling Bidirectional Ports	243
Setting Individual Bidirectional Port Behavior	243
Fixed Direction Bidirectional Ports	243
Assigning Test Port Attributes	244
Architecting Test Clocks	244
Defining Test Clocks	245
Specifying a Hookup Pin for DFT-Inserted Clock Connections	246
Requirements for Valid Scan Chain Ordering	247
Lock-Up Latch Insertion Between Clock Domains	248
Automatically Creating Skew Subdomains Within Clock Domains	253
Manually Creating Skew Subdomains at Associated Internal Pins	256
Manually Creating Skew Subdomains With Scan Skew Groups	258
Defining Scan Chains by Scan Clock	260
Handling Multiple Clocks in LSSD Scan Styles	261
Using Multiple Master Clocks	261
Dedicated Test Clocks for Each Clock Domain	262
Controlling LSSD Slave Clock Routing	263
Configuring Clock-Gating Cells	265
Introduction to Clock Gating in DFT Flows	266
Clock-Gating Control Points	266
Configuring Clock-Gating Control Points	267
Scan-Enable Signal Versus Test-Mode Control Signal	268
Improving Observability When Using Test-Mode Control Signals	269
Discrete-Logic Clock-Gating Cells and Integrated Clock-Gating Cells	271
Inferred and Instantiated Clock-Gating Cells	272
Inferring Clock-Gating Cells Using Power Compiler	272
Instantiating Clock-Gating Cells in the RTL	272
Choosing a Clock-Gating Control Point Configuration	274
Initialization for Special Cases of Before-Latch Control Points	275
Reporting Unconnected Clock-Gating Cell Test Pins During Pre-DFT DRC	276

Automatically Connecting Test Pins During DFT Insertion	277
Specifying Signals for Clock-Gating Cell Test Pin Connections	278
Identifying Clock-Gating Cells in an ASCII Netlist Flow	279
Limitations	280
Specifying a Location for DFT Logic Insertion	280
Creating New DFT Logic Blocks	284
Partitioning a Scan Design With DFT Partitions	285
Defining DFT Partitions	286
Configuring DFT Partitions	287
Per-Partition Scan Configuration Commands	289
set_scan_configuration	289
set_dft_signal	290
set_dft_location	290
set_scan_path	290
set_testability_configuration	291
set_wrapper_configuration	291
Known Issues of the DFT Partition Flow	291
Modifying Your Scan Architecture	292
10. Advanced DFT Architecture Methodologies	293
Inserting Test Points	293
Test Point Types	294
Force Test Points	294
Control Test Points	296
Observe Test Points	298
Multicycle Test Points	298
Test Point Structures	299
Test Point Components	299
Test Point Register Clocks	300
Test Point Enable Logic	301
Sharing Test Point Registers	303
Automatically Inserted Test Points	305
Enabling Automatic Test Point Insertion	307
Configuring Global Test Point Insertion Settings	307
Configuring the Random-Resistant Test Point Target	309
Configuring the Untestable Logic Test Point Target	311
Configuring the X-Blocking Test Point Target	313
Configuring the Multicycle Path Test Point Target	314
Configuring the Shadow Wrapper Test Point Target	315
Configuring the Core Wrapper Test Point Target	316

Configuring the XOR Self-Gating Test Point Target	318
Configuring the User-Defined Test Point Target	319
Enabling Multiple Targets in a Single Command	320
Implementing Test Points From an External File	321
Customizing the Test Point Analysis	324
Running Test Point Analysis	325
Automatic Test Point Insertion Example Script	326
Limitations	327
User-Defined Test Points	327
Enabling User-Defined Test Point Insertion	328
Configuring User-Defined Test Points	328
Limitations	330
Previewing the Test Point Logic	330
Inserting the Test Point Logic	332
Using AutoFix	333
When to Use AutoFix	333
Uncontrollable Clock Signals	334
Uncontrollable Asynchronous Set and Reset Signals	334
Uncontrollable Three-State Bus Enable Signals	335
Uncontrollable Bidirectional Enable Signals	336
The AutoFix Flow	337
Configuring AutoFix	339
Enabling AutoFix Capabilities	339
Configuring Clock AutoFixing	340
Configuring Set and Reset AutoFixing	340
Configuring Three-State Bus AutoFixing	342
Configuring Bidirectional AutoFixing	342
Applying Hierarchical AutoFix Specifications	343
Previewing the AutoFix Implementation	345
AutoFix Script Example	345
Using Pipelined Scan Enables for Launch-On-Extra-Shift (LOES)	346
The Pipelined Scan-Enable Architecture	346
Pipelined Scan-Enable Requirements	349
Implementing Pipelined Scan-Enable Signals	350
Pipelined Scan-Enable Signals in Hierarchical Flows	351
Implementation Considerations for Pipelined Scan-Enable Signals	353
Pipelined Scan Enable Limitations	356
Excluding Elements from a Pipelined Scan-Enable Configuration	356
Multiple Test Modes	357
Introduction to Multiple Test Modes	357
Defining Test Modes	358

Defining the Usage of a Test Mode	359
Defining the Encoding of a Test Mode	360
Applying Test Specifications to a Test Mode	362
Recommended Ordering of Global and Mode-Specific Commands	365
Using Multiple Test Modes in Hierarchical Flows	365
Supported Test Specification Commands for Test Modes	367
set_dft_signal	367
set_scan_configuration	368
set_scan_path	368
Multiple Test-Mode Scan Insertion Script Examples	369
Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces	377
IEEE 1500 Test Mode Control Architecture	377
Core-Level Test-Mode Control	378
Core Integration With IEEE 1500 Test-Mode Control	380
Chip-Level Test-Mode Control	381
Inserting IEEE 1500 at the Core Level	382
Inserting IEEE 1500 and IEEE 1149.1 at the Chip Level	383
Customizing the IEEE 1500 Architecture	384
Configuring the WIR	384
Configuring the DFT-Inserted TMCDR	385
Using an Existing TMCDR	385
Using WIR Test-Mode Decoding With No TMCDR	386
Controlling the Test-Mode Encoding Style	387
Reporting the Test Mode Encodings	387
Specifying WIR Opcodes for CDRs	388
Writing Test Protocols	388
Script Examples	390
Limitations	392
Multivoltage Support	393
Configuring Scan Insertion for Multivoltage Designs	393
Configuring Scan Insertion for Multiple Power Domains	394
Mixture of Multivoltage and Multiple Power Domain Specifications	394
Reusing Multivoltage Cells	395
Reusing Level Shifters in Scan Paths	396
Reusing Isolation Cells in Scan Paths	397
Scan Path Routing and Isolation Strategy Requirements	403
Using Domain-Based Strategies for DFT Insertion	406
DFT Considerations for Low-Power Design Flows	407
Previewing a Multivoltage Scan Chain	409
Scan Extraction Flows in the Presence of Isolation Cells	410

Limitations	411
Controlling Power Modes During Test	411
Inserting Power Controller Override Logic	411
Limitations	414
Reducing Shift Power Using Functional Output Pin Gating	414
Controlling Clock-Gating Cell Test Pin Connections	420
Connecting User-Instantiated Clock-Gating Cells	421
Script Example	423
Limitations	424
Excluding Clock-Gating Cells From Test-Pin Connection	424
Connecting Clock-Gating Cell Test Pins Without Scan Stitching	427
Internal Pins Flow	429
Defining Signals on Internal Pins	430
Writing Out the Test Protocol	431
Limitations of the Internal Pins Flow	431
Creating Scan Groups	432
Configuring Scan Grouping	432
Creating Scan Groups	432
Removing Scan Groups	433
Integrating an Existing Scan Chain Into a Scan Group	434
Reporting Scan Groups	435
Scan Group Flows	435
Known Limitations	436
Shift Register Identification	436
Simple Shift Register Identification	436
Synchronous-Logic Shift Register Identification	437
Shift Register Identification in an ASCII Netlist Flow	438
Performing Scan Extraction	439
11. Wrapping Cores	441
Core Wrapping Concepts	441
Wrapper Cells and Wrapper Chains	442
Wrapper Test Modes	445
The Simple Core Wrapping Flow	446
Simple Core Wrapper Cells	446
Simple Core Wrapper Chains	451
The Maximized Reuse Core Wrapping Flow	452

Maximized Reuse Core Wrapper Cells	452
Maximized Reuse Core Wrapper Chains	454
Maximized Reuse Shift Signals	456
Wrapping Three-State and Bidirectional Ports	456
Wrapping a Core	458
Enabling Core Wrapping	458
Defining Wrapper Shift Signals	459
Defining Dedicated Wrapper Clock Signals	460
Configuring Global Wrapper Settings	461
Configuring Port-Specific Wrapper Settings	461
Controlling Wrapper Chain Count and Length	462
Configuring Simple Core Wrapping	464
Configuring Dedicated Wrapper Cell Clocks	464
Using Shared Wrapper Cells	464
Configuring Shared Wrapper Cell Clocks	465
Using In-Place Shared Wrapper Cells	465
Creating Separate Input and Output Wrapper Chains	466
Configuring Maximized Reuse Core Wrapping	467
Enabling Maximized Reuse Core Wrapping	467
Applying a Register Reuse Threshold	468
Applying a Combinational Depth Threshold	471
Specifying Port-Specific Maximized Reuse Behaviors	472
Special Cases for Register Reuse	473
Using Dedicated Wrapper Cells	476
Configuring Dedicated Wrapper Cell Clocks	476
Defining Input/Output Clock-Domain-Based Wrapper Shift Signals	477
Including Additional Scan Cells in Input and Output Wrapper Chains	477
Using the Pipelined Scan-Enable Feature	478
Low-Power Maximized Reuse Features	479
Hierarchical Core Wrapping	482
Limitations of the Maximized Reuse Flow	484
Determining Power Domains for Dedicated Wrapper Cells	485
Using the set_scan_path Command With Wrapper Chains	485
Previewing the Wrapper Cells	487
Previewing Maximized Reuse Wrapper Cells	488
Post-DFT DRC Rule Checks	491
Creating User-Defined Core Wrapping Test Modes	491
Creating Compressed EXTEST Core Wrapping Test Modes	492
Creating an IEEE 1500 Wrapped Core	494
Wrapping Cores With OCC Controllers	495

Wrapping Cores With OCC Clock Outputs	495
Wrapping Cores With DFT Partitions	497
Wrapping Cores With Multibit Registers	497
Wrapping Cores With Synchronizer Registers	499
Wrapping Cores With Existing Scan Chains	500
Creating an EXTEST-Only Core Netlist	504
Integrating Wrapped Cores in Hierarchical Flows	505
Scheduling Wrapped Cores	505
Integrating Wrapped Cores in a Compressed Scan Flow	508
Nested Integration of Wrapped Cores	510
Mixing Wrapped and Unwrapped Cores	511
Top-Down Flat Testing With Transparent Wrapped Cores	511
Introduction to Transparent Test Modes	512
Defining Core-Level Transparent Test Modes	513
Defining Top-Level Flat Test Modes	514
Limitations	515
SCANDEF Generation for Wrapper Chains	515
Core Wrapping Scripts	516
Core Wrapping With Dedicated Wrapper Cells	516
Core Wrapping With Maximized Reuse	517
12. On-Chip Clocking Support	519
Background	520
Supported DFT Flows	521
Clock Type Definitions	521
Capabilities	522
OCC Controller Structure and Operation	523
DFT-Inserted and User-Defined OCC Controllers	523
Synchronous and Asynchronous OCC Controllers	526
OCC Controller Signal Operation	527
Clock Chain Operation	528
Logic Representation of an OCC Controller and Clock Chain	529
Scan-Enable Signal Requirements for OCC Controller Operation	530
Enabling On-Chip Clocking Support	530
Specifying OCC Controllers	531

Specifying DFT-Inserted OCC Controllers	531
Defining Clocks	531
Defining Global Signals	534
Configuring the OCC Controller	535
Configuring the Clock Selection Logic	537
Configuring the Clock-Chain Clock Connection	540
Specifying Scan Configuration	540
Performing Timing Analysis	541
Script Example	541
Specifying Existing User-Defined OCC Controllers	542
Defining Clocks	543
Defining Global Signals	546
Specifying Clock Chains	547
Scan Configuration for User-Defined OCC Controllers	548
Script Example	548
Specifying OCC Controllers for External Clock Sources	550
Using OCC Controllers in Hierarchical DFT Flows	551
Integrating Cores That Contain OCC Controllers	551
Defining Signals for Cores Without Preconnected OCC Signals	552
Defining Signals for Cores With Preconnected OCC Signals	553
Handling Cores With OCC Clock Output Pins	554
Reporting Clock Controller Information	554
DFT-Inserted OCC Controller Flow	554
Existing User-Defined OCC Controller Flow	555
DRC Support	556
Enabling the OCC Controller Bypass Configuration	556
DFT-Inserted OCC Controller Configurations	557
Single OCC Controller Configurations	557
Example 1	557
Example 2	558
Example 3	558
Multiple DFT-Inserted OCC Controller Configurations	559
Example 1	560
Example 2	560
Waveform and Capture Cycle Example	561
Limitations	562
<hr/>	
13. Pre-DFT Test Design Rule Checking	565
Test DRC Basics	565

Test DRC Flow	565
Preparing Your Design	567
Creating the Test Protocol	568
Assigning a Known Logic State	568
Performing Test Design Rule Checking	568
Reporting All Violating Instances	569
Analyzing and Debugging Violations	569
Summary of Violations	569
Enhanced Reporting Capability	570
Test Design Rule Checking Messages	572
Understanding Test Design Rule Checking Messages	572
Effects of Violations on Scan Replacement	572
Viewing the Sequential Cell Summary	573
Classifying Sequential Cells	573
Sequential Cells With Violations	574
Cells With Scan Shift Violations	574
Black-Box Cells	575
Constant Value Cells	575
Sequential Cells Without Violations	575
Checking for Modeling Violations	575
Black-Box Cells	575
Correcting Black-Box Cells	576
Unsupported Cells	577
Generic Cells	579
Scan Cell Equivalents	579
Scan Cell Equivalents and the dont_touch Attribute	580
Latches	580
Nonscan Latches	580
Setting Test Timing Variables	581
Protocols for Common Design Timing Requirements	581
Preclock Measure Protocol	582
End-of-Cycle Measure Protocol	582
Setting Timing Variables	582
test_default_period Variable	583
test_default_delay Variable	583
test_default_bidir_delay Variable	583
test_default_strobe Variable	584
test_default_strobe_width Variable	585
The Effect of Timing Variables on Vector Formatting	586
Creating Test Protocols	587

Design Characteristics for Test Protocols	587
Scan Style	588
New DFT Signals	588
Existing Clock Ports	588
Existing Asynchronous Control Ports	588
Bidirectional Ports	589
STIL Test Protocol File Syntax	589
Defining the test_setup Macro	589
Defining Basic Signal Timing	590
Defining the load_unload Procedure	592
Defining the Shift Procedure	592
Defining an Initialization Protocol	592
Scan Shift and Parallel Measure Cycles	595
Multiplexed Flip-Flop Scan Style	595
Clocked-Scan Scan Style	596
LSSD Scan Style	596
Scan-Enabled LSSD Scan Style	596
Examining a Test Protocol File	597
Updating a Protocol in a Scan Chain Inference Flow	600
Masking Capture DRC Violations	600
Configuring Capture DRC Violation Masking	600
Reporting Capture DRC Violation Masking	601
Resetting Capture DRC Violation Masking	602
14. Previewing, Inserting, and Checking DFT Logic	603
Previewing the DFT Logic	603
Running the preview_dft Command	604
Previewing Additional Scan Chain Information	604
Previewing Test Mode Information	608
Previewing the DFT Design Using Script Commands	609
Inserting the DFT Logic	610
Scan Replacement	610
Scan Element Allocation and Ordering	611
Test Signals	611
Pad Cells	612
Post-DFT Insertion Test Design Rule Checking	612
Running Post-DFT DRC After DFT Insertion	613
Checking for Topological Violations	613
Checking for Scan Connectivity Violations	614

Scan Chain Extraction	615
Causes of Common Violations	615
Ability to Load Data Into Scan Cells	615
Incomplete Test Configuration	616
Invalid Clock Logic	617
Incorrect Clock Timing Relationship	620
Nonscan Sequential Cells	622
Ability to Capture Data Into Scan Cells	623
Clock Driving Data	624
Untestable Functional Path	625
Uncontrollable Asynchronous Pins	626
Post-DFT DRC Limitations	627
15. Exporting Data to Other Tools	628
Exporting a Design to TestMAX ATPG	628
Introduction to STIL Protocol Files	629
Exporting Your Design to TestMAX ATPG	630
Adjusting WaveformTable Timing for Delay Test	632
Reading Designs With Black-Box Test Models Into TestMAX ATPG	633
STIL Protocol File Procedure and WaveformTable Examples	633
Limitations	635
Using The SCANDEF-Based Reordering Flow	635
Introduction to SCANDEF	635
SCANDEF Constructs	636
Generating SCANDEF Information	638
Writing Out the SCANDEF Information	638
Script Example	639
Generating SCANDEF Information in Hierarchical DFT Flows	640
Preventing Scan Optimization in a Core	640
Allowing Scan Optimization in a Core	641
Using SCANDEF Information in a Manual Core Integration Flow	643
SCANDEF Examples	643
Default (Two Scan Chains)	643
Mixed Clock Edges	644
set_scan_path With No Elements	644
set_scan_path With Unordered Elements	645
set_scan_path With Ordered Elements	645
Scan Elements That Cannot Be Reordered or Repartitioned	646
Unrouted Scan Groups	646
Serial-Routed Scan Groups	647

CTL-Modeled Core	647
Inferred Shift Register	647
PARTITION Name Conventions	648
Support for Other DFT Features	649
Limitations of SCANDEF Generation	650
Verifying DFT Inserted Designs for Functionality	650
Verification Setup File Generation	651
Test Information Passed to the Verification Setup File	652
Script Example	652
Formality Tool Limitations	653

Part 3: DFTMAX Compression

16. Introduction to DFTMAX	655
The DFTMAX Compression Architecture	655
The DFTMAX Codec	655
Decompressor Operation	657
Compressor Operation	657
The Congestion-Aware DFTMAX Codec	658
DFTMAX Compression Requirements	658
Design Requirements	659
Pin Requirements	659
Multicore Processing	660
License Usage	660
Limitations	662
Current Limitations	662
DFTMAX Compression Limitations	662
17. Using DFTMAX Compression	663
Top-Down Flat Compressed Scan Flow	663
Top-Down Flat Compressed Scan Flow With DFT Partitions	668
When to Use DFT Partitions in a Scan Compression Flow	668
Configuring Partition Codecs	669
Choosing a Partitioned Codec Insertion Method	670
Per-Partition Scan Compression Configuration Commands	672

set_scan_compression_configuration	673
set_dft_location	673
set_dft_signal	673
Limitations of DFT Partitions in Scan Compression Flow	674
DFT Partition Script Example	674
DFTMAX Scan Compression and Multiple Test Modes	675
Defining Multiple Compressed Scan Modes	676
Per-Test-Mode Scan Compression Configuration Commands	678
set_scan_compression_configuration	678
set_scan_path	679
Multiple Test-Mode Script Examples	679
Multiple Standard Scan Modes and One Compressed Scan Mode	680
Multiple Standard Scan and Compressed Scan Modes	682
Standard Scan Flow Using Multiple Test Modes and Partitions	684
Scan Compression Flow Using Multiple Test Modes and Partitions	685
Excluding Scan Chains From Scan Compression	686
Scan Compression and OCC Controllers	687
Using Compressed Clock Chains	688
Defining External Clock Chains	689
Specifying a Different Scan Pin Count for Compressed Scan Mode	691
Adding Compressed Chain Lock-Up Latches	693
Reducing Power Consumption in DFTMAX Designs	694
Reducing Compressor Power When Codec Is Inactive	694
Preserving Compressor Gating Cells During Optimization	696
Reducing Scan Shift Power Using Shift Power Groups	696
The Shift Power Groups Architecture	696
Scan-Enable Signal Requirements for Shift Power Groups	698
Configuring Shift Power Groups	698
Integrating Cores With Shift Power Groups in Hierarchical Flows	699
Configuring Shift Power Groups in TestMAX ATPG	701
Using Shift Power Groups With Other DFT Features	702
Limitations of Shift Power Groups	703
Forcing a Compressor With Full Diagnostic Capabilities	704
Performing Congestion Optimization on Compressed Scan Designs	706
Using AutoFix With Scan Compression	706
One-Pass DFTMAX Example With AutoFix	707
One-Pass DFTMAX Example With AutoFix and Multiple Test Modes	708

18. Hierarchical Adaptive Scan Synthesis	712
The HSS Flow	712
The HASS Flow	714
Preparing Cores in the HASS Flow	714
HASS Integration of Compressed Scan Cores	715
HASS Integration of Additional Uncompressed Scan Logic	717
The Hybrid Flow	720
Performing Top-Level Hybrid Integration	721
Performing Top-Level Hybrid Integration with Partitions	723
Using Multiple Test Modes in Hierarchical Flows	725
Default Core-Level Test Mode Assignment	726
User-Defined Core-Level Test Mode Scheduling	728
Top-Level Integration Script Examples	730
Typical HASS Flow Script	731
Typical Hybrid Flow Script	731
Hybrid Flow Script With Multiple Test Modes	732
HASS and Hybrid Flow Limitations	733
19. Managing X Values in Scan Compression	735
High X-Tolerance Scan Compression	735
The High X-Tolerance Architecture	735
Enabling High X-Tolerance	737
Scan-In and Scan-Out Requirements	737
Limitations	739
Static-X Analysis	739
Architecting X Chains	741
The X-Chain Architecture	741
Enabling X Chains	742
Manually Specifying X-Chain Cells	743
Using the <code>set_scan_path</code> Command With X Chains	744
Using AutoFix With X Chains	744
Using X Chains in Hierarchical Flows	746
Static-X Cells in the HASS Flow	746
Hierarchical Blocks and X Sources	748
Using the <code>test_simulation_library</code> Variable	749

Representing X Chains in SCANDEF Files	750
Passing X-Chain Information to TestMAX ATPG	751
Error and Warning Summaries	752
X-Chain Usage Guidance	753
<hr/>	
20. Advanced DFTMAX Compression	754
Specifying a Location for Codec Logic Insertion	754
Pipelined Scan Data	755
Introduction to Pipelined Scan Data	756
Using Pipelined Scan Data	757
Enabling Pipelined Scan Data	757
Automatically Inserting Head and Tail Pipeline Registers	757
Specifying User-Defined Head and Tail Pipeline Registers	759
Using Pipelined Scan Data With Scan Compression	761
Configuring Pipelined Scan Data in a Compressed Scan Flow	761
Avoiding X Capture in Head Pipeline Registers	764
Adding Pipeline Stages at the Compressor Inputs	766
Pipelined Scan Data Specifications	768
Scan Architecture	768
Scan Register Synchronization	768
Pipelined Scan Data Test Protocol Format	769
Pipelined Scan Data Limitations	769
Hierarchical Flows With Pipelined Scan Data	770
General Rules	771
Pipelined Scan Data in the Standard Scan HSS Flow	772
Pipelined Scan Data in the HASS and Hybrid Flows	772
Sharing Codec Scan I/O Pins	773
Specifying the I/O Sharing Configuration	775
Determining the Fully Shared I/O Configuration	777
Determining Shared Input Pin Types	777
Adding High X-Tolerance Block-Select Pins	780
Automatically Computing the Fully Shared Configuration	781
Manually Computing the Fully Shared Configuration	782
Codec I/O Sharing in the HASS Flow	783
Codec I/O Sharing in the Hybrid Flow	783
Codec I/O Sharing in the Top-Down Flat Flow	785
Codec I/O Sharing With OCC Controllers	787
Codec I/O Sharing With Identical Cores	789
Identical Core Connections	789

Specifying Identical Cores	790
Using Scrambled Output Connections	791
Specifying Shared Codec Inputs With Dedicated Codec Outputs	792
Codec I/O Sharing With Shared Codec Controls	793
Configuring Shared Codec Controls	793
Specifying User-Defined Codec Enable Signals	795
Codec I/O Sharing Groups	796
Defining Sharing Groups in the HASS Flow	796
Defining Sharing Groups in the Hybrid Flow	797
Defining Sharing Groups for Codecs in Partitioned Cores	800
Defining Sharing Groups in the Top-Down Flat Flow	802
Codec I/O Sharing and Standard Scan Chains	803
Codec I/O Sharing and Pipelined Scan Data	806
Integrating Cores That Contain Shared Codec I/O Connections	809
Integrating Shared I/O Cores	809
Integrating Identical High X-Tolerance Shared I/O Cores	811
Integrating Shared I/O Cores Using Shared Codec Controls	813
Integrating Shared I/O Cores That Contain Shared Codec Controls	814
Shared Codec I/O Limitations	815
Implicit Scan Chains	816
Defining Implicit Scan Chains	817
Implicit Scan Chain Script Example	819
Protocol Example	820
Limitations	821
21. DFTMAX Compression With Serializer	822
Overview	823
Architecture	824
Serializer Clock Controller	825
Deserializer Registers	825
Serializer Registers	825
Serializer Operation	825
Higher Shift Speed and Update Stage	826
Scan-Enable Signal Requirements for Serializer Operation	830
Timing Paths	831
Scan Clocks	832
Deserializer/Serializer Update Stage Register Clocks	833
Specifying a Clock for Deserializer/Serializer Registers	833

Staggered Scan Clocks	834
Specifying Scan Clock Ports	835
User Interface	835
Configuring Serialized Compressed Scan	836
Deserializer/Serializer Register Size	837
Serializer Implementation Flow	838
Serialized Compressed Scan Core Creation	838
Serializer Core-Level Flow	839
User-Defined Ports for the Serializer Core-Level Flow	840
Nondefault Scan Clock Timing for Core-Level Flows	841
Top-Down Flat Flow	842
Serial Mode and Standard Scan Mode	842
Serial Mode, Parallel Mode, and Standard Scan Mode	842
Top-Down Partition Flow	844
Serializer Chains Dedicated to Each Partition	845
Serializer Chains Concatenated Across Partitions	847
HASS Flow	849
Serializer Chains Dedicated to Each Core	850
Serializer Chains Concatenated Across Cores	855
Hybrid Flow	856
Serializer Chains Concatenated Across Cores	859
Serializer IP Insertion	859
Configuring Serializer IP Insertion	861
Serializer IP Insertion in the Top-Down Flat Flow	862
Serializer IP Insertion in the Top-Down Flat Flow With Partitions	863
Serializer IP Insertion in the HASS Flow	864
Referencing Multiple Codecs in Compressed Scan Cores	866
Serializer IP Insertion in the Hybrid Flow	868
Serializer IP Insertion in the Hybrid Flow With Top-Level Partitions	869
Incorporating External Chains Into the Hybrid Serializer IP Flows	870
Serializer IP Insertion and Standard Scan Chains	873
Limitations	874
Wide Duty Cycle Support for Serializer	875
Block Diagram	876
Timing Diagram	877
Internally Generated Clocks	878

Wide Duty Cycle in a Core-Level Flow	880
Wide Duty Cycle in the HASS Flow	880
Wide Duty Cycle in the Hybrid Flow	880
Dual STIL Flow Parallel Patterns	881
Limitations	883
Serializer in Conjunction With On-Chip Clocking Controllers	883
OCC and SPC Chains in a Serializer Design	883
Using Serializer With User-Defined OCC Controllers	884
Using a Serializer Clock Controller With Multiple OCC Controllers	884
Waveforms for a Serializer With OCC Controllers	886
Using Integrated Clock-Gating Cells in the Serializer Clock Controller	887
User-Defined Pipelined Scan Data	887
Running TestMAX ATPG on Serializer Designs	888
Simulation and Patterns	888
STIL Protocol File	889
load_unload Procedure	889
UserKeywords SerializerStructures	892
Compressor Structures	896
ClockStructures	898
Debugging TestMAX ATPG Serializer DRC Errors	899
Debugging R33 Through R38 DRC Errors	899
Providing Guidance for R34 and R36 DRC Errors	902
Pattern Translation	905
Translating Parallel Mode Patterns to Serial Mode Patterns	905
Translating Serial Mode Patterns to Standard Scan Mode Patterns	908
Known Issues	908
C1 Violations	908
Serializer Core-Level Flow With Pipelined Scan Data Insertion	909
DFTMAX Compression With Serializer Limitations	909
Out-of-Scope Serializer Functionality	910
DFTMAX Compression Error Messages	910
TEST-1093	910
TEST-1094	911
TEST-1095	911
TEST-1096	911
TEST-1097	911

Part 4: DFTMAX Ultra Compression

22. Introduction to DFTMAX Ultra	913
The DFTMAX Ultra Compression Architecture	913
Usage Flow	914
Hierarchical DFT Insertion	915
Test Pattern Creation Using TestMAX ATPG	916
Pattern Simulation	917
23. DFTMAX Ultra Compression Architecture	918
DFTMAX Ultra Compression Architecture	918
Input Shift Register and Decompression MUX	919
Control Register	920
Output XOR Compression Tree and Shift Register	920
Test Pattern Scan Procedure	921
Scan-Enable Signal Requirements for Codec Operation	922
Multiple-Input, Multiple-Output Architecture	922
DFTMAX Ultra Architectures for On-Chip Clocking (OCC)	924
External Clock Chain	924
Compressed Clock Chain	925
24. Using DFTMAX Ultra Compression	927
DFTMAX Ultra Compression Requirements	927
Top-Down Insertion Compressed Scan Flow	928
Enabling DFTMAX Ultra Compression	928
Configuring the DFTMAX Ultra Codec	930
Configuring the Codec Clock	932
Top-Down Insertion Compressed Scan Flow With Partitions	934
Using Dedicated Scan Data Connections for Each Partition	934
Using Serial Scan Data Connections Between Partitions	936
Per-Partition Streaming Configuration Commands	938
set_streaming_compression_configuration	938
set_dft_signal	939
The Multiple-Input, Multiple-Output Codec Architecture	939

DFTMAX Ultra Compression and Multiple Test Modes	940
Defining Multiple DFTMAX Ultra Compressed Scan Modes	941
Mixing DFTMAX and DFTMAX Ultra Compression Modes	942
Per-Test-Mode Streaming Configuration Options	943
Using OCC Controllers With DFTMAX Ultra Compression	944
Creating External Clock Chains	944
Automatically Creating External Clock Chains	945
Manually Specifying External Clock Chains	945
Budgeting Scan I/Os and External Clock Chains	946
Creating Compressed Clock Chains	946
OCC Controllers and Streaming Codec Scan-Enable Constraints	947
Reducing Power Consumption in DFTMAX Ultra Designs	948
Reducing Compressor Power When Codec Is Inactive	948
Reducing Scan Shift Power Using Shift Power Groups	949
The Shift Power Groups Architecture	949
Configuring Shift Power Groups	951
Integrating Cores With Shift Power Groups in Hierarchical Flows	952
Configuring Shift Power Groups in TestMAX ATPG	954
Using Shift Power Groups With Other DFT Features	955
Limitations of Shift Power Groups	956
Planning, Previewing, and Inserting DFTMAX Ultra Compression	957
Planning the Streaming DFT Architecture	957
DFT Planner Flow Report	957
DFT Planner Elements Report	960
DFT Planner Limitations	964
Previewing and Inserting DFT Logic	965
Writing Out Test Protocols for TestMAX ATPG	966
Library Cell Requirements for Codec Implementation	967
25. Hierarchical DFTMAX Ultra Compression	968
Overview of Hierarchical DFTMAX Ultra Compression	968
Creating Cores for Integration	969
Performing Core Integration	970
Automatic Detection of Existing Logic Types	970
Configuring Core Integration	971
Configuring the Standard Scan Mode	971
Configuring the Compressed Scan Mode	972
Core Integration Script Examples	975

Integrating Only Compressed Scan Cores	975
Integrating Compressed Scan Cores With Uncompressed Logic	976
Using DFT Partitions During Core Integration	976
Using Multiple Test Modes in Hierarchical Flows	978
Mixing DFTMAX and DFTMAX Ultra Compression Core Modes	979
<hr/>	
26. DFTMAX Ultra Limitations and Known Issues	981
DFT Synthesis Limitations	981
Supported DFT Insertion Flows	983
<hr/>	
27. DFTMAX Ultra STIL Protocol File Syntax	984
STIL Protocol File Contents	984
STIL Protocol File Example	984
<hr/>	
28. DFTMAX Ultra Flow Naming Conventions	990
Describing Existing Logic	990
Describing DFT Logic To Be Inserted	994
Describing Additional DFT Features	996
Partitions	996
Scan I/Os	997
Multiple Test Modes	999
Additional Naming Convention Rules	999
Scan Flow Mapping	999
<hr/>	
Part 5: DFTMAX LogicBIST Self-Test	
<hr/>	
29. Introduction to LogicBIST	1003
Introduction to LogicBIST	1003
LogicBIST Requirements	1004
The LogicBIST Flow	1004
<hr/>	
30. The LogicBIST Architecture	1007

LogicBIST Architecture Overview	1007
The LogicBIST Decompressor	1008
The LogicBIST Compressor	1009
The LogicBIST BIST Controller	1010
The LogicBIST Clock Controller	1011
The LogicBIST Control and Data Signals	1012
The LogicBIST Operational Modes	1012
The LBIST_EN and START Signals	1012
The STATUS_0 and STATUS_1 Signals	1013
The Scan-In and Scan-Out Signals	1014
LogicBIST Clock Control	1015
Overview of Clock Configurations	1015
External Clocks	1016
OCC-Controlled Clocks With Default Capture Behavior	1017
OCC-Controlled Clocks With Weighted Clock Capture Groups	1018
External and Internal Clocks in the Same Design	1020
Isolating the Design During LogicBIST Self-Test	1021
Isolating the Self-Test Design Using Core Wrapping	1021
Isolating the Self-Test Design Using Test Points	1022
Comparing the Two Isolation Approaches	1023
Providing Testability for LogicBIST Self-Test	1024
Enabling DFT Logic During Autonomous Self-Test	1024
Blocking Internal X Sources	1027
Ensuring Testability for Reset Signals	1028
Ensuring Testability for Integrated Clock-Gating Cells	1029
<hr/>	
31. Using LogicBIST Self-Test	1031
Configuring LogicBIST Self-Test	1031
Defining the LogicBIST Control Signals	1032
Defining the LogicBIST Scan-In Signal	1032
Defining the LogicBIST Self-Test Mode	1032
Configuring the PRPG and MISR Lengths	1033
Configuring the Pattern Counter and Shift Counter Lengths	1034
Configuring the Self-Test Capture Clock Timing	1034
Configuring Clock and Reset Weights	1036
Configuring Self-Test Isolation Logic	1037
Configuring Wrapper Chain Isolation Logic	1037
Configuring Test Point Isolation Logic	1039

Controlling Self-Test Through IEEE 1500 Logic	1039
Inserting LogicBIST in Designs With Trailing-Edge Flip-Flops	1041
Inserting LogicBIST in Designs With External Chains	1041
Inserting LogicBIST in Designs With Clock-Gating Cells	1042
Previewing and Inserting the LogicBIST Implementation	1043
Previewing the LogicBIST Implementation	1043
Inserting the LogicBIST Logic	1046
Writing Out the LogicBIST Design Files	1046
Computing the Seed and Signature Values in TestMAX ATPG	1047
Setting the Seed and Signature Values in Synthesis	1049
Simulating Autonomous BIST Operation	1050
Integrating the Self-Test Logic into the Functional Design Logic	1051
Connecting the Self-Test Signals to the Functional Design Logic	1051
Ensuring the Required Test Mode for Autonomous Self-Test	1054
Monitoring the Self-Test Status Signals	1055
Example LogicBIST Scripts	1057
Example Core Insertion Script Using Core Wrapping	1057
Example Core Insertion Script Using Test-Point Isolation	1058
Example Script to Automatically Set Seed and Signature Values	1059
32. Advanced LogicBIST Configuration	1062
Using Programmable LogicBIST Configuration Values	1062
Simplifying the MISR XOR Compressor	1064
Simplifying the Weighted Clock/Reset Logic	1065
Minimizing Reconfiguration MUXs Across Test Modes	1065
Choosing a Particular Integrated Clock-Gating Cell	1066
Implementing Burn-In Mode	1067
Implementing Power Ramp-Up and Ramp-Down Logic	1068
Implementing MISR Monitoring Logic	1070
Changing the Test Mode Used for Autonomous Self-Test	1071
Post-DFT Design Optimization	1071
Post-DFT Optimization and BIST Constants	1072
Preserving the BIST Constants in a compile Flow	1073
Preserving the BIST Constants in a compile_ultra Flow	1073
Regenerating Seed and Signature Values after Design Changes	1074

Ungrouping LogicBIST Blocks for Additional Area Reduction	1075
---	------

33. LogicBIST Limitations and Known Issues	1077
---	------

LogicBIST Limitations and Known Issues	1077
--	------

Appendices

A. DFT Attributes	1080
Cell Attributes	1080
Design Attributes	1082
Pin Attributes	1082
Port Attributes	1083
 B. Legacy Test Point Insertion	1084
Introduction	1084
Differences Between Newer and Legacy Test Point Features	1085
Test Point Types	1085
Force Test Points	1086
Control Test Points	1087
Observe Test Points	1090
Test Point Signals	1090
Sharing Test Point Scan Cells	1091
Automatically Inserted Test Points (Legacy)	1093
Enabling Automatic Test Point Insertion	1093
Configuring Pattern Reduction and Testability Test Point Insertion	1094
Script Example	1096
User-Defined Test Points (Legacy)	1097
Configuring User-Defined Test Points	1097
User-Defined Test Points Example	1099
Previewing the Test Point Logic	1101
Inserting the Test Point Logic	1102

C. Legacy RTL Design Rule Checking	1103
Understanding the Flow	1103
Specifying Setup Variables	1105
Generating a Test Protocol	1105
Defining a Test Protocol	1105
Reading in an Initialization Protocol in STIL Format	1106
Setting the Scan Style	1109
Design Examples	1110
Test Protocol Example 1	1110
Test Protocol Example 2	1111
Running RTL Test DRC	1114
Understanding the Violations	1115
Violations That Prevent Scan Insertion	1115
Uncontrollable Clocks	1115
Asynchronous Control Pins in Active State	1116
Violations That Prevent Data Capture	1116
Clock Used As Data	1117
Black Box Feeds Into Clock or Asynchronous Control	1117
Source Register Launch Before Destination Register Capture	1118
Registered Clock-Gating Circuitry	1119
Three-State Contention	1120
Clock Feeding Multiple Register Inputs	1120
Violations That Reduce Fault Coverage	1121
Combinational Feedback Loops	1121
Clocks That Interact With Register Input	1122
Multiple Clocks That Feed Into Latches and Flip-Flops	1122
Black Boxes	1124
Limitations	1125
Glossary	1126

About This User Guide

The *TestMAX DFT User Guide* describes the process for inserting standard scan, compressed scan, and self-test logic into a design, using either a flat (top-down) or hierarchical (bottom-up) flow. You can then generate test patterns for these designs with the Synopsys® TestMAX™ ATPG tool.

The *TestMAX DFT User Guide* is organized into the following parts:

- Part I: DFT Overview – Provides overview of design-for-test (DFT) concepts and flows
- Part II: DFT Compiler Scan – Describes DFT configuration and insertion for standard scan logic
- Part III: DFTMAX Compression – Describes configuration and insertion for DFTMAX compressed scan, including serialized compressed scan logic
- Part IV: DFTMAX Ultra Compression – Describes configuration and insertion for DFTMAX Ultra streaming compressed scan logic
- Part V: DFTMAX LogicBIST Self-Test – Describes configuration and insertion for DFTMAX LogicBIST self-test logic

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

This preface includes the following sections:

- [New in This Release](#)
- [Related Products, Publications, and Trademarks](#)
- [Conventions](#)
- [Customer Support](#)

New in This Release

Information about new features, enhancements, and changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the TestMAX DFTRelease Notes on the SolvNetPlus site.

To see the *TestMAX DFT Release Notes*:

1. Go to the SolvNet Download Center located at the following address:
<https://solvnet.synopsys.com/DownloadCenter>
2. Select “TestMAX DFT (Synthesis),” and then select a release in the list that appears.

Related Products, Publications, and Trademarks

For additional information about the TestMAX DFT tool, see the documentation on the Synopsys SolvNetPlus support site at the following address:

<https://solvnetplus.synopsys.com>

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

- Design Compiler®
- Design Vision™
- Library Compiler™
- PrimeTime®
- Power Compiler™
- TestMAX™ Advisor
- TestMAX™ ATPG

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates syntax, such as <code>write_file</code> .
<i>Courier italic</i>	Indicates a user-defined value in syntax, such as <code>write_file design_list</code>
Courier bold	Indicates user input—text you type verbatim—in examples, such as <code>prompt> write_file top</code>
Purple	<ul style="list-style-type: none">Within an example, indicates information of special interest.Within a command-syntax section, indicates a default, such as <code>include_enclosing = true false</code>
[]	Denotes optional arguments in syntax, such as <code>write_file [-format fmt]</code>
...	Indicates that arguments can be repeated as many times as needed, such as <code>pin1 pin2 ... pinN</code> .
	Indicates a choice among alternatives, such as <code>low medium high</code>
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Bold	Indicates a graphical user interface (GUI) element that has an action associated with it.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy .
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.

Customer Support

Customer support is available through SolvNetPlus.

Accessing SolvNetPlus

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

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

<https://solvnetplus.synopsys.com>

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

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

Contacting Customer Support

To contact Customer Support, go to <https://solvnetplus.synopsys.com>.

Part 1: DFT Overview

1

Introduction to Synopsys DFT Tools

The DFT Compiler and TestMAX DFT tools comprise the Synopsys test synthesis solution. They enable transparent implementation of DFT capabilities into the Synopsys synthesis flow without interfering with functional, timing, signal integrity, or power requirements.

This chapter introduces the basic features, benefits, and components of the DFT Compiler and TestMAX DFT tools. It includes the following topics:

- [Key Features](#)
 - [Key Benefits](#)
 - [DFT Compiler and the Synopsys TestMAX Product Platform](#)
 - [DFTMAX Scan Compression](#)
 - [DFTMAX Ultra Scan Compression](#)
 - [DFTMAX LogicBIST Self-Test](#)
 - [Other Tools in the Synopsys Test and Yield Solution](#)
-

Key Features

The DFT Compiler and TestMAX DFT tools offer the following features:

- One-pass test synthesis
- Comprehensive RTL and gate-level DFT design rule checking
- Rapid scan synthesis
- DFTMAX scan compression technologies
- Hierarchical scan synthesis
- Automatic and manual test point insertion
- Core wrapping with register reuse
- Physical scan ordering
- Boundary-scan insertion

Key Benefits

DFT Compiler enables you to quickly and accurately account for testability and resolve any test issues early in the design cycle. RTL test design rule checking enables you to create test-friendly RTL that can then be easily synthesized in the one-pass test synthesis environment. The integration of test within the Design Compiler topographical environment ensures predictable timing closure and achieves physically optimized scan designs.

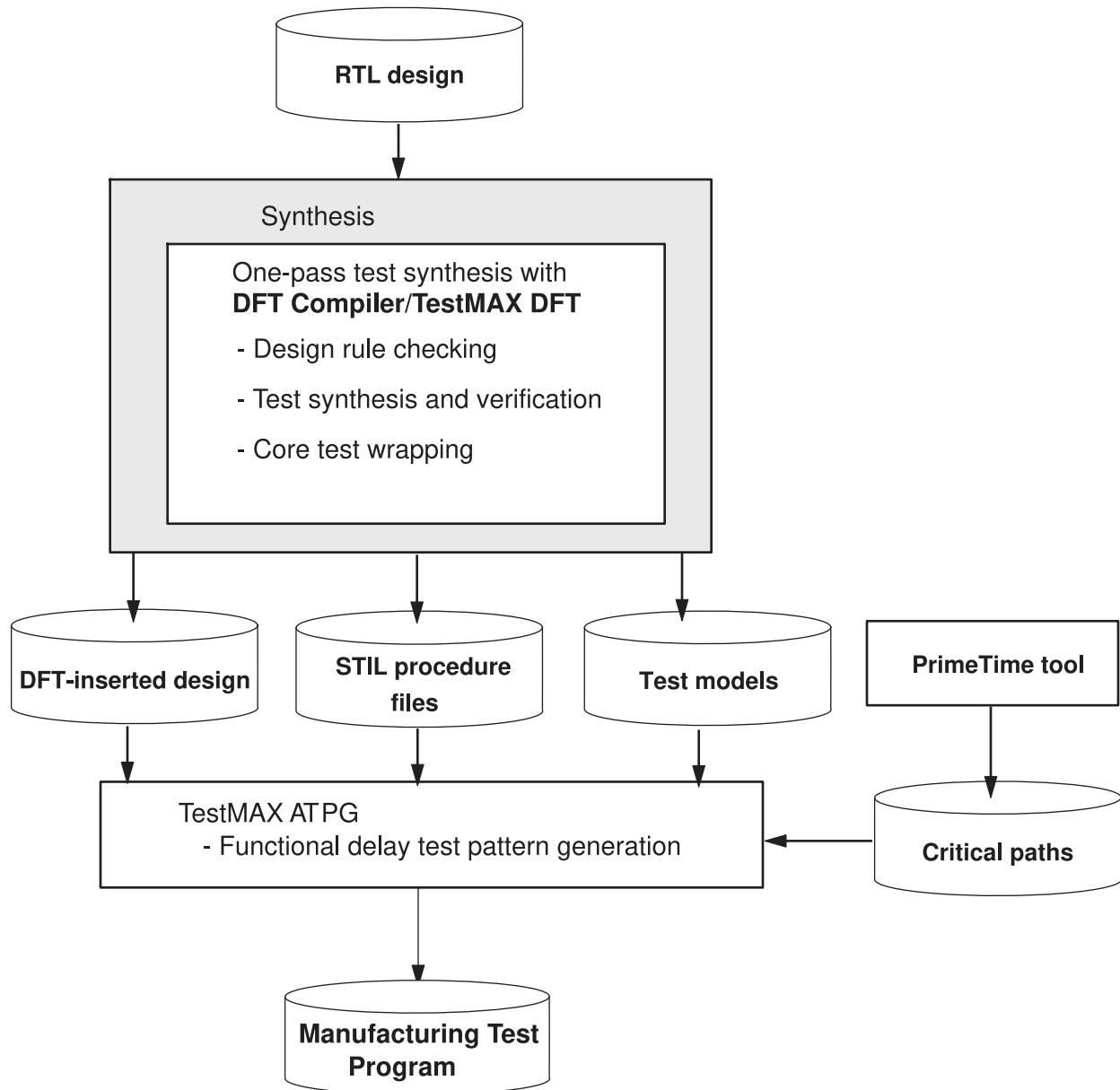
Note the following key benefits of DFT Compiler:

- Offers transparent DFT implementation within the synthesis flow
 - Accounts for testability early in the design cycle at RTL
 - Removes unpredictability from the back end of the design process
 - Achieves predictable timing, power, and signal integrity closure concurrent with test
-

DFT Compiler and the Synopsys TestMAX Product Platform

The DFT Compiler and TestMAX DFT tools, along with the TestMAX ATPG tool, are part of the larger Synopsys TestMAX product platform, as shown in [Figure 1](#).

Figure 1 *Synopsys TestMAX Test Automation Solution*



DFTMAX Scan Compression

The TestMAX DFT tool provides DFTMAX scan compression, which enables you to implement test data volume scan compression without affecting the functional, timing, or power requirements of your design.

DFTMAX scan compression provides the following key benefits and features:

- Significant test time and test volume reduction
- Same high test coverage and ease of use as standard scan
- No impact on design timing or design physical implementation
- One-pass test compression synthesis flow
- Hierarchical scan synthesis flows

See Also

- [Chapter 16, Introduction to DFTMAX](#) for more information about DFTMAX scan compression
-

DFTMAX Ultra Scan Compression

The TestMAX DFT tool provides DFTMAX Ultra scan compression, an advanced test compression solution that is designed for hierarchical flows to deliver high quality results as measured by test time, data volume, design area and congestion, and time to implementation.

DFTMAX Ultra scan compression provides the following key benefits and features:

- Familiar user interface for DFTMAX compression users
- Very high scan compression ratios, even with few scan I/O pins
- Same high test coverage and ease of use as standard scan
- Minimal impact to the clock tree (no codec clock controller required)
- Improved ease of use for hierarchical scan synthesis flows

See Also

- [Chapter 22, Introduction to DFTMAX Ultra](#) for more information about DFTMAX Ultra scan compression

DFTMAX LogicBIST Self-Test

Built-in self-test (BIST) capability enables a design to test itself autonomously without using external test data. DFTMAX LogicBIST self-test provides a low-overhead logic BIST (LBIST) solution for digital logic designs, such as automotive applications.

DFTMAX LogicBIST self-test provides the following key benefits and features:

- Low BIST controller area overhead
- Reuses the scan chain and test-mode control logic already implemented for manufacturing test
- Low LogicBIST-mode pin requirements
- Easy interface to functional logic
- Seed and expected signature values can be hardcoded or programmable
- Targets stuck-at and transition-delay faults
- Simple one-pass DFT insertion flow

See Also

- [Chapter 29, Introduction to LogicBIST](#) for more information about DFTMAX LogicBIST self-test

Other Tools in the Synopsys Test and Yield Solution

Synopsys provides a complete test and yield solution that accelerates higher quality, reliability, and yield. It spans the full flow from IP all the way through post-silicon test, and it covers all the key design blocks: logic, memory, I/O, and analog and mixed-signal (AMS). The return loop indicates the ability to optimize design rules, layouts, circuit design, and so on, based on learning from silicon analysis.

This solution includes:

- TestMAX™ ATPG, TetraMAX® II ATPG, and TetraMAX ATPG
 - Provides high-coverage pattern generation for digital-logic designs
- TestMAX™ Advisor and SpyGlass® DFT ADV
 - Provides RTL testability analysis and improvement for maximum ATPG coverage
- STAR Hierarchical System
 - Provides IEEE standards-based hierarchical SoC test and pattern-porting capabilities

- STAR Memory System[®] IP
 - Provides advanced test, diagnostics and repair for embedded and external memory manufacturing defects (including FinFET technologies); and provides error correction for lifetime transient errors
- DesignWare IP
 - Provides high-speed interfaces (such as USB, DDR, and PCIe) with embedded self-test capability
- Yield Explorer
 - Identifies dominant yield loss mechanisms by combining fabrication plant, test, and design data
- Camelot[™]
 - Enables product engineers to exactly identify the failure mechanisms to drive quick resolution of the problems

2

Designing for Manufacturing Test

The manufacturing test process ensures high-quality integrated circuits by screening out devices with manufacturing defects. You can attain maximum test coverage of your integrated circuit by using DFT Compiler when you adopt structured DFT techniques.

This chapter includes the following topics:

- [Functional Testing Versus Manufacturing Testing](#)
- [Modeling Manufacturing Defects](#)
- [Achieving Maximum Fault Coverage for Sequential Cells](#)
- [Understanding the Full-Scan Test Methodology](#)
- [Scan Styles Supported by DFT Compiler](#)
- [Describing the Test Environment](#)
- [Test Design Rule Checking Functions](#)
- [Getting the Best Results With Scan Design](#)

Functional Testing Versus Manufacturing Testing

IC test is composed of two primary approaches: functional testing and manufacturing testing.

Functional testing verifies that your circuit performs as it is designed to perform. For example, assume that your design is an adder circuit. Functional testing verifies that your circuit performs the addition function and computes the correct results over the range of values tested. However, exhaustive testing of all possible input combinations grows exponentially as the number of inputs increases. To maintain a reasonable test time, you need to focus functional test patterns on the general function and corner cases.

Manufacturing testing verifies that your circuit does not have manufacturing defects by focusing on circuit structure rather than functional behavior.

Manufacturing defects include problems such as

- Power or ground shorts
- Open interconnect on the die caused by dust particles
- Short-circuited source or drain on the transistor, caused by metal spike-through

Manufacturing defects might remain undetected by functional testing yet cause undesirable behavior during circuit operation. To provide the highest-quality products, development teams must prevent devices with manufacturing defects from reaching customers. Manufacturing testing enables development teams to screen devices for manufacturing defects.

Typically, development teams perform both functional and manufacturing testing of devices.

Modeling Manufacturing Defects

When a manufacturing defect occurs, the physical defect has a logical effect on the circuit behavior. An open connection can appear to float either high or low, depending on the technology. A signal shorted to power appears to be permanently high. A signal shorted to ground appears to be permanently low. Many manufacturing defects can be represented using the industry-standard stuck-at fault model.

This topic covers the following:

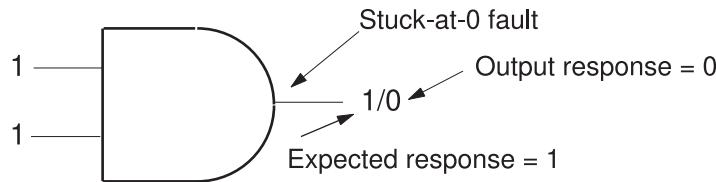
- [Understanding Stuck-At Fault Models](#)
- [Determining Coverage](#)
- [Understanding Fault Simulation](#)
- [Automatically Generating Test Patterns](#)
- [Formatting Test Patterns](#)

Understanding Stuck-At Fault Models

The stuck-at-0 model represents a signal that is permanently low, regardless of the other signals that normally control the node. The stuck-at-1 model represents a signal that is permanently high, regardless of the other signals that normally control the node.

For example, assume that you have a 2-input AND gate that has stuck-at-0 fault on the output pin. As shown in [Figure 2](#), the output is always 0, regardless of the logic level of the two inputs.

Figure 2 2-Input AND Gate With Stuck-At-0 Fault on Output Pin



Controllable and Observable Faults

The node of a stuck-at fault must be controllable and observable for the fault to be detected.

A node is controllable if you can drive it to a specified logic value by setting the primary inputs to specific values. A primary input is an input that can be directly controlled in the test environment.

A node is observable if you can predict the response on it and propagate the fault effect to the primary outputs, where you can measure the response. A primary output is an output that can be directly observed in the test environment.

To detect a stuck-at fault on a target node:

- Control the target node to the opposite of the stuck-at value by applying data at the primary inputs.
- Make the node's fault effect observable by controlling the value at all other nodes affecting the output response, so the targeted node is the active (controlling) node.

The set of logic 0s and 1s applied to the primary inputs of a design is called the input stimulus. The resulting values at the primary outputs, assuming a fault-free design, are called the expected response. The actual values measured at the primary outputs are called the output response.

If the output response does not match the expected response for a given input stimulus, the input stimulus has detected the fault.

Detecting Stuck-At Faults

To detect a faulty node that is stuck-at-0, you need to apply an input stimulus that forces a particular node to 1. For the 2-input AND gate shown in [Figure 2](#), for example, apply a logic 1 at both inputs. The expected response for this input stimulus is logic 1, but the output response is logic 0. This input stimulus detects the stuck-at-0 fault.

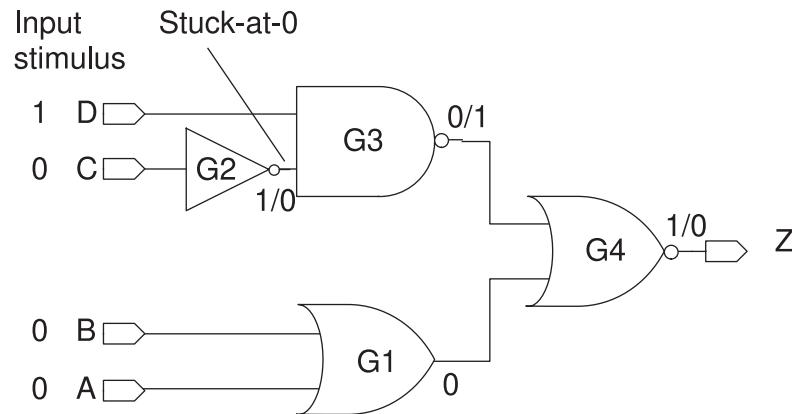
This method of determining the input stimulus to detect a fault uses the single stuck-at-fault model. The single stuck-at fault model assumes that only one node is faulty and that all other nodes in the circuit are good.

The single stuck-at fault model greatly reduces the complexity of fault modeling and is technology independent, enabling the use of algorithmic pattern generation techniques.

A more complicated example shows the requirement of controlling all other nodes to ensure the observability of a particular target node.

[Figure 3](#) shows a circuit with a detectable stuck-at-0 fault at the output of cell G2.

Figure 3 Simple Circuit With Detectable Stuck-At Fault



To detect the fault, control the output of cell G2 to logic 1 (the opposite of the faulty value) by applying a logic 0 value at primary input C.

To ensure that the fault effect is observable at primary output Z, control the other nodes in the circuit so that the response value at primary output Z depends only on the output of cell G2, as follows:

- Apply a logic 1 at primary input D so the output of cell G3 depends only on the output of cell G2. The output of cell G2 is the controlling input of cell G3.
- Apply logic 0s at primary inputs A and B so the output of cell G4 depends only on the output of cell G2.

Given the input stimulus of A = 0, B = 0, C = 0, and D = 1, a fault-free circuit produces a logic 1 at output port Z. If the output of cell G2 is stuck-at-0, the value at output port Z is a logic 0 instead. Thus, this input stimulus detects a stuck-at-0 fault on the output of cell G2.

This set of input stimulus and expected response values is called a test vector. Following the process previously described, you can generate test vectors to detect stuck-at-1 and stuck-at-0 faults for each node in the design.

Determining Coverage

A common definition of the testability of a design is the extent to which the design can be tested for the presence of manufacturing defects, as represented by the single stuck-at fault model.

Common metrics for measuring coverage are:

- Test coverage—the percentage of detected faults for all detectable faults; gives the most meaningful measure of test pattern quality
- Fault coverage—the percentage of detected faults for all faults; gives no credit for undetectable faults
- ATPG effectiveness—the percentage of faults that are resolvable by the ATPG process; full credit is given to faults which are detected and faults which are proven to be untestable

For larger combinational designs and sequential designs, it is not feasible to analyze the coverage results for existing functional test vectors or to manually generate test vectors to achieve high fault coverage results. Fault simulation tools determine the coverage of a set of test vectors. Automatic test pattern generation (ATPG) tools generate manufacturing test vectors. Both types of automated tools require models for all logic elements in your design to correctly calculate the expected response. Your semiconductor vendor provides these models.

For more details on how these metrics are calculated, see “Coverage Calculations” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Understanding Fault Simulation

Fault simulation determines the fault coverage of a set of test vectors. It can be thought of as performing many logic simulations concurrently—one that represents the fault-free circuit (the good machine) and many that represent the circuits containing single stuck-at faults (the faulty machines). Fault simulation detects a fault each time the output response of the faulty machine differs from the output response of the good machine for a given vector.

Fault simulation determines all faults detected by a test vector. Fault simulating the test vector generated to detect the stuck-at-0 fault on the output of G2 in [Figure 3](#) shows that this vector also detects the following single stuck-at faults:

- Stuck-at-1 on all pins of G1 (and ports A and B)
- Stuck-at-1 on the input of G2 (and port C)
- Stuck-at-0 on the inputs of G3 (and port D)
- Stuck-at-1 on the output of G3
- Stuck-at-1 on the inputs of G4
- Stuck-at-0 on the output of G4 (and port Z)

You can generate manufacturing test vectors by manually generating test vectors and then fault-simulating them to determine the fault coverage. For large or complex designs, this process is time-consuming and often does not result in high fault coverage results.

Automatically Generating Test Patterns

You use an ATPG tool (such as the TestMAX ATPG tool) to generate test patterns and provide fault coverage statistics for the generated pattern set. The difference between test vectors and test patterns is defined in [Chapter 3, Scan Design Techniques](#). For now, consider the terms *test vector* and *test pattern* as synonymous.

When using ATPG for combinational circuits, it is usually possible to generate test vectors that provide high fault coverage for combinational designs. Combinational ATPG tools use both random and deterministic techniques to generate test patterns for stuck-at faults on cell pins.

During random pattern generation, the tool assigns input stimulus in a pseudorandom manner and then fault-simulates the generated vector to determine which faults are detected. As the number of faults detected by successive random patterns decreases, ATPG shifts to a deterministic technique.

During deterministic pattern generation, the tool uses a pattern generation algorithm based on path-sensitivity concepts to generate a test vector that detects a specific fault in the design. After generating a vector, the tool fault simulates the vector to determine the complete set of faults detected by the vector. Test-pattern generation continues until all faults have either been detected or have been identified as undetectable by this algorithm.

Because of the effects of memory and timing, ATPG for sequential circuits is much more difficult than for combinational circuits. It is often not possible to generate high-fault-coverage test vectors for complex sequential designs, even when using sequential ATPG. Sequential ATPG tools use deterministic pattern generation algorithms based on extended applications of the path-sensitivity concepts.

Structured DFT techniques, such as internal scan, simplify the test-pattern generation task for complex sequential designs, resulting in higher fault coverage and reduced testing costs.

See Also

- [Chapter 3, Scan Design Techniques](#) for more information about testing a design with internal scan

Formatting Test Patterns

To screen out manufacturing defects in your chips, you need to translate the generated test patterns into a format acceptable to the automated test equipment (ATE). On the ATE, the logic 0s and 1s in the input stimulus are translated into low or high voltages to be applied to the primary inputs of the device under test. The logic 0s and 1s in the output response are compared with the voltages measured at the primary outputs. One test vector corresponds to one ATE cycle.

You might use more than one set of test vectors for manufacturing testing. The collection of all test vector sets used to test a design is often referred to as the test program.

Achieving Maximum Fault Coverage for Sequential Cells

You can achieve the best fault coverage results for sequential cells when all the nodes in your design are controllable and observable. Adding scan logic to your design enhances its controllability and observability.

Controllability of Sequential Cells

For sequential cells, controllability requirements ensure that all state elements can be controlled, by scan or other means, to desired state values from the boundary of the design. These requirements are involved primarily with the shift operations in scan test.

In an ideal full-scan design, the scan chain contains all state elements, the scan chain operates correctly, and the circuit is fully controllable. In this ideal full-scan circuit, any circuit state can be achieved.

Observability of Sequential Cells

For sequential cells, observability requirements ensure predictable capture of the next state of the circuit and visibility at the boundary of the design. These requirements are involved primarily with the capture operations in scan test. Although scan shift problems

affect the observability of sequential cells, they are typically detected during controllability checks.

In the context of scan design, a circuit is observable when the tester can successfully clock the scan cells in the circuit and then shift their state to the scan outputs.

For a circuit to be observable, the tester must be able to

1. Observe the primary outputs of the circuit after shifting in a scan pattern.

Normally, this involves no DFT and does not present problems.

2. Reliably capture the next state of the circuit.

If the functional operation is impaired, unpredictable, or unknown, the next state is unknown. This unknown state makes at least part of the circuit unobservable.

3. Extract the next state by shifting out the output response of the scan cells.

This process is similar to shifting in a scan pattern. The additional requirement is that the shift registers pass data reliably to the output ports.

The rules governing the controllability and observability of scan cells are called *test design rules*.

See Also

- [Chapter 13, Pre-DFT Test Design Rule Checking](#) for more information about checking for test design rule violations before DFT insertion

Understanding the Full-Scan Test Methodology

In the full-scan methodology, DFT Compiler replaces all sequential cells in your design with their scannable equivalents during scan insertion.

A sequential cell might not be scannable because of test design rule violations or because you have explicitly excluded the cell from the scan chain. In this case, DFT Compiler classifies the cell as a black-box sequential cell during test design rule checking. Black-box sequential cells lower fault coverage results.

Because it is a more predictable methodology, full scan typically provides higher fault coverage in a shorter period of time than partial scan. Full scan also provides improved diagnostic capabilities compared to partial scan.

However, because full scan substitutes scannable equivalents for all sequential cells, it increases design area and decreases design performance. Integration with synthesis minimizes the area and performance overhead of full scan. In most cases, performance can be maintained in a full-scan design, but at the cost of additional area.

See Also

- [Chapter 3, Scan Design Techniques](#) for more information about full scan testing of a design

Scan Styles Supported by DFT Compiler

DFT Compiler supports the following scan styles:

- [Multiplexed Flip-Flop Scan Style](#)
- [Clocked-Scan Scan Style](#)
- [Level-Sensitive Scan Design \(LSSD\) Style](#)
- [Scan-Enabled Level-Sensitive Scan Design \(LSSD\) Style](#)
- [Summary of Supported Scan Cells](#)

These scan styles are described in the following topics.

Multiplexed Flip-Flop Scan Style

The multiplexed flip-flop scan style uses a multiplexed data input to provide serial shift capability. In functional mode, the scan-enable signal, acting as the multiplexer select line, selects the system data input. During scan shift, the scan-enable signal selects the scan data input. The scan data input comes from either the scan-input port or the scan output pin of the previous cell in the scan chain.

The following test pins are required on a multiplexed flip-flop equivalent cell:

- Scan-input
- Scan-enable
- Scan-output (can be shared with a functional output pin)

Test pins are identified in the `test_cell` group of the cell description in the logic library. For information on modeling test cells in your logic library, see the Library Compiler user guides.

Multiplexed flip-flop is the scan style most commonly supported in logic libraries. Most libraries provide multiplexed flip-flop equivalents for D, JK, and master-slave flip-flops.

See Also

- [Multiplexed Flip-Flop Scan Style on page 68](#) for more information about the multiplexed flip-flop scan style

Clocked-Scan Scan Style

The clocked-scan scan style uses a dedicated, edge-triggered test clock to provide serial shift capability. In functional mode, the system clock is active and system data is clocked into the cell. During scan shift, the test clock is active and scan data is clocked into the cell.

The following test pins, identified in the `test_cell` group of the scan cell description in the logic library, are required on a clocked-scan cell:

- Scan-input
- Test-clock
- Scan-output (can be shared with a functional output pin)

DFT Compiler supports clocked-scan cells for both flip-flops and latches.

See Also

- [Clocked-Scan Scan Style on page 72](#) for more information about the clocked-scan scan style

Level-Sensitive Scan Design (LSSD) Style

DFT Compiler supports three variations of the level-sensitive scan design (LSSD) style:

- Single-latch
- Double-latch
- Clocked

These variations can be mixed in a single design. The following section briefly describes these variations.

Both the single-latch and double-latch variations use the classical LSSD scan cell, which consists of two latches acting as a master-slave pair. The master latch has dual input ports and can latch either *functional* data or *scan* data. In functional mode, the system master clock input controls the data input. In scan mode, the test master clock input controls the transfer of data from the data input to the master latch. The slave clock input controls the transfer of data from the master latch to the slave latch.

The following test pins, identified in the `test_cell` group of the scan cell description in the logic library, are required on an LSSD cell:

- Scan-input
- Test master-clock
- Test slave-clock (except for double-latch LSSD)
- Scan-output (can be shared with a functional output pin)

See Also

- [LSSD Scan Style on page 76](#) for more information about the LSSD scan style, including the single-latch, double-latch, and clocked LSSD scan styles

Scan-Enabled Level-Sensitive Scan Design (LSSD) Style

The scan-enabled LSSD scan style provides edge-triggered flip-flop behavior in functional mode, but it uses master-slave test clocks to provide skew-tolerant serial shift capability in test mode. In functional mode, when the scan-enable signal is de-asserted, system data is clocked into the cell on clock edges. In test mode, when the scan-enable signal is asserted, test data is clocked through the cell using master-slave clocking. In test mode, the system clock is repurposed as the scan-shift slave clock, and a separate scan-shift master clock signal is required.

The following test pins, identified in the `test_cell` group of the scan cell description in the logic library, are required on a clocked-scan cell:

- Scan-input
- Test master-clock
- Test slave-clock (shared with functional clock pin)
- Scan-enable
- Scan-output (can be shared with a functional output pin)

See Also

- [Scan-Enabled LSSD Style on page 85](#) for more information about the scan-enabled LSSD scan style

Summary of Supported Scan Cells

Table 1 shows the scan cells supported by DFT Compiler. The columns represent circuit clocking in functional mode; the rows represent circuit clocking in scan mode. Use this table to determine scan cell support and the corresponding scan style for your design.

Table 1 Supported Scan Cells

Scan mode	Functional mode		
	Edge-triggered clock	Single level-sensitive clock	Dual master-slave level-sensitive clocks
Same edge-triggered clock	MUX flip-flop(multiplexed_flip_flop)		
Same master-slave level-sensitive clocks			MUX master-slave latch(multiplexed_flip_flop)
Different edge-triggered clock	Clocked-scan flip-flop(clocked_scan)	Clocked-scan latch(clocked_scan)	
Different dual master-slave level-sensitive clocks	Clocked LSSD(lssd)	Single-latch LSSD(lssd)	
Master-slave clocks with different master clock, same slave clock	Scan-enabled LSSD(scan_enabled_lssd)		Double-latch LSSD(lssd)

Note: The `scan_style` argument is shown in parentheses.

For example, if your design has one level-sensitive clock (C) in functional mode and two nonoverlapping clocks (A and B) for scan shift, you need to set the scan style to `lssd`.

Logic Library Considerations

The ability of DFT Compiler to support a particular scan style depends on whether the scan cells can be modeled in the logic library.

With the Library Compiler tool, you can use a state table to model sequential cells. State table models can accurately model the behavior of complex scan cells, such as those that have multiple clocks active at the same time. DFT Compiler does not support every complex sequential cell that can be modeled by using state table models.

For more information about modeling scan cells, see Library Compiler User Guide.

Describing the Test Environment

A test protocol completely describes the test environment for a design. The test protocol includes

- The test timing information
- The initialization sequence used to configure the design for scan testing
- The test configuration used to select between scan shift and parallel cycles during pattern application
- The pattern application sequence

The process for scan-testing a design is basically the same for every design. It consists of scanning data in, performing the normal operation sequence, and scanning data out.

The instructions for performing scan testing, however, are unique to each design. Those instructions include how to configure the design for scan testing, what ports are involved, and so on. A test protocol is the set of specific instructions for scan testing a design.

Test Design Rule Checking Functions

The test design rule checker has two distinct functions:

- As a standalone program, it provides feedback on the testability of the design to guide DFT.
- As a preprocessor to scan insertion, it flags valid sequential cells for scan replacement. In this mode, it produces no user output.

In test DRC, scan data is simulated symbolically, but the design is simulated deterministically.

Because rule checking depends on the dynamic operation of the design, design rule violations can be caused by both structural problems and operational problems. You can often modify the dynamics of the scan operation to fix a problem that appears to be structural.

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.

3

Scan Design Techniques

A variety of scan design techniques are available to help you prepare your design to take advantage of manufacturing test techniques. Each major technique is discussed in this chapter.

This chapter includes the following topics:

- [Internal Scan Design](#)
 - [Test for System-On-A-Chip Designs](#)
 - [Boundary Scan Design](#)
-

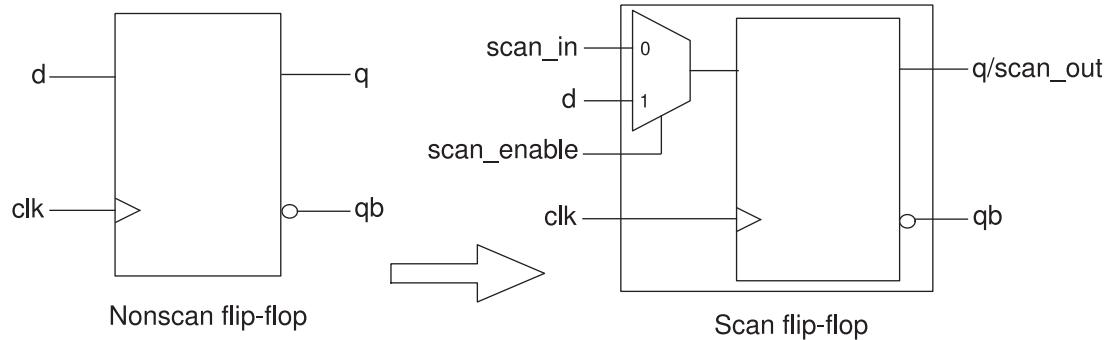
Internal Scan Design

Internal scan design is the most popular DFT technique; it also has the greatest potential for high fault coverage results. This technique simplifies the pattern generation problem by dividing complex sequential designs into fully isolated combinational blocks (full-scan design) or partially isolated combinational blocks (partial-scan design). Internal scan modifies existing sequential elements in the design to support a serial shift capability in addition to their normal functions. This serial shift capability enhances internal node controllability and observability with a minimum of additional I/O pins.

Scan Cells

[Figure 4](#) shows a D flip-flop modified to support internal scan by the addition of a multiplexer (this scan style is called multiplexed flip-flop). Inputs to the multiplexer are the data input of the flip-flop (d) and the scan-input signal (scan_in). The active input of the multiplexer is controlled by the scan-enable signal (scan_enable). Input pins are added to the cell for the scan_in and scan_enable signals. One of the data outputs of the flip-flop (q or qb) is used as the scan-output signal (scan_out). The scan_out signal is connected to the scan_in signal of another scan cell to form a serial scan (shift) capability.

Figure 4 D Flip-Flop With Scan Capability



Scan Chains

The modified sequential cells are chained together to form one or more large shift registers, called scan chains or scan paths. The sequential cells connected in a scan chain are scan controllable and scan observable. A sequential cell is scan controllable when it can be set to a known state by serially shifting in specific logic values. ATPG tools consider scan-controllable cells pseudo-primary inputs of the design. A sequential cell is scan observable when its state can be observed by serially shifting out data. ATPG tools consider scan-observable cells pseudo-primary outputs of the design.

Scan Cells in Semiconductor Vendor Libraries

Most semiconductor vendor libraries include pairs of equivalent nonscan and scan cells that support a given scan style. One special test cell is a scan flip-flop that logically combines a D flip-flop and a multiplexer, as shown in [Figure 4](#).

The Effect of Adding Scan Circuitry to a Design

Adding scan circuitry to a design usually has the following effects:

- Design size and power increase slightly because scan cells are usually larger than the nonscan cells they replace and the nets used for the scan signals occupy additional area.
- Design performance (speed) decreases marginally because of changes in the electrical characteristics of the scan cells that replace the nonscan cells.
- Global test signals that drive many sequential elements might require buffering to prevent electrical design rule violations.

The effects of adding scan circuitry vary, depending on the scan style and the semiconductor vendor library you use. For some scan styles, such as LSSD, introducing scan circuitry produces a negligible local change in performance.

By integrating DFT capabilities within synthesis, the DFT Compiler tool minimizes the overhead of scan circuitry based on performance, area, and electrical design rules.

ATPG and Internal Scan

For scan designs, ATPG tools generate input stimulus, for the primary inputs and pseudo-primary inputs, and expected responses, for the primary outputs and pseudo-primary outputs. The set of input stimulus and output response that includes primary inputs, primary outputs, pseudo-primary inputs, and pseudo-primary outputs is called a test pattern or scan pattern.

A test pattern represents many test vectors because

- The pseudo-primary input data must be serialized to be applied at the input of the scan chain
- The pseudo-primary output data must be serialized to be measured at the output of the scan chain

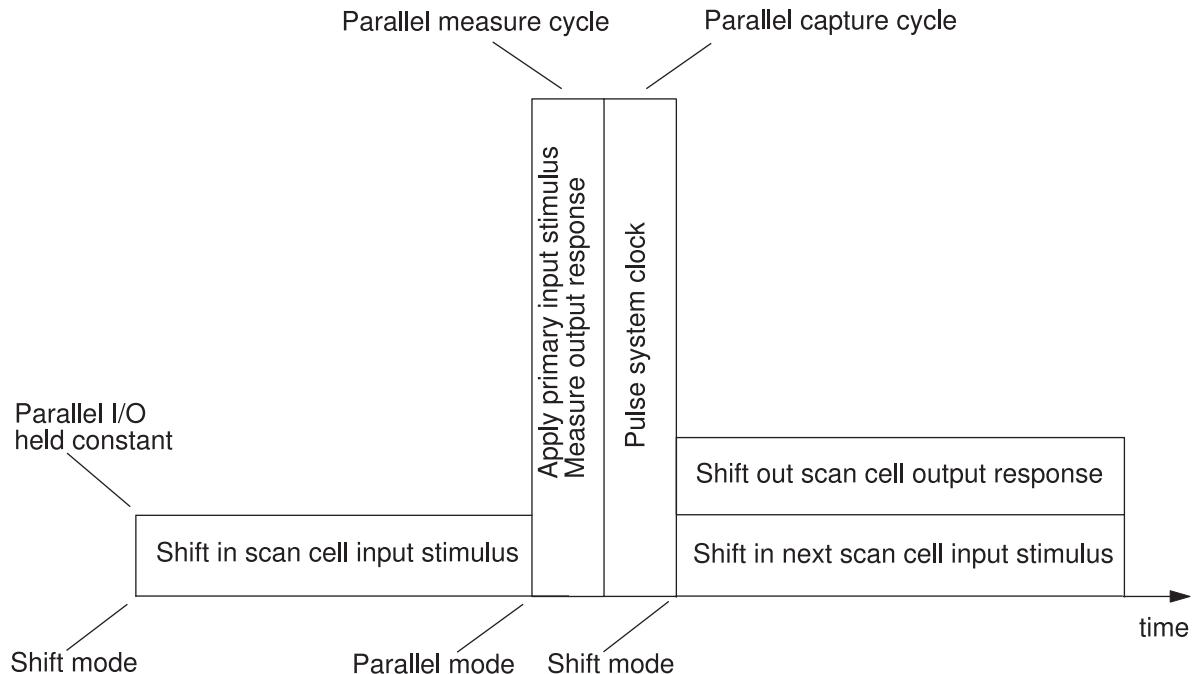
Applying Scan Patterns

Test patterns are applied to a scan-based design through the scan chains. The process is the same for full-scan and partial-scan designs.

Scan cells operate in one of two modes: parallel mode or shift mode. For the multiplexed flip-flop scan style shown in [Figure 4 on page 62](#), the mode is controlled by the scan-enable pin. In parallel mode, the input to each scan element comes from the combinational logic block. In shift mode, the input comes from the output of the previous scan cell or a scan-input port. Other scan styles work similarly.

The target tester applies a scan pattern, as illustrated in [Figure 5](#).

Figure 5 Scan Pattern Application Sequence



To apply a scan pattern, the target tester

1. Selects shift mode by setting the scan-enable port. This test signal is connected to all scan cells.
2. Shifts in the input stimulus for the scan cells (pseudo-primary inputs) at the scan-input ports.
3. Selects parallel mode by inverting the scan-enable port.
4. Applies the input stimulus to the primary inputs.
5. Checks the output response at the primary outputs after the circuit has settled and compares it to the expected fault-free response. This process is called parallel measure.
6. Pulses one or more clocks to capture the steady-state output response of the nonscan logic blocks into the scan cells. This process is called parallel capture.

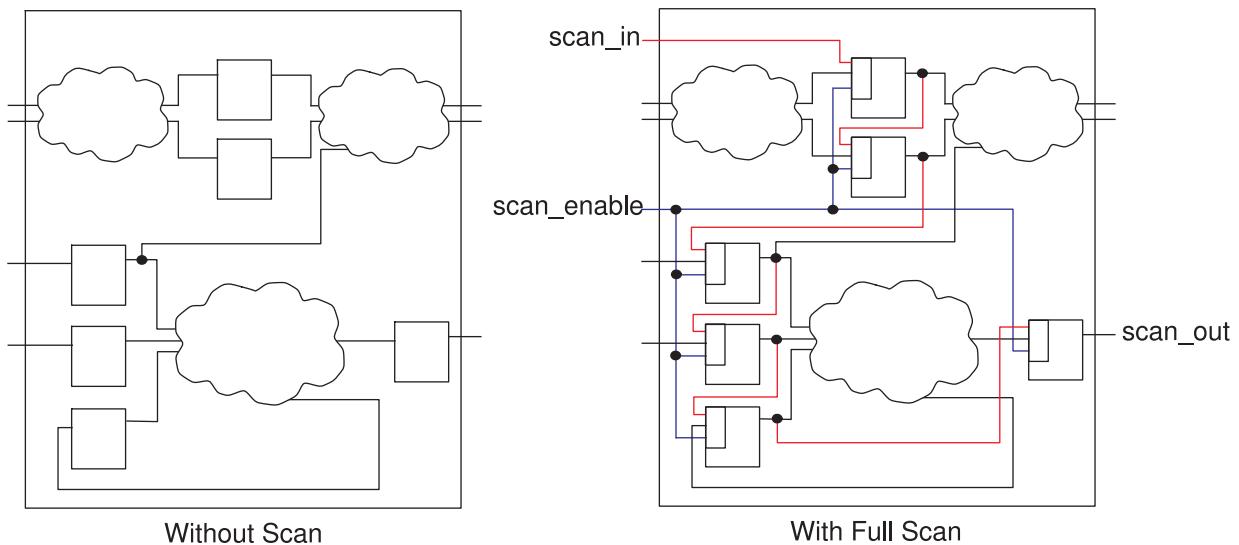
7. Selects shift mode by resetting the scan-enable port.
8. Shifts out the output response of the scan cells (pseudo-primary outputs) at the scan-output ports and compares the scan cell contents with the expected fault-free response.

Full-Scan Design

In the full-scan design technique, all sequential cells in your design are modified to perform a serial shift function. Sequential elements that are not scanned are treated as black-box cells (cells with unknown function).

Full scan divides a sequential design into combinational blocks, as shown in [Figure 6 on page 65](#). In the figure, clouds represent combinational logic and rectangles represent sequential logic. The full-scan diagram shows the scan path through the design.

Figure 6 Scan Path Through a Full-Scan Design



Through pseudo-primary inputs, the scan path enables direct control of inputs to all combinational blocks. Through pseudo-primary outputs, the scan path enables direct observability of outputs from all combinational blocks. You can use efficient combinational ATPG algorithms to achieve high fault coverage results on the full-scan design.

Test for System-On-A-Chip Designs

SoC Test provides a standards-based infrastructure for automatically incorporating a variety of structured and scalable DFT structures into system-on-a-chip (SoC) designs, all within the Synopsys synthesis environment. The value of SoC DFT directly parallels that of SoC functional design. By incorporating functional block test reuse, you can focus attention and effort on the integration of system components and optimization of the final test system.

Boundary Scan Design

Boundary scan is a DFT technique that simplifies printed circuit board testing using a standard chip-board test interface. The Institute of Electrical and Electronics Engineers (IEEE) has established the industry standard for this test interface. This standard is known as the IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Std 1149.1).

The boundary-scan technique is often referred to as JTAG. JTAG stands for Joint Test Action Group, the group that initiated the standardization of this test interface.

Boundary scan enables board-level testing by providing direct access to the input and output pads of the integrated circuits on a printed circuit board. Boundary scan modifies the I/O circuitry of individual ICs and adds control logic so the input and output pads of every boundary-scan IC can be joined to form a board-level serial scan chain.

The boundary-scan technique uses the serial scan chain to access the I/O ports of chips on a board. Because the scan chain is composed of the input and output pads of a chip's design, the chip's primary inputs and outputs are accessible on the board for applying and sampling data. Boundary scan supports the following board-level test functions:

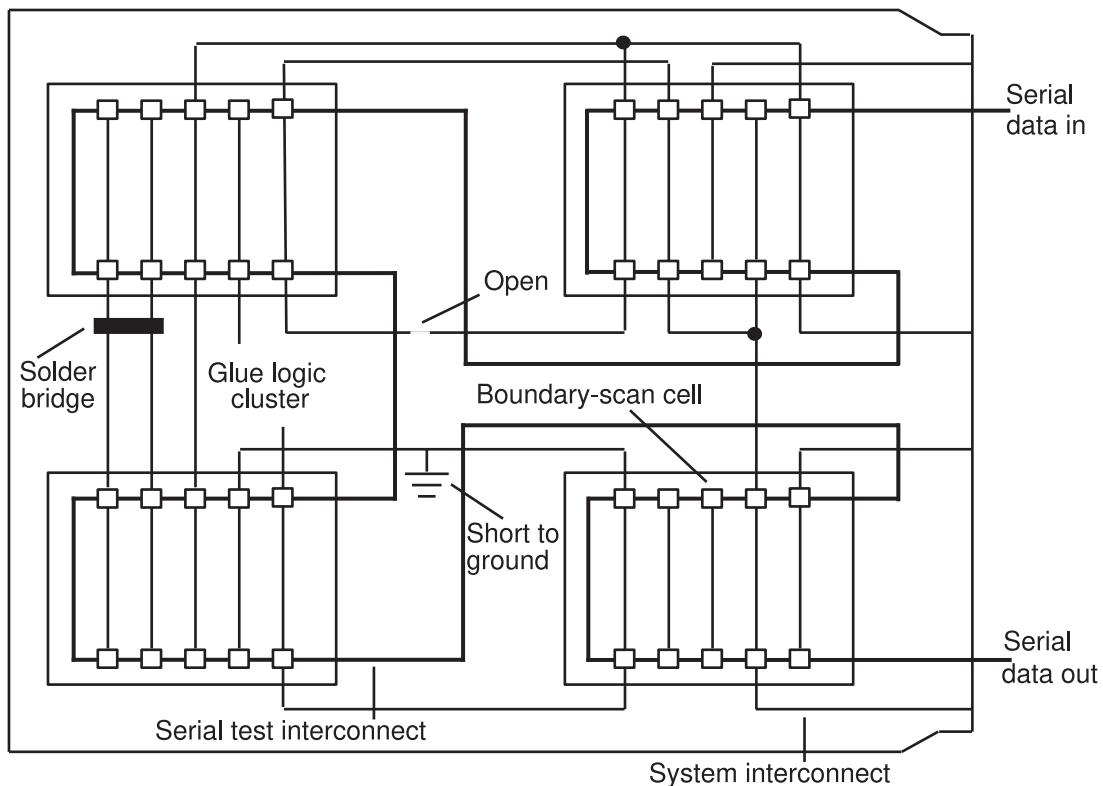
- Testing the interconnect wiring on a printed circuit board for shorts, opens, and bridging faults
- Testing clusters of non-boundary-scan logic
- Identifying missing, misoriented, or wrongly selected components
- Identifying fixture problems
- Limited testing of individual chips on a board

Note:

Although boundary scan addresses several board-test issues, it does not directly address chip-level testability. Combine chip-test techniques (such as internal scan) with boundary scan to provide testability at both the chip and board level.

[Figure 7](#) depicts a simple printed circuit board with several boundary-scan ICs and illustrates some of the failures that boundary scan can detect.

Figure 7 Board Testing With IEEE Std 1149.1 Boundary Scan



In the Synopsys design environment, the TestMAX DFT tool supports implementation of boundary-scan ports, interconnections, and control.

For more information about IEEE Std 1149.1 and IEEE Std 1149.6 boundary-scan, see TestMAX DFT Boundary Scan User Guide and the TestMAX DFT Boundary Scan Reference Manual.

4

Scan Styles

DFT Compiler supports a variety of scan styles. This chapter discusses each supported scan style.

This chapter includes the following topics:

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

Multiplexed Flip-Flop Scan Style

The multiplexed flip-flop scan style uses a multiplexed data input to provide scan shift capability. In functional mode, the scan-enable signal, acting as the multiplexer select line, selects the system data input. During scan shift, the scan-enable signal selects the scan data input. The scan data input comes from either the scan-input port or the scan-output pin of the previous cell in the scan chain.

The following test pins are required on a multiplexed flip-flop equivalent cell:

- Scan-input
- Scan-enable
- Scan-output (can be shared with a functional output pin)

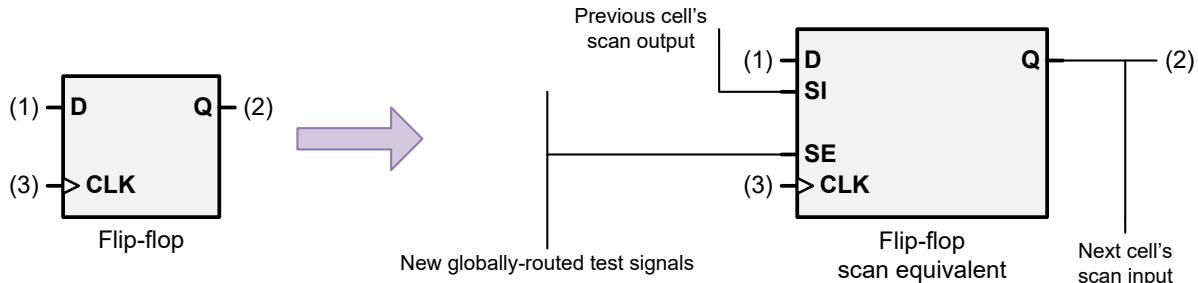
Test pins are identified in the `test_cell` group of the cell description in the logic library. For information on modeling test cells in your logic library, see the Library Compiler user guides.

Multiplexed flip-flop is the scan style most commonly supported in logic libraries. Most libraries provide multiplexed flip-flop equivalents for D flip-flops.

Flip-Flop Equivalents

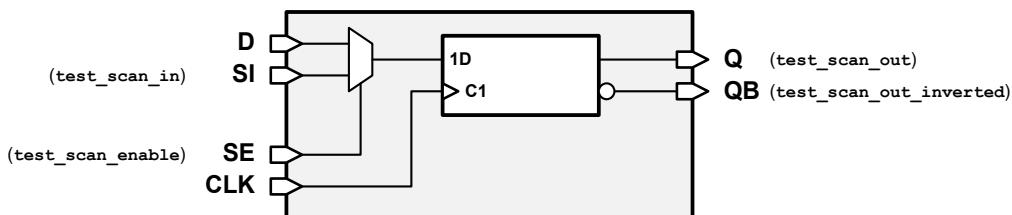
[Figure 8](#) shows an example of a D flip-flop before and after scan substitution, using the multiplexed flip-flop scan style. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the scan-enable pin is SE, and the scan-out pin is shared with the functional output pin Q.

Figure 8 D Flip-Flop After Multiplexed Scan Cell Substitution



[Figure 9](#) shows the generic model used by DFT Compiler for multiplexed scan flip-flops. The library model's test_cell ports are shown in parentheses.

Figure 9 Default Multiplexed Flip-Flop Scan Cell



[Table 2](#) lists the signal-type pin connections for the multiplexed scan flip-flop cell.

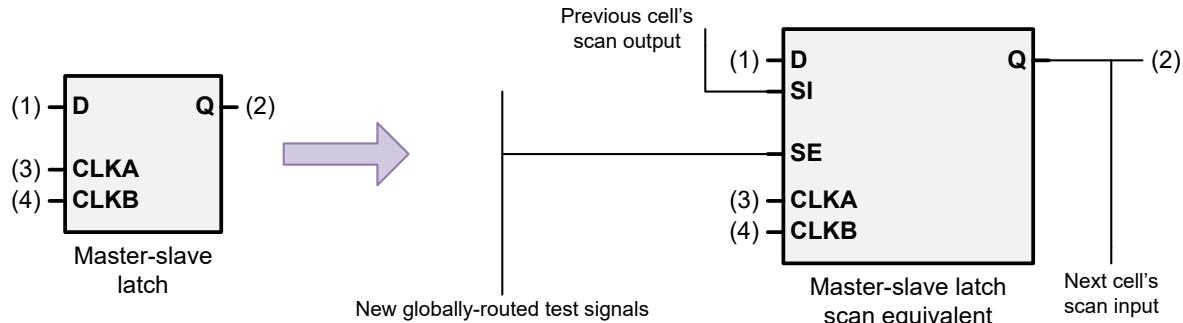
Table 2 Signal-Type Pin Connections for Multiplexed Scan Flip-Flop

Pin	Signal type
SI	ScanDataIn
SE	ScanEnable
Q	ScanDataOut
QB	ScanDataOut

Master-Slave Latch Equivalents

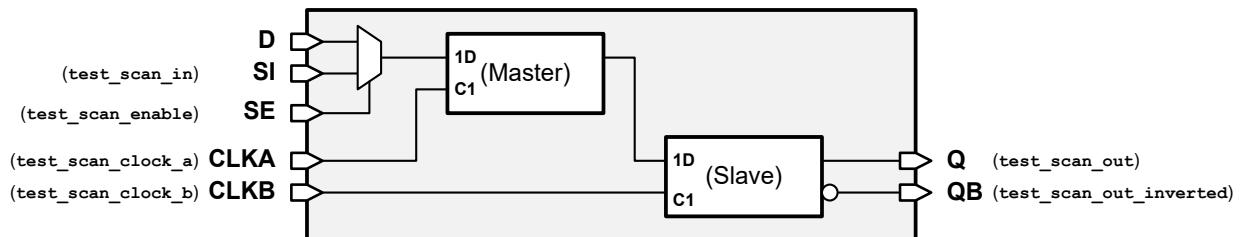
[Figure 10](#) shows an example of a master-slave latch before and after scan substitution, using the multiplexed scan style. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the scan-enable pin is SE, and the scan-out pin is shared with the functional output pin Q.

Figure 10 Latch After Multiplexed Scan Cell Substitution



[Figure 11](#) shows the logic diagram for the generic model of a multiplexed master-slave latch. The library model's test_cell ports are shown in parentheses.

Figure 11 Default Multiplexed Master-Slave Latch Scan Cell



[Table 3](#) is the truth table for this model. Note that the master clock CLKA and slave clock CLKB are nonoverlapping, as shown in [Figure 12](#).

Table 3 Truth Table for Multiplexed Master-Slave Latch Scan Cell

D	SI	SE	CLKA	CLKB	Q	QB	Mode
0	X	0	↑↓	↑↓	0	1	Functional
1	X	0	↑↓	↑↓	1	0	Functional

D	SI	SE	CLKA	CLKB	Q	QB	Mode
X	0	1	↑↓	↑↓	0	1	Scan
X	1	1	↑↓	↑↓	1	0	Scan
X	X	X	0	0	Q	QB	Either

↑↓ = Positive pulse

Note: The master clock (CLKA) pulse precedes the slave clock (CLKB) pulse, and the clocks are nonoverlapping.

X = Don't care

Figure 12 Nonoverlapping Master-Slave Scan Clocks

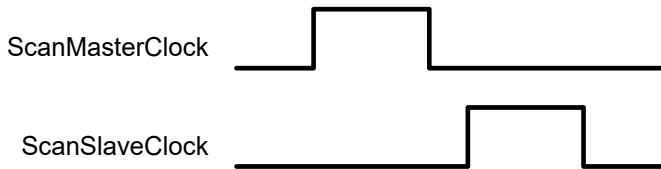


Table 4 lists the signal-type pin connections for the multiplexed master-slave latch scan cell.

Table 4 Signal-Type Pin Connections for Multiplexed Master-Slave Latch Cell

Pin	Signal type
SI	ScanDataIn
SE	ScanEnable
Q	ScanDataOut
QB	ScanDataOut

Multiplexed Flip-Flop Scan Style Characteristics

The multiplexed flip-flop scan style has the following general characteristics:

- Additional delay caused by the multiplexer in the functional path.
- Low cell area overhead. A multiplexed D-type flip-flop is typically 15 percent to 30 percent larger than a standard D-type flip-flop.
- Low routing overhead due to one additional global scan-enable signal. Skew is not critical on this signal.
- A minimum of one additional I/O port (scan-enable). You might not need an additional I/O port for scan-in or scan-out if you can multiplex these functions with existing functional ports in your design.
- Typically used with edge-triggered design styles.

Clocked-Scan Scan Style

The clocked-scan scan style uses a separate dedicated edge-triggered test clock to provide scan shift capability. In functional mode, the system clock is active and system data is clocked into the cell. During scan shift, the test clock is active and scan data is clocked into the cell.

The following test pins, identified in the `test_cell` group of the scan cell description in the logic library, are required on a clocked-scan cell:

- Scan-input
- Test-clock
- Scan-output (can be shared with a functional output pin)

DFT Compiler supports clocked-scan cells for both flip-flops and latches.

Flip-Flop Equivalents

[Figure 13](#) shows an example of a D flip-flop before and after scan substitution with the clocked-scan scan style. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the dedicated edge-triggered test clock pin is SCLK, and the scan-out pin is shared with the functional output pin Q.

Figure 13 D Flip-Flop After Clocked-Scan Cell Substitution

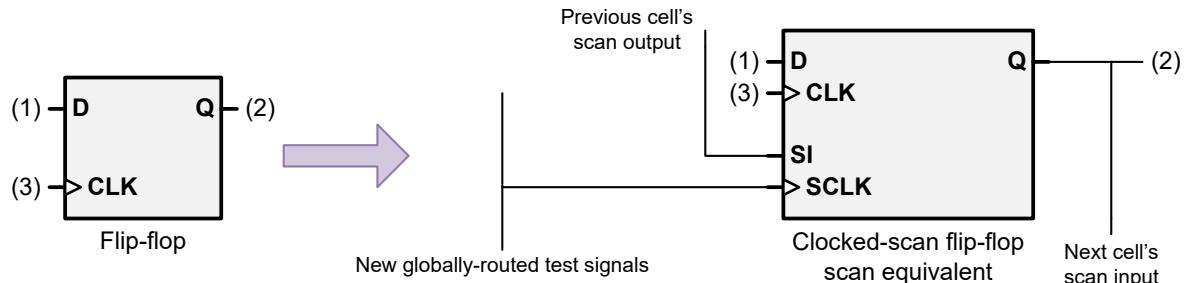


Figure 14 shows the generic model used by DFT Compiler for clocked-scan flip-flops. The library model's test_cell ports are shown in parentheses.

Figure 14 Default Clocked-Scan Flip-Flop Scan Cell

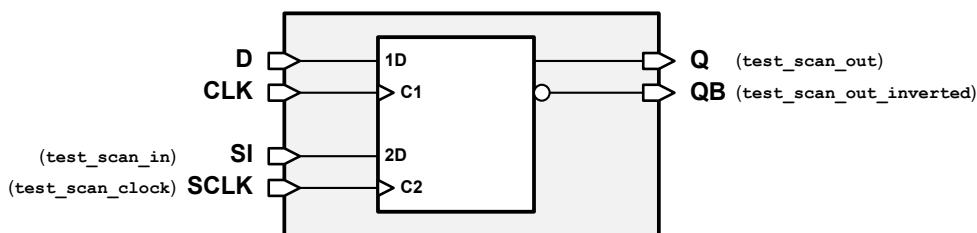


Table 5 is the truth table for this model.

Table 5 Truth Table for Clocked-Scan Flip-Flop Cell

D	SI	SCLK	CLK	Q	QB	Mode
0	X	0		0	1	Functional
1	X	0		1	0	Functional
X	0		0	0	1	Scan
X	1		0	1	0	Scan
X	X	0/1	0	Q	QB	Either

-  = Positive pulse
-  = Rising edge of clock
- X = Don't care

Table 6 lists the signal-type pin connections for the clocked-scan flip-flop cell.

Table 6 Signal-Type Pin Connections for Clocked-Scan Flip-Flop

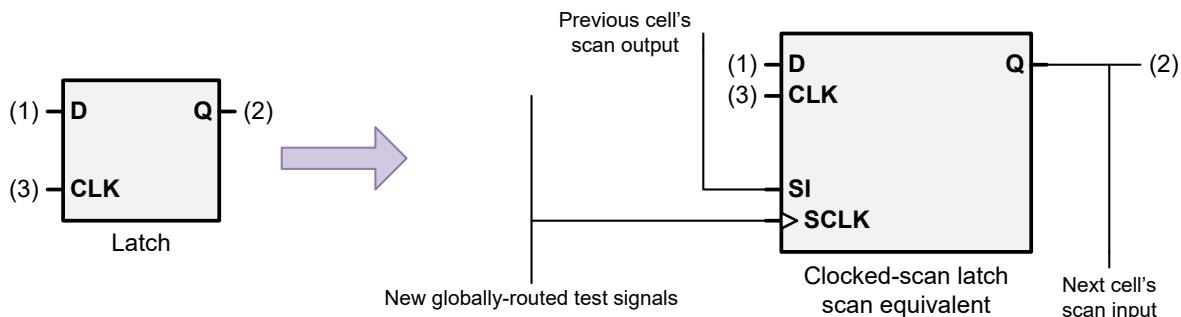
Pin	Signal type
SI	ScanDataIn
SCLK	ScanMasterClock
Q	ScanDataOut
QB	ScanDataOut

Latch Equivalents

Clocked-scan cells for latches are level sensitive in functional mode but are edge triggered during scan shift.

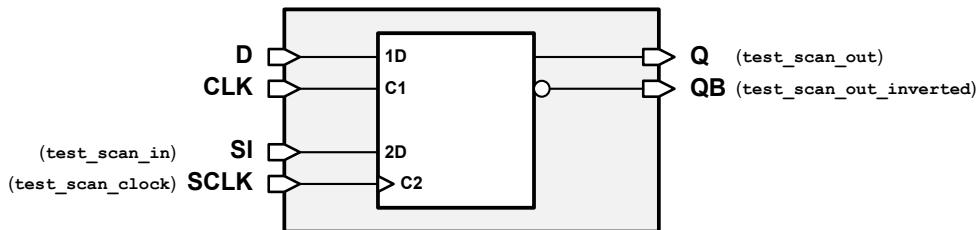
[Figure 15](#) shows an example of a latch before and after scan substitution with the clocked-scan scan style. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the dedicated edge-triggered test clock pin is SCLK, and the scan-out pin is shared with the functional output pin Q.

Figure 15 Latch After Clocked-Scan Cell Substitution



[Figure 16](#) shows the logic diagram of the default model used by DFT Compiler for a clocked-scan latch. The library model's test_cell ports are shown in parentheses.

Figure 16 Default Clocked-Scan Latch Cell



[Table 7](#) is the truth table for this model.

Table 7 Truth Table for Clocked-Scan Latch Cell

D	SI	SCLK	CLK	Q	QB	Mode
0	X	0		0	1	Functional
1	X	0		1	0	Functional
X	0		0	0	1	Scan
X	1		0	1	0	Scan
X	X	0/1	0	Q	QB	Either

= Positive pulse

= Rising edge of clock

X = Don't care

[Table 8](#) lists the signal-type pin connections for the clocked-scan latch cell.

Table 8 Signal-Type Pin Connections for Clocked-Scan Latch Cell

Pin	Signal type
SI	ScanDataIn
SCLK	ScanMasterClock
Q	ScanDataOut
QB	ScanDataOut

Clocked-Scan Scan Style Characteristics

Characteristics of the clocked-scan scan style include the following:

- It has negligible performance overhead.
- Low cell area overhead. A clocked-scan cell is typically 15 percent to 30 percent larger than a standard D-type flip-flop.
- Moderate increased routing overhead due to an additional global test clock signal. This clock signal is an edge-triggered clock signal, and skew must be managed to avoid hold violations along the scan path.
- Logic libraries supporting this scan style typically have both flip-flop and latch-equivalent cells.
- It is well suited to use in partial-scan designs. A dedicated test clock provides a mechanism for easily maintaining the state of nonscan cells during scan shift.
- It supports latches with asynchronous preset or clear pins.
- It is typically used with edge-triggered design styles.

LSSD Scan Style

DFT Compiler supports three variations of the LSSD scan style, depending on the type of sequential cell being scan-replaced:

- Single-latch
 - Scan-replaces standard latch cells with LSSD scan cells
- Double-latch
 - Scan-replaces master-slave latch cells with LSSD scan cells
- Clocked
 - Scan-replaces edge-triggered flip-flops with LSSD-compatible flip-flop scan cells

These variations can be mixed in a single design.

Both the single-latch and double-latch variations use a classical LSSD scan cell for scan replacement, which consists of two latches acting as a master-slave pair. The master latch has dual input ports and can latch either *functional* data or *scan* data. In functional mode, the system master clock input controls the functional data input. In scan mode, the test master clock input controls the transfer of data from the scan data input to the master

latch. The slave clock input controls the transfer of data from the master latch to the slave latch.

The clocked variation uses a special LSSD-compatible flip-flop cell that operates as a flip-flop during functional mode, but operates as an LSSD cell during scan shift.

The following test pins, identified in the `test_cell` group of the scan cell description in the logic library, are required on an LSSD scan cell:

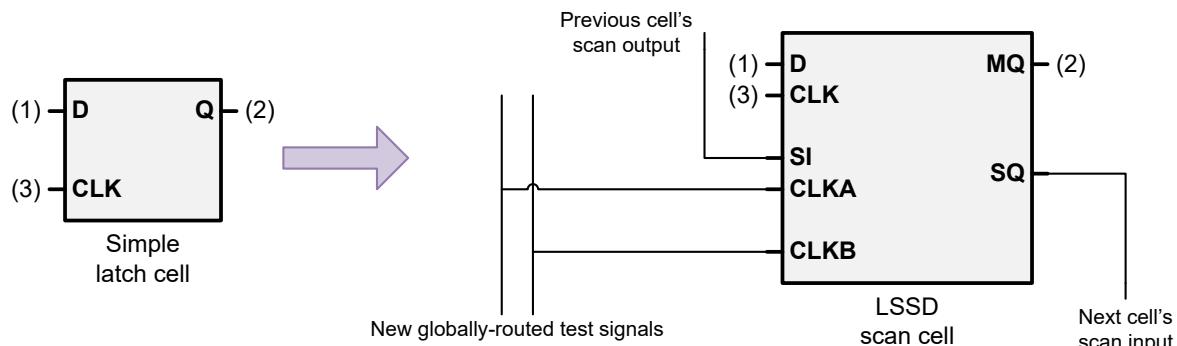
- Scan-input
- Test master-clock
- Test slave-clock (except for double-latch LSSD)
- Scan-output (can be shared with a functional output pin)

Single-Latch LSSD

In the single-latch variation, DFT Compiler replaces simple latches in your design with LSSD scan cells.

[Figure 17](#) shows a latch before and after scan substitution, using the single-latch LSSD scan style. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the master test clock is CLKA, the slave test clock is CLKB, the system master clock is CLK, and the scan-out pin is SQ.

Figure 17 Latch After LSSD Single-Latch Scan Replacement



[Figure 18](#) shows the generic model used by DFT Compiler for single-latch LSSD. The library model's `test_cell` ports are shown in parentheses.

Figure 18 Default Single-Latch LSSD Scan Cell

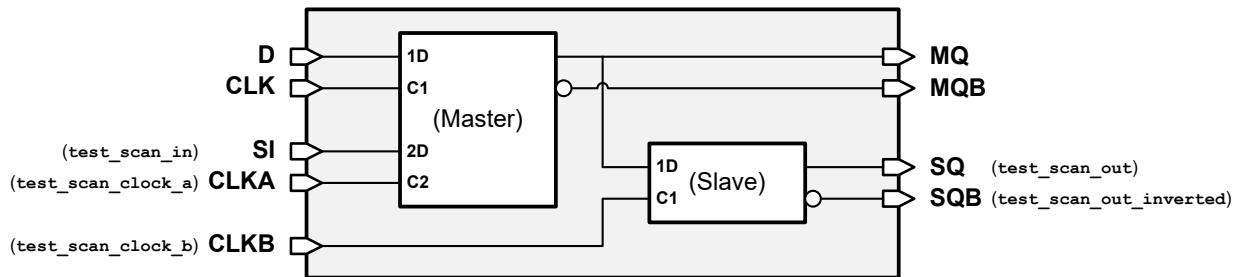


Table 9 is the truth table for the LSSD master latch.

Table 9 Truth Table for LSSD Master Latch

D	SI	CLKA	CLK	MQ	Mode
0	X	0	□	0	Functional
1	X	0	□	1	Functional
X	0	□	0	0	Scan
X	1	□	0	1	Scan
X	X	0	0	MQ	Either

□ = Positive pulse

X = Don't care

Table 10 lists the signal-type pin connections for the LSSD master latch.

Table 10 Signal-Type Pin Connections for LSSD Master Latch

Pin	Signal type
SI	ScanDataIn
CLKA	ScanMasterClock

Table 11 is the truth table for the LSSD slave latch.

Table 11 Truth Table for LSSD Slave Latch

MQ	CLKB	SQ	Mode
0		0	Scan
1		1	Scan
X	0	SQ	Scan

= Positive pulse

X = Don't care

Table 12 lists the signal-type pin connections for the LSSD slave latch.

Table 12 Signal-Type Pin Connections for LSSD Slave Latch

Pin	Signal type
CLKB	ScanSlaveClock
SQ	ScanDataOut

In functional mode, the master latch of the scannable LSSD cell functions like the original latch in the design. When the level-sensitive clock CLK is asserted, data is transferred from input pin D to master latch output pin MQ. During functional operation, the scan shift master clock CLKA remains quiet.

In scan shift mode, the two nonoverlapping test clocks CLKA and CLKB are used to shift data from the scan input through both master and slave latches to the slave output pin SQ. [Figure 12](#) illustrates the nonoverlapping master-slave test clocks.

Single-Latch LSSD Scan Style Characteristics

The characteristics of the single-latch LSSD variation are the following:

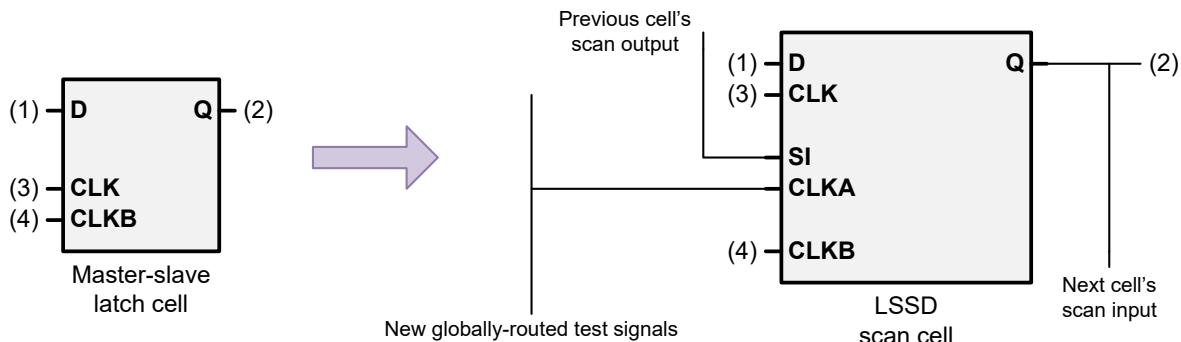
- Negligible performance overhead.
- High cell area overhead. Replacing a simple latch with an LSSD cell can increase sequential logic area by 100 percent or more, because the new cell adds a slave latch.

- Significant increased routing overhead due to two additional global master-slave test clock signals. However, master-slave test clocks do not require as much careful skew control as edge-triggered clock signals.
- It supports latches with asynchronous preset or clear pins.
- It is well suited for use in partial-scan designs because of the dedicated test clocks.

Double-Latch LSSD

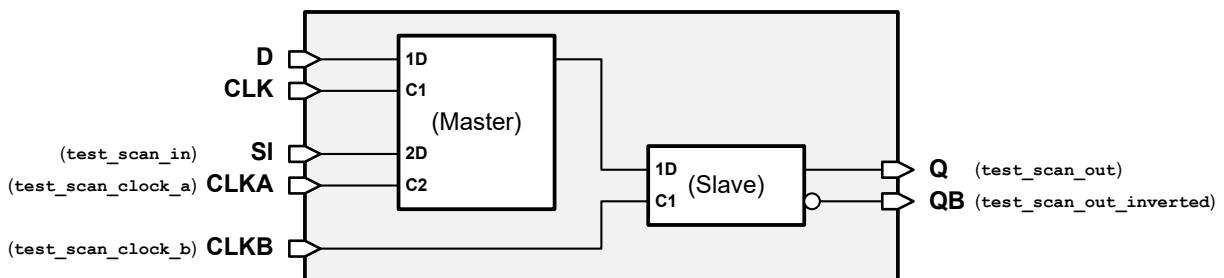
In the double-latch LSSD variation, DFT Compiler replaces master-slave latch pair cells with LSSD scan cells. [Figure 19](#) shows a cell before and after scan substitution. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the master test clock is CLKA, the slave test clock is CLKB, and the scan-out pin is shared with the functional output pin Q.

Figure 19 Master-Slave Latch After LSSD Double-Latch Scan Replacement



[Figure 20](#) shows the generic model used by DFT Compiler for double-latch LSSD. The library model's test_cell ports are shown in parentheses. The truth tables for this model are the same as for the single-latch LSSD model (see [Table 9](#) and [Table 11](#)).

Figure 20 Default Double-Latch LSSD Scan Cell



In functional mode, the master and slave latches in the LSSD cell take over the function of both the master and the slave latches in the original flip-flop. Data from input pin D is clocked through the latch pair using master-slave clocking on the master clock CLK and slave clock CLKB inputs. Data is clocked out to the slave output Q. A master latch output cannot be used in the double-latch LSSD scan style.

In scan shift mode, two-phase, nonoverlapping master-slave test clocks CLKA and CLKB are applied to the clock inputs to shift data from the scan input through both master and slave latches to the slave output pin Q.

Note that in the double-latch LSSD variation, the slave clock input CLKB is used in both functional mode and scan shift mode. In the single-latch LSSD variation, the slave clock input CLKB is used only in scan shift mode.

Table 13 is the truth table for this model.

Table 13 Truth Table for Double-Latch LSSD Scan Cell

D	SI	CLK	CLKA	CLKB	Q	QB	Mode
0	X		0		0	1	Functional
1	X		0		1	0	Functional
X	0	0			0	1	Scan
X	1	0			1	0	Scan
X	X	0	0	0	Q	QB	Either

= Positive pulse
 = Rising edge of clock
X = Don't care

Note: The master clock CLK or CLKA pulse precedes the slave clock CLKB pulse, and the clocks are nonoverlapping, as shown in [Figure 12 on page 71](#).

Table 14 lists the signal-type pin connections for the clocked LSSD scan cell.

Table 14 Signal-Type Pin Connections for Double-Latch LSSD Scan Cell

Pin	Signal type
SI	ScanDataIn

Pin	Signal type
CLKA	ScanMasterClock
CLKB	ScanSlaveClock
Q	ScanDataOut
QB	ScanDataOut

Double-Latch LSSD Scan Style Characteristics

The characteristics of the double-latch LSSD variation are as follows:

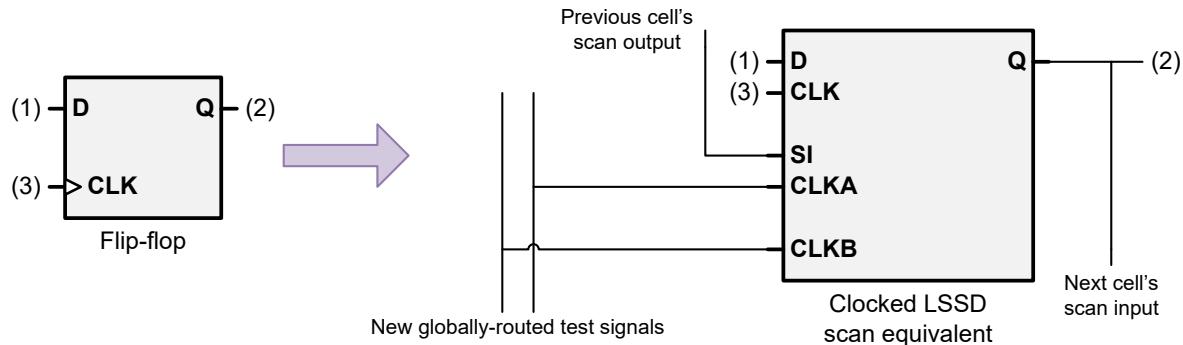
- Negligible performance overhead.
- Low cell area overhead (15 percent to 30 percent).
- Moderate increased routing overhead due to one additional global master test clock signal. However, master-slave test clocks do not require as much careful skew control as edge-triggered clock signals.
- Support for latches with asynchronous preset or clear pins.
- Suitability for partial-scan designs because of the dedicated test clocks.

Clocked LSSD

In the clocked LSSD variation, DFT Compiler replaces edge-triggered flip-flops with LSSD-compatible flip-flop cells that use the standard LSSD master-slave test clocks for scan shift. In functional mode, a clocked LSSD cell functions as an edge-triggered cell with the system clock active and system data clocked into the cell. In scan mode, two-phase, nonoverlapping master-slave test clocks are applied to the master test and slave test clock inputs to shift data from the scan-input pin to the scan-output pin.

[Figure 21](#) shows an edge-triggered flip-flop before and after clocked LSSD scan cell substitution. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the master test clock is CLKA, the slave test clock is CLKB, and the scan-out pin is shared with the functional output pin Q. The functional system clock is CLK.

Figure 21 Edge-Triggered Flip-Flop After Clocked LSSD Scan Replacement

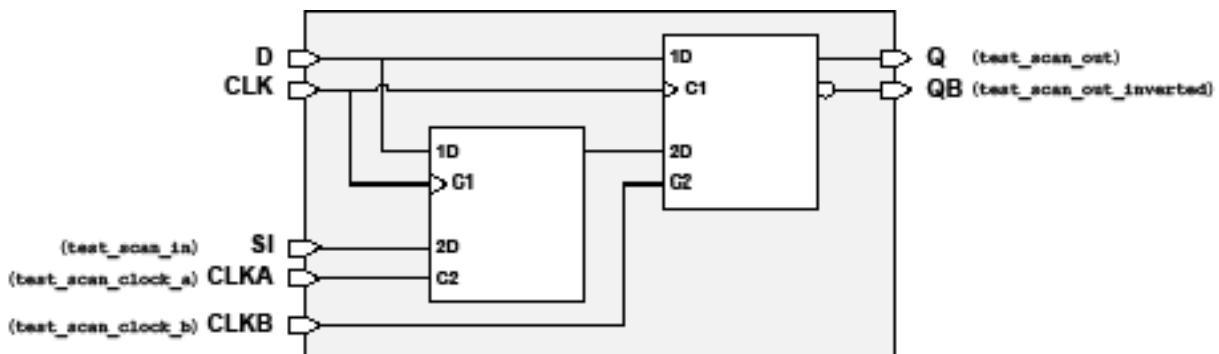


Different implementations of the clocked LSSD scan cell can be modeled by use of a state table. [Figure 22](#) shows the generic model used by DFT Compiler for clocked LSSD if the logic library does not include an explicit state table model. The library model's test_cell ports are shown in parentheses.

Note:

The generic model provides backward compatibility with previous versions of DFT Compiler. This model is compatible with either A-before-B clocking during shift or B-before-A clocking during shift if the right protocol is used. If the generic model is not adequate, it is strongly recommended that you use a logic library with an explicit state table model for the clocked LSSD scan style.

Figure 22 Default Clocked LSSD Scan Cell



[Table 15](#) is the truth table for this model.

Table 15 Truth Table for Clocked LSSD Scan Cell

D	SI	CLKA	CLKB	CLK	Q	QB	Mode
0	X	0	0		0	1	Functional

D	SI	CLKA	CLKB	CLK	Q	QB	Mode
1	X	0	0		1	0	Functional
X	0			0	0	1	Scan
X	1			0	1	0	Scan
X	X	0	0	0/1	Q	QB	Either

= Positive pulse
 = Rising edge of clock
X = Don't care

Note: The master clock CLKA pulse precedes the slave clock CLKB pulse, and the clocks are nonoverlapping, as shown in [Figure 12 on page 71](#).

[Table 16](#) lists the signal-type pin connections for the clocked LSSD scan cell.

Table 16 Signal-Type Pin Connections for Clocked LSSD Scan Cell

Pin	Signal type
SI	ScanDataIn
CLKA	ScanMasterClock
CLKB	ScanSlaveClock
Q	ScanDataOut
QB	ScanDataOut

Clocked LSSD Scan Style Characteristics

The characteristics of the clocked LSSD variation are as follows:

- Negligible performance overhead.
- Moderate cell area overhead. A scan cell is 40 percent to 80 percent larger than a flip-flop.

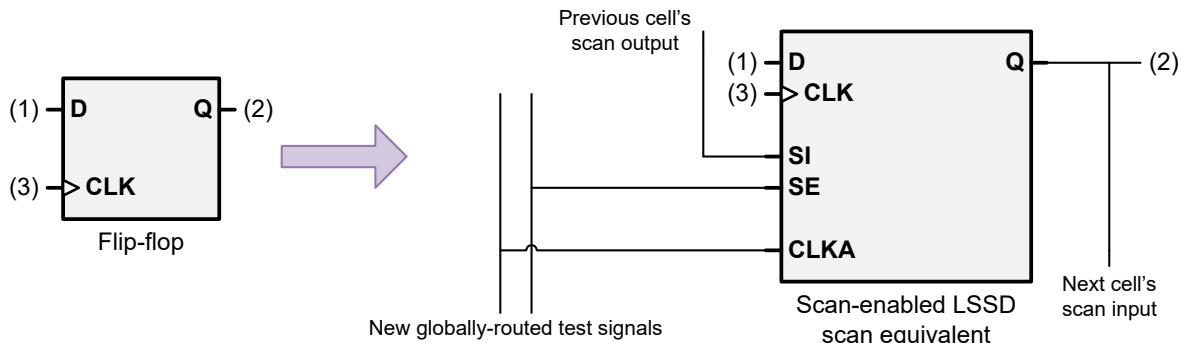
- Significant increased routing overhead due to two additional global master-slave test clock signals. However, master-slave test clocks do not require as much careful skew control as edge-triggered clock signals.
- Suitability for partial-scan designs because of the dedicated test clocks.

Scan-Enabled LSSD Style

The scan-enabled LSSD scan style uses a scan-enable signal to control the behavior of the scan cell. In functional mode, the de-asserted scan-enable signal causes the cell to behave like an edge-triggered flip-flop. During scan shift, the asserted scan-enable signal causes the cell to behave like a master-slave latch scan cell.

[Figure 23](#) shows a flip-flop before and after scan substitution, using the scan-enabled LSSD style. The pin connection mappings are shown in parentheses. In this example, the scan-in pin is SI, the scan-enable pin is SE, the master test clock is CLKA, the slave test clock is shared with the functional clock pin CLK, and the scan-out pin is shared with the functional output pin Q.

Figure 23 Edge-Triggered Flip-Flop After Scan-Enabled LSSD Scan Replacement



[Figure 24](#) shows the generic model used by DFT Compiler for scan-enabled LSSD. The library's test_cell ports are shown in parentheses.

Figure 24 Default Scan-Enabled LSSD Cell

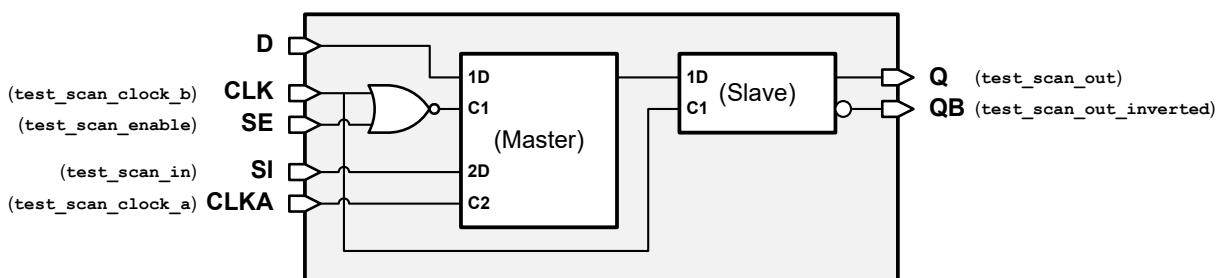


Table 17 is the truth table for this model.

Table 17 Truth Table for Scan-Enabled LSSD Scan Cell

D	SI	SE	CLKA	CLK	Q	QB	Mode
0	X	0	0		0	1	Functional
1	X	0	0		1	0	Functional
X	0	1			0	1	Scan
X	1	1			1	0	Scan
X	X	X	0	0/1	Q	QB	Either

 = Positive pulse
 = Rising edge of clock
X = Don't care

Note: The master clock CLKA pulse precedes the slave clock CLK pulse, and the clocks are nonoverlapping, as shown in [Figure 12 on page 71](#).

Table 18 lists the signal-type pin connections for the scan-enabled LSSD cell.

Table 18 Signal-Type Pin Connections for Scan-Enabled LSSD Cell

Pin	Signal type
SI	ScanDataIn
SE	ScanEnable
CLKA	ScanMasterClock
Q	ScanDataOut
QB	ScanDataOut

Note:

The functional clock pin CLK should be defined as a master clock signal, but with slave clock timing.

In functional mode, the primary clock signal CLK is provided to the master-slave latches in inverted/noninverted form, resulting in edge-triggered operation. In scan shift mode, the

primary clock is blocked from the master latch, and CLKA is used as a master clock to clock scan data into the master latch from the SI pin instead. The CLK signal is still used as a slave clock to transfer data from the master latch to the slave latch.

The scan-enabled LSSD style is similar to the clocked LSSD style, except that the scan-enable signal is used to re-purpose an existing clock signal.

Scan-Enabled LSSD Scan Style Characteristics

The characteristics of the scan-enabled LSSD scan style are as follows:

- Negligible performance overhead.
- Moderate area overhead. A scan cell is 30 percent to 60 percent larger than a flip-flop.
- Significant increased routing overhead due to two additional global test signals. However, minimal skew control is needed on the scan-enable signal, and master-slave test clocks do not require as much careful skew control as edge-triggered clock signals.
- Suitability for partial-scan designs because of the dedicated test clocks.

5

Scan Design Requirements

This chapter discusses the scan design requirements for test ports, test timing, test clocks, and test protocols in design-for-test strategies.

This chapter includes the following topics:

- [Test Port Requirements](#)
 - [Test Timing Requirements](#)
 - [Test Clock Requirements](#)
 - [Test Protocol Requirements](#)
-

Test Port Requirements

One goal of the scan design technique is to minimize the number of physical I/O pins (corresponding to logic ports) allocated for test purposes. The following I/O ports are required by the different scan styles:

- Scan In

This input port, required by all scan styles, drives a serial scan chain. If your design has multiple scan chains, each scan chain must have its own scan-in port. On the clock transition when the scan cells are shifted, the data at each scan-in port is clocked into the first cell in the corresponding scan chain.

In some cases, you can use a functional data input port as a scan-in port, thus saving an additional test port in your design. Confirm that this configuration is acceptable to your ASIC vendor.

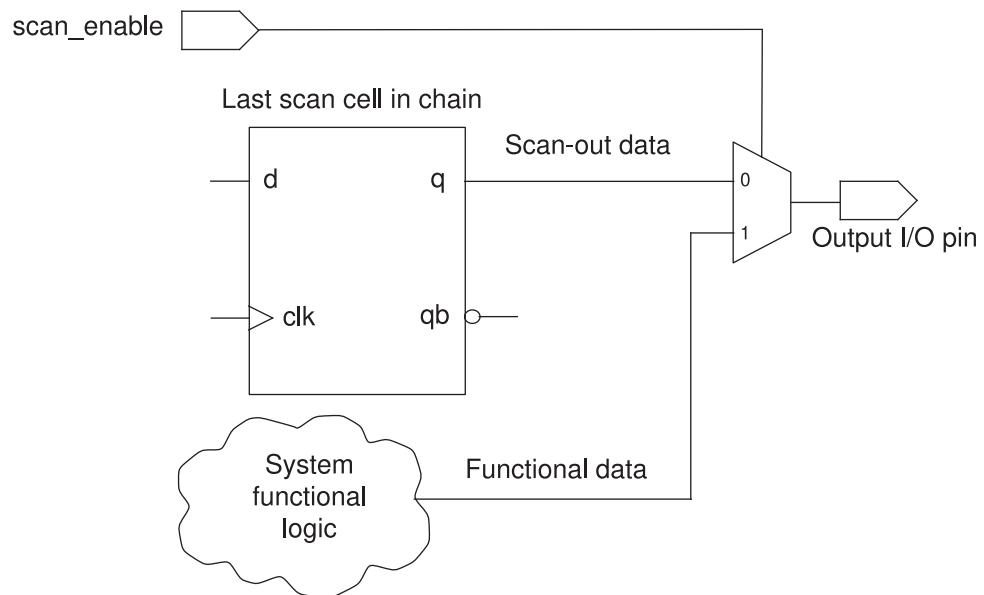
- Scan Out

This output port, required by all scan styles, sees the value at the end of a serial scan chain. If your design has multiple scan chains, each scan chain must have a scan-out port. On the clock transition when the scan cells are shifted, the data at each scan-out port changes to reflect the value held in the last cell of the corresponding scan chain.

You can use a functional data output pin as a scan-out port in one of two ways:

- The scan-out pin of the last scan cell in a scan chain is connected directly (or through buffers or inverters) to a functional output port.
- Use the scan-enable signal to multiplex scan-out data and functional data at a pin, as shown in [Figure 25](#). DFT Compiler inserts this multiplexing logic automatically when you define a functional output as a scan-output port before routing the scan chains.

Figure 25 Multiplexing Scan-Out Data and Functional Data



- Scan Clock A (master)

This input port, required by all LSSD variations, is a level-sensitive signal that controls the latching of serial scan data into the master latch of LSSD.

- Scan Clock B (slave)

This input port is optional for double-latch LSSD but is required for other types of LSSD. The port is a level-sensitive signal that controls latching of data from the master latch to the slave latch of LSSD cells.

- Scan Enable

This input port, required by the multiplexed flip-flop scan style, configures scan cells for their serial shift mode. For all scan styles, you can use a scan-enable input to

control the behavior of other devices, such as disabling logic for three-state drivers or multiplexing logic between individual scan cells or between scan cells and I/O ports.

The scan-enable signal can be active high or active low, if the sense is consistent throughout the design.

- Test Scan Clock

This input port, required by the clocked-scan scan style, is an edge-sensitive signal that controls the clocking of serial scan data.

Table 19 shows the additional I/O ports used by each scan style. The term *sharable* means that the port is required and can be shared. The I/O ports in the table are named according to their function, but you can assign another name.

Table 19 Additional I/O Port Requirements for Scan Styles

I/O signal function	Multiplexed flip-flop	Clocked-scan	LSSD
Scan in	Sharable	Sharable	Sharable
Scan out	Sharable	Sharable	Sharable
Scan clock A	Don't use	Don't use	Required
Scan clock B	Don't use	Don't use	Required for single-latch; optional for double-latch
Scan enable	Required	Required for multiplexed outputs and three-state disabling	Required for multiplexed outputs and three-state disabling
Test scan clock	Don't use	Required	Don't use

Test Timing Requirements

The default timing in DFT Compiler is appropriate for most pre-clock measure (strobe before clock) implementations. End-of-cycle measure (strobe after clock) timing can be manually configured, but is strongly discouraged. For more information, see [Setting Timing Variables on page 582](#).

Your semiconductor vendor might have timing requirements that are different from those set by default in DFT Compiler and those recommended for use with TestMAX ATPG. At

the start of the design process, discuss test timing requirements with your semiconductor vendor. Understand semiconductor vendor test timing requirements before running DFT Compiler.

Semiconductor vendors commonly specify the following timing parameters:

- Test period
- Input timing
- Bidirectional timing
- Output strobe timing
- Clocking requirements

Test Clock Requirements

Each scan style imposes requirements on the scan chain clocks. When you specify design constraints involving clock control, the default settings are the safest. If you override the defaults, consider the issues in this topic.

Clock Requirements in Edge-Sensitive Scan Shift Styles

Scan chain clocking issues for edge-sensitive scan shift are different from those for LSSD scan styles. Edge-sensitive scan shift scan styles include multiplexed flip-flop and clocked scan.

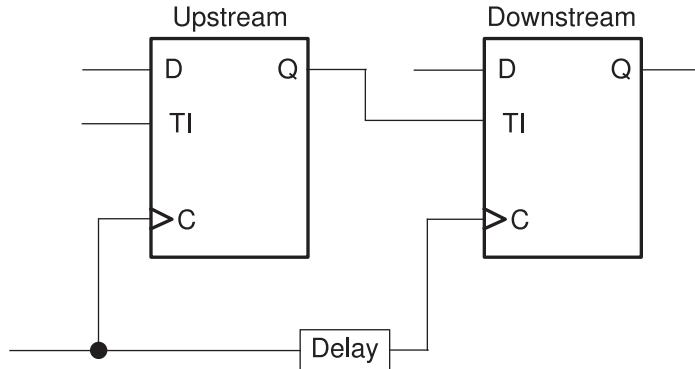
To avoid timing-related problems in edge-sensitive scan shift styles, you must align the clocks to all flip-flops in each chain. The safest way to align the clocks is to make all flip-flops share a common clock with the same phase and no skew. Under these conditions, if the path from the clock-to-Q to the test input of the next stage is longer than the hold time of that stage, the shift occurs reliably on the active edge of the clock. The clock must have a controlled path from an external primary input so the shift function can be performed on demand in any state.

Skew Issues

Skew can arise from clock tree buffers, but a more serious source of skew is gating and multiplexing logic that is inserted on clock lines to obtain supposedly congruent clocks for the scan chain. Gating logic is not necessarily discouraged, but its introduction can cause hold time problems that must be addressed. Fortunately, the Synopsys environment includes timing analysis that detects hold time problems.

Construct scan chains to reduce the risk of skew outside of tolerable limits. Skew can result in hold time violations if it delays downstream flip-flop clocks longer than it delays upstream flip-flop clocks. This condition is illustrated in [Figure 26](#).

Figure 26 Hold Time Violation Caused by Delayed Downstream Clock



If the skew is bad enough, two flip-flops can share a single state, causing data to fall through and resulting in incorrect scan operation.

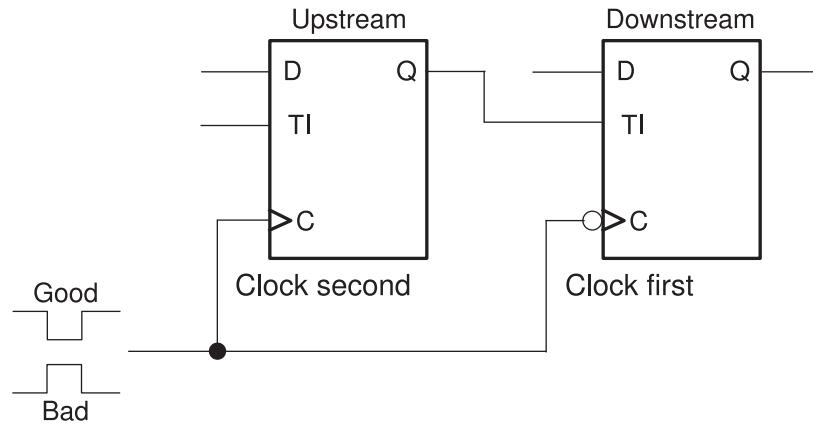
Use the `set_scan_configuration -internal_clocks [single|multi]` to avoid problems when placing gating logic on the clock lines. Alternatively, you can enable hold time violation fixing by using the `set_fix_hold` command before running the `insert_dft` command.

Although unlikely, skew can also cause setup violations.

Mixed Edges

If clocking on the scan chain is both inverted and noninverted, a problem similar to excessive skew can occur. For example, if the upstream flip-flop is rising-edge triggered and the downstream flip-flop is falling-edge triggered, a positive clock pulse first clocks a value into the upstream flip-flop. Then, on the same pulse's falling edge, it clocks the same value into the downstream flip-flop. [Figure 27](#) illustrates this condition.

Figure 27 Mixing Edges in a Scan Chain



If you must mix edges, there can be only one point in the chain where the clock phase reverses and clocks upstream flip-flops on the trailing edge of the clock. You must clock downstream flip-flops on the leading edge. For a positive pulse, the falling-edge-triggered flip-flop must be clocked first; for a negative pulse, the rising-edge-triggered flip-flop must be clocked first.

If you specify an invalid arrangement, the `preview_dft` command rejects the specification.

Multiple Clocks

If clocks originate from different sources, you must consider whether these clocks can be constrained to the same timing, given real-world tester constraints. The timing is determined from the waveforms defined with the `set_dft_signal` command. If optimistic assumptions are made in the defined waveforms, the circuit might not test properly. Consequently, using multiple clocks results in greater risk and lower predictability than using a single clock for all cells on a scan chain.

The `insert_dft` command reduces risks by

- Allowing only one clock per scan chain (default)
- Minimizing clock domain crossing
- Inserting lockup latches between clock domains (default)

If you change the defaults, you must pay attention to these risks.

Clock Requirements in LSSD Scan Styles

The same clocking requirements apply to all of the LSSD scan styles:

- Single-latch LSSD
- Double-latch LSSD
- Clocked LSSD

LSSD implements the shift function by using separate scan clocks on the master latch and the slave latch. Each master-slave latch pair is called a shift register latch. Test data transferred from a preceding shift register latch is stored in the master latch of the shift register latch when the “a scan clock” is active. The master test clock then changes to the inactive state, causing all latches to retain their stored values. The slave test clock then changes to the active state, permitting the slave latch to take on the state held in the master latch.

Scan mode does not require a scan mode control. Scan control is performed by using a clock different from the one used in functional mode.

Master Scan Clock and Slave Clock

To make an LSSD circuit scan-controllable, the scan chain must be complete from the scan input to the last flip-flop in the chain. The master scan clock and slave clock must be controllable from primary inputs.

Synchronized Clocks

You must synchronize the clocks for the various scan cells in the chain. Master latches must be enabled by the same clock pulse; slave latches must be enabled by a different, nonoverlapping clock pulse. If there is overlap, all latches are simultaneously transparent, causing incorrect operation.

If you specify an invalid arrangement, the `preview_dft` command rejects the specification.

Skew Control

Unlike edge-sensitive scan shift scan styles, LSSD scan styles do not require skew control because there is no race between clock and data. Correct low-frequency operation of the circuit for scan is ensured with LSSD.

Test Protocol Requirements

This topic discusses valid and invalid test protocols, methods of generating test protocols, and protocol types.

Valid and Invalid Test Protocols

DFT Compiler uses the test protocol for test design rule checking. TestMAX ATPG uses the test protocol generated by DFT Compiler for its own test design rule checking, as well as pattern generation and vector formatting steps. Using a single test protocol throughout these steps ensures consistency in processing the design for test.

Protocols are valid or invalid, depending on the following:

- Valid protocol

A protocol is valid when the design, test attributes, or test specifications have not changed since the last use of the `create_test_protocol` or `read_test_protocol` command.

- Invalid protocol

A protocol is invalid if there are any changes to a design's test attributes or test specifications since the last `create_test_protocol` or `read_test_protocol` command.

A protocol can become invalid if any of the following commands change the design, test attributes, or test specifications and you do not rerun the `create_test_protocol` or `read_test_protocol` command:

- `create_port`
 - `link`
 - `compile`
 - `compile_ultra`
 - `set_dft_signal -view spec -type`
 - `set_dft_signal -view existing_dft`
 - `set_dft_signal -view existing_dft -type constant -active_state`
-

Methods of Generating Test Protocols

DFT Compiler requires a test protocol before you can perform test design rule checking. This topic discusses the ways you can generate a test protocol.

Reading In an Existing Test Protocol

You might want to use an existing test protocol that was created for the current design or for another design that has a similar structure. You do this with the `read_test_protocol` command.

Creating a Fully User-Specified Test Protocol

A fully specified test protocol, in which all timing and test configuration constraints are defined, produces the most accurate results for your design. It also provides the shortest runtime. You do this with the `create_test_protocol` command.

Inferring a Test Protocol Based on Partial Specification

DFT Compiler can infer a test protocol for test design rule checking if you do not fully specify the test timing and configuration. Because DFT Compiler has to infer timing and test configuration, this protocol might not provide results that are as accurate as a fully specified protocol. Also, the runtime might be significantly slower than the fully specified protocol. You infer a test protocol with the `-infer_clock` and `-infer_asynch` options of the `create_test_protocol` command.

Inferring a Test Protocol

If you do not provide a test protocol, the protocol creation step can infer a protocol for your design. However, you are advised to fully specify your design and create a test protocol from that full specification. Certain environment variables and design characteristics determine the specific scan test instructions generated in the test protocol.

Initialization Protocol

An initialization protocol consists of the default pattern application sequence plus a user-defined initialization sequence. The initialization protocol is a series of initialization sequences read in from disk with the `read_test_protocol -section test_setup` command plus the `create_test_protocol` command, with or without the `-infer_clock` and `-infer_asynch` options.

Protocol Types

When test design rule checking infers a test protocol, the relationship between clock timing and strobe time determines the type of protocol it uses. If the tester strobe occurs in the middle of the cycle, before active clock edges, test design rule checking uses the strobe-before-clock protocol. If the tester strobe occurs at the end of the cycle, after the active edge of the clock, test design rule checking uses the strobe-after-clock protocol.

TestMAX ATPG uses the strobe-before-clock protocol.

Strobe-Before-Clock Protocol

If DFT Compiler infers a strobe-before-clock protocol, it uses the following steps to expand each scan pattern:

1. Data scan in
2. Parallel measure and capture cycle
3. Data scan out

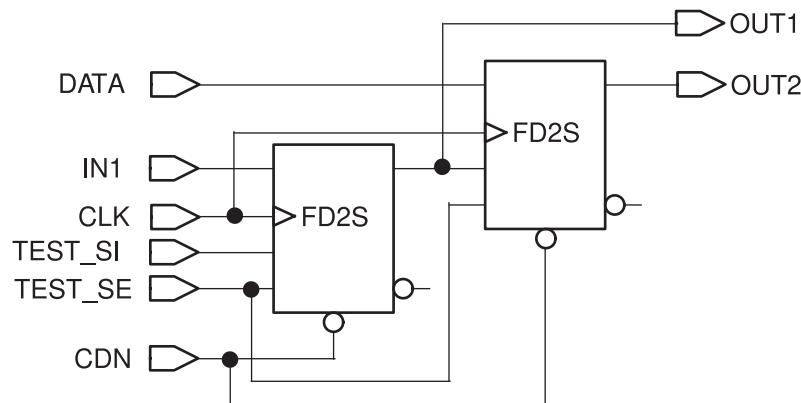
TestMAX ATPG uses the strobe-before-clock protocol.

Because the strobe occurs before the active edge of the clock, parallel measure and capture can take place in the same vector. Also, because the scan output is strobed before the scan data is shifted, an extra vector is not required to compare the first scan out. The strobe-before-clock protocol provides a small savings in vector count because you save two cycles for each operation.

A Strobe-Before-Clock Example

[Figure 28](#) provides a simple multiplexed flip-flop design example.

Figure 28 Strobe-Before-Clock Multiplexed Flip-Flop Design Example



The test protocol generated by DFT Compiler for the design in [Figure 28](#) contains the following instructions for scan testing the design:

1. Initialize the tester and configure the design.

a. Initialize the tester.

Place the system clock (CLK) in inactive state.

All nonclock input ports (IN1, IN2, TEST_SI, TEST_SE, and CDN) are “don’t care” values.

The output ports (OUT1 and OUT2) are masked.

b. Configure the design.

Disable the asynchronous pins by applying a logic 1 to the asynchronous control port (CDN).

All other ports are unchanged.

2. For each scan pattern, perform the following steps:

a. Scan in data.

Assert the scan-enable signal by applying a logic 1 to the scan-enable input port (TEST_SE).

Disable the asynchronous pins on the scan cells by applying a logic 1 to the asynchronous control port (CDN).

For each bit in the scan chain, apply data to the scan-input port (TEST_SI) and toggle the clock port (CLK).

All other input ports (IN1 and IN2) are don’t care values; the output ports (OUT1 and OUT2) are masked.

b. Perform parallel measure and capture.

Apply parallel data to all nonclock inputs.

Measure the response at the outputs, then pulse the system clock to capture the response in the scan cells.

Note:

In strobe-after-clock protocols, this step must be split into two steps, where measure is one step and pulse is the next.

c. Scan out data.

Assert the scan-enable signal by applying a logic 1 to the scan-enable input port (TEST_SE).

Disable the asynchronous pins on the scan cells by applying a logic 1 to the asynchronous control port (CDN).

All other input ports (IN1, IN2, and TEST_SI) are unchanged.

The nonscan-output ports (OUT1) are masked.

For each bit in the scan chain, measure the response at the scan-output port (OUT2), then toggle the clock port (CLK).

Strobe-After-Clock Protocol

If DFT Compiler infers a strobe-after-clock protocol, it uses the following steps to apply a scan pattern:

1. Data scan in
2. Parallel measure cycle
3. Parallel capture cycle
4. Measure first scan out
5. Data scan out

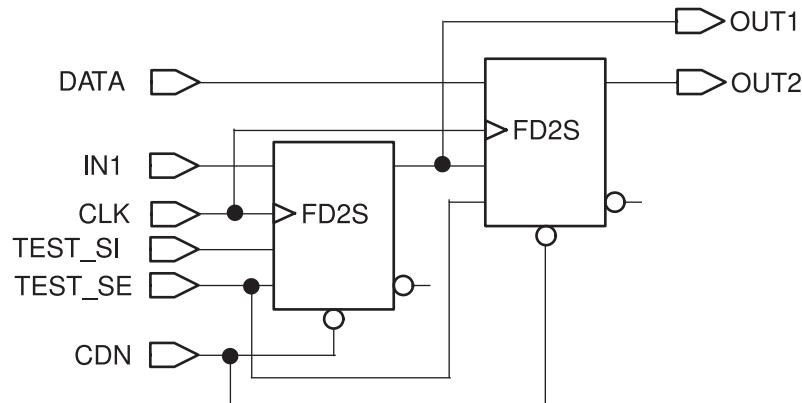
Note:

Older versions of DFT Compiler used default timing values that required a strobe-after-clock protocol. This is no longer the default behavior and using a strobe-after-clock protocol is strongly discouraged.

A Strobe-After-Clock Example

[Figure 29](#) provides a simple multiplexed flip-flop design example.

Figure 29 Strobe-After-Clock Multiplexed Flip-Flop Design Example



The test protocol generated by DFT Compiler for the design in [Figure 29](#) contains the following instructions for scan testing the design:

1. Initialize the tester and configure the design:

- a. Initialize the tester.

Place the system clock (CLK) in inactive state.

All nonclock input ports (IN1, IN2, TEST_SI, TEST_SE, and CDN) are “don’t care” values.

The output ports (OUT1 and OUT2) are masked.

- b. Configure the design.

Disable the asynchronous pins by applying a logic 1 to the asynchronous control port (CDN).

All other ports are unchanged.

2. For each scan pattern, perform the following steps:

- a. Scan in data.

Assert the scan-enable signal by applying a logic 1 to the scan-enable input port (TEST_SE).

Disable asynchronous pins on the scan cells by applying a logic 1 to the asynchronous control port (CDN).

For each bit in the scan chain, apply data to the scan-input port (TEST_SI) and toggle the clock port (CLK).

All other input ports (IN1 and IN2) are don't care values; the output ports (OUT1 and OUT2) are masked.

b. Perform parallel measure.

Apply parallel data to all nonclock inputs; the clock is held inactive.

Measure the response at the outputs.

c. Perform capture.

Pulse the system clock to capture the response in the scan cells.

All nonclock inputs remain unchanged; all outputs are masked.

d. Measure the first scan out.

Assert the scan-enable signal by applying a logic 1 to the scan-enable input port (TEST_SE).

Disable asynchronous pins on the scan cells by applying a logic 1 to the asynchronous control port (CDN).

All other input ports (IN1, IN2, and TEST_SI) are unchanged.

Nonscan-output ports (OUT1) are masked.

Measure the response at the scan-output port (OUT2).

e. Scan out data.

For each bit in the scan chain, toggle the clock port (CLK) and measure the response at the scan-output port (OUT2). The values applied to nonclock inputs are unchanged. All nonscan outputs (OUT1) are masked.

Part 2: DFT Compiler Scan

6

Getting Started

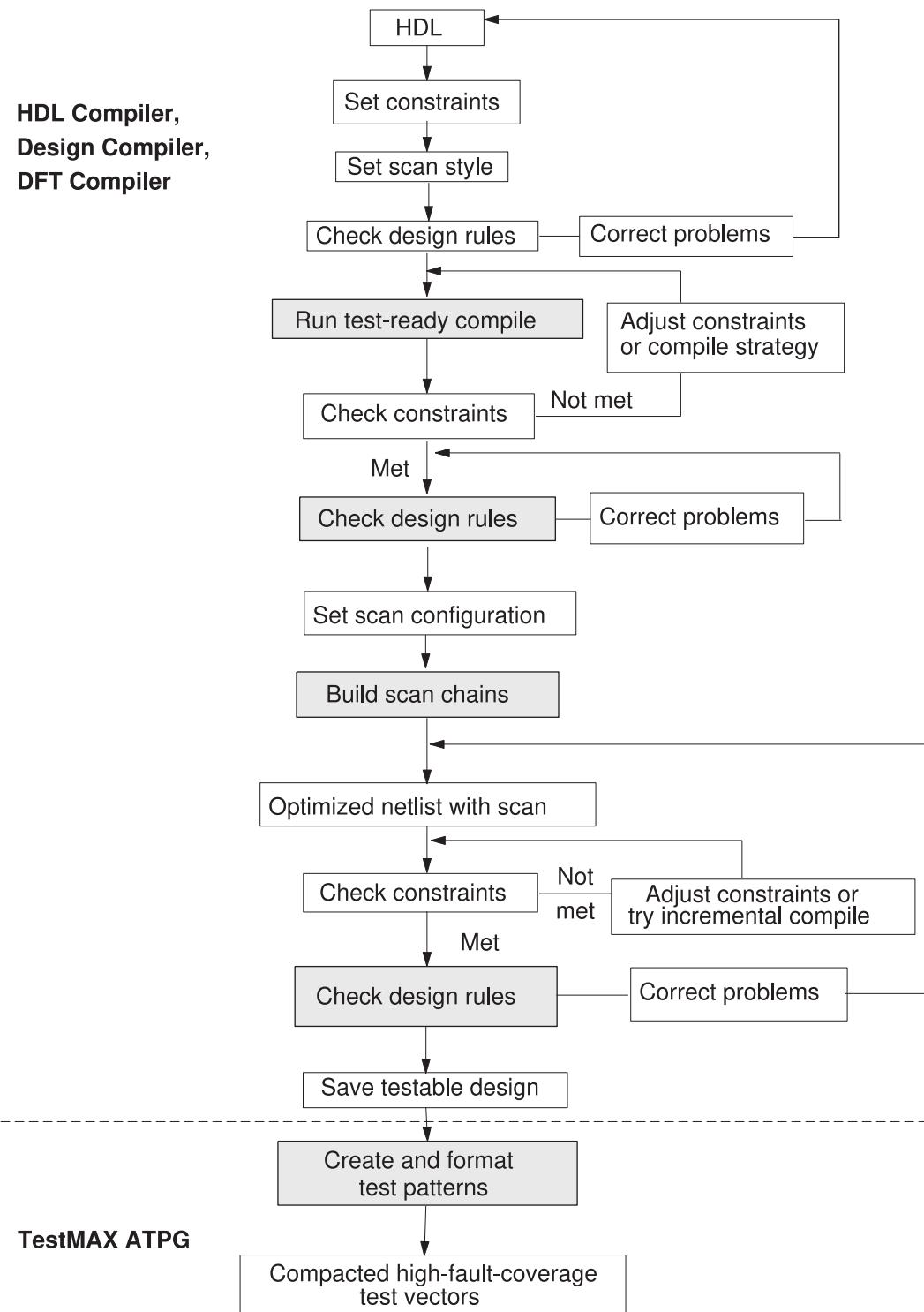
This chapter provides an overview of the basic flows, processes, and reports for DFT Compiler. References are provided throughout this chapter to more detailed information in subsequent chapters.

The basic processes, flows, and reports for using DFT Compiler are described in the following topics:

- [Preparing to Run DFT Compiler](#)
- [Performing Scan Synthesis](#)
- [Analyzing Your Post-DFT Design](#)
- [Reporting](#)
- [Designing Block by Block](#)
- [Performing Scan Extraction](#)
- [Hierarchical Scan Synthesis](#)
- [Physical DFT Features in Design Compiler](#)
- [DFT Flows in DC Explorer](#)

[Figure 30](#) shows a typical DFT insertion design flow that starts from RTL.

Figure 30 Typical Flat Design Flow for an Unmapped Design



Preparing to Run DFT Compiler

Before running DFT Compiler, you need to set up your interface, prepare your design environment, and read in your design.

This topic covers the following:

- [Invoking the Synthesis Tool](#)
- [Setting Up Your Design Environment](#)
- [Reading In Your Design](#)
- [Setting the Scan Style](#)
- [Configuring the Test Cycle Timing](#)
- [Defining the DFT Signals](#)

Invoking the Synthesis Tool

You can use DFT Compiler from within any of the following synthesis tools:

- Design Compiler
- Design Vision
- DC Explorer

The invocation commands are shown in [Table 19](#). DC Explorer runs in topographical mode by default, and it does not support DFT insertion or post-DFT commands.

User interface	Invocation command	Prompt
Design Compiler	dc_shell	dc_shell>
	dc_shell -topographical_mode	dc_shell-topo>
Design Compiler NXT	dcnxt_shell	dcnxt_shell>
Design Vision	design_vision	design_vision>
	design_vision -topographical_mode	design_vision-topo>
	dcnxt_shell -gui	dcnxt_shell>
DC Explorer	de_shell	de_shell>

Setting Up Your Design Environment

To set up your design environment, you need to define the paths for the logic libraries and designs you are using, and define any special reporting parameters. The following Synopsys system variables enable you to define the key parameters of your design environment:

Variable	Description
link_library	The ASIC vendor logic library where your design is initially represented.
target_library	Usually the same as your link_library, unless you are translating a design between technologies.
symbol_library	A file that contains definitions of the graphic symbols that represent cells in design schematics.
search_path	A list of alternative directory names to search to find the link_library, target_library, symbol_library, and design files.
hdlin_enable_dft_drc_info	Reports file names and line numbers associated with each violation during test design rule checking (DRC). This makes it easier for you to later edit the source code and fix violations.

For example,

```
# configure logic libraries
set_app_var target_library {my_library.db}
set_app_var link_library {* my_library.db}
set_app_var hdlin_enable_rtldrc_info true
```

Logic library configuration commands are normally included in the .synopsys_dc.setup file or user script, rather than entered manually at a tool prompt.

If you are using the Design Compiler tool in a physical synthesis flow, you must also configure your physical libraries. For example,

```
# configure physical libraries for topographical mode
if {! [file isdirectory my_mw_design_library]} {
    create_mw_lib -technology my_technology.tf \
        -mw_reference_library mw_reference_library \
        my_mw_design_library
} else {
    set_mw_lib_reference my_mw_design_library \
        -mw_reference_library my_mw_reference_library
}
```

```
open_mw_lib my_mw_design_library
set_tlu_plus_files \
    -max_tluplus my_TLUPLUS_MAX.tluplus \
    -min_tluplus my_TLUPLUS_MIN.tluplus \
    -tech2itf_map my_tech.map
```

Physical library configuration commands are normally included in your user script.

For more information about setting up your design environment, including logic and physical libraries, see “Setting Up the Libraries” chapter in the Design Compiler User Guide.

Reading In Your Design

To read in your design, specify the appropriate file read commands depending on the file format: `read_ddc`, `read_verilog`, `read_vhdl`. The following example reads in a list of Verilog files:

```
dc_shell> read_verilog {my_design.v my_block.v}
```

Use the `current_design` and `link` commands to link the top level of the current design:

```
dc_shell> current_design my_design
dc_shell> link
```

If DFT Compiler is unable to resolve any references, you must provide the missing designs before proceeding.

After linking, use the `read_sdc` command (or the `source` command) to apply the design constraints:

```
dc_shell> read_sdc top_constraints.sdc
```

Note:

If you read in the top-level design in the Synopsys logic database (.ddc) format, the design constraints might already be applied.

For more information about reading in your design, see “Reading Designs” section in the Design Compiler User Guide.

Setting the Scan Style

DFT Compiler uses the selected scan style to perform scan synthesis. A scan style dictates the appropriate scan cells to insert during optimization. This scan style is used on all modules of your design.

There are four types of scan styles available in DFT Compiler, shown in [Table 19](#).

Scan style	Keyword
Multiplexed flip-flop (default)	<code>multiplexed_flip_flop</code>
Clocked scan	<code>clocked_scan</code>
Level-sensitive scan design	<code>lssd</code>
Scan-enabled level-sensitive scan design	<code>scan_enabled_lssd</code>

The default style is multiplexed flip-flop. To specify another scan style, use the `-style` option of the `set_scan_configuration` command. For example,

```
dc_shell> set_scan_configuration -style clocked_scan
```

See Also

- [Chapter 4, Scan Styles](#) for more information about the supported scan styles
- [Specifying a Scan Style on page 167](#) for more information about the process for selecting and specifying a scan style for your design

Configuring the Test Cycle Timing

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

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

These are the default settings; you do not need to add them to your script.

Defining the DFT Signals

Most DFT Compiler commands include the concept of a *view*, specified with the `-view` option. The valid view values are:

- `-view existing_dft`

The existing DFT view is *descriptive* and describes an existing signal network. An example is an existing functional clock signal that is also used as a scan clock in test mode.

- `-view spec`

The specification view is *prescriptive* and describes action that must be taken during DFT insertion. It indicates that the signal network or connection does not yet exist, and the `insert_dft` command must create it. An example is a scan-enable signal network that must be routed to all scannable flip-flops during DFT insertion.

A view is typically specified in scan specification commands, such as `set_dft_signal`. When performing scan synthesis, you use a combination of the two views. When you define existing signals that are used in test mode, you use the existing DFT view. When you define the DFT structure you want inserted, you use the specification view.

Define any clocks and asynchronous set and reset signals in the existing DFT view:

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock ...
dc_shell> set_dft_signal -view existing_dft -type Reset ...
```

If you have a dedicated scan-enable port, define it in the specification view:

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

If no scan-enable port is identified, DFT Compiler creates a new scan-enable port.

If you are using existing ports as scan-in and scan-out ports, define them in the specification view (even if they have existing functional logic connections):

```
dc_shell> set_dft_signal -view spec -type ScanDataIn -port DAT_IN[7]
dc_shell> ...
dc_shell> set_dft_signal -view spec -type ScanDataIn -port DAT_IN[0]
dc_shell> set_dft_signal -view spec -type ScanDataOut -port DAT_OUT[7]
dc_shell> ...
dc_shell> set_dft_signal -view spec -type ScanDataOut -port DAT_OUT[0]
```

Otherwise, DFT Compiler creates new scan-in and scan-out ports as needed.

After defining your DFT signals, create a test protocol:

```
dc_shell> create_test_protocol
```

See Also

- [Chapter 9, Architecting Your Test Design](#) for more information about defining DFT signals

Performing Scan Synthesis

The scan synthesis process reads your RTL design, synthesizes it, tests it again, performs scan insertion, and analyzes your post-DFT design.

This topic covers the following processes:

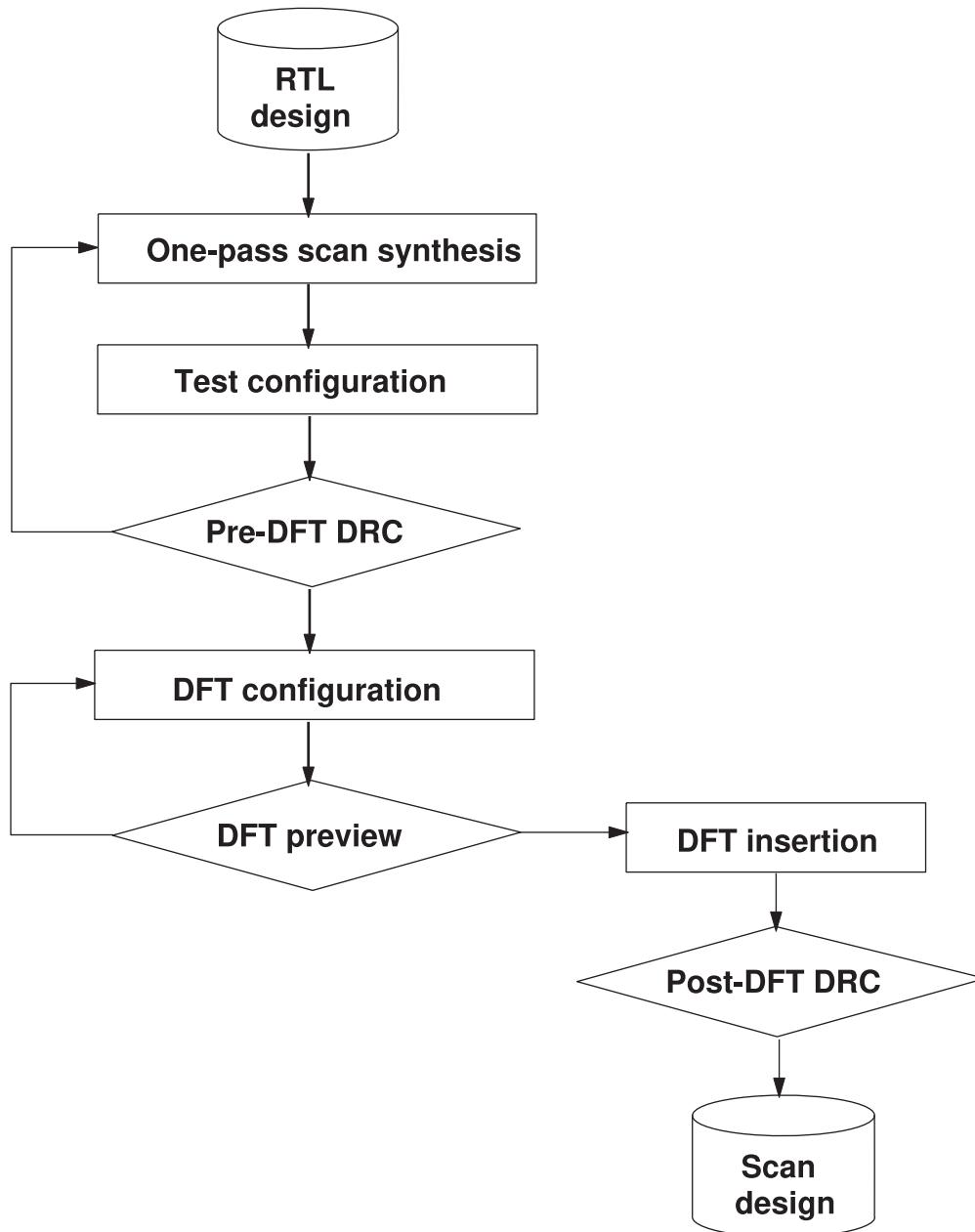
- [Performing One-Pass Scan Synthesis](#)
- [Performing Scan Insertion](#)
- [Performing Post-DFT Optimization](#)

Note:

The first step pertains only to RTL designs. If your design is already mapped to logic gates, start with the [Performing Pre-DFT Test DRC](#) step.

[Figure 31](#) shows a basic scan synthesis flow.

Figure 31 Basic Scan Synthesis Flow



Performing One-Pass Scan Synthesis

After reading in your RTL files, you are ready to perform one-pass scan synthesis, which performs test-ready compilation. To do this, specify the `compile -scan` command, as shown in the following example:

```
dc_shell> compile -scan
```

When using the DC Ultra tool (such as with topographical mode), use the following command:

```
dc_shell> compile_ultra -scan
```

The `-scan` option performs a *test-ready compile*, which maps directly to scan cells. This helps eliminate logically untestable circuitry and is an important part of the Synopsys test methodology. The resulting netlist with unstitched scan cells is called an *unrouted scan design*.

See Also

- [Chapter 8, Performing Scan Replacement](#) for more information about one-pass scan synthesis
-

Performing Scan Insertion

The scan insertion process consists of four primary phases:

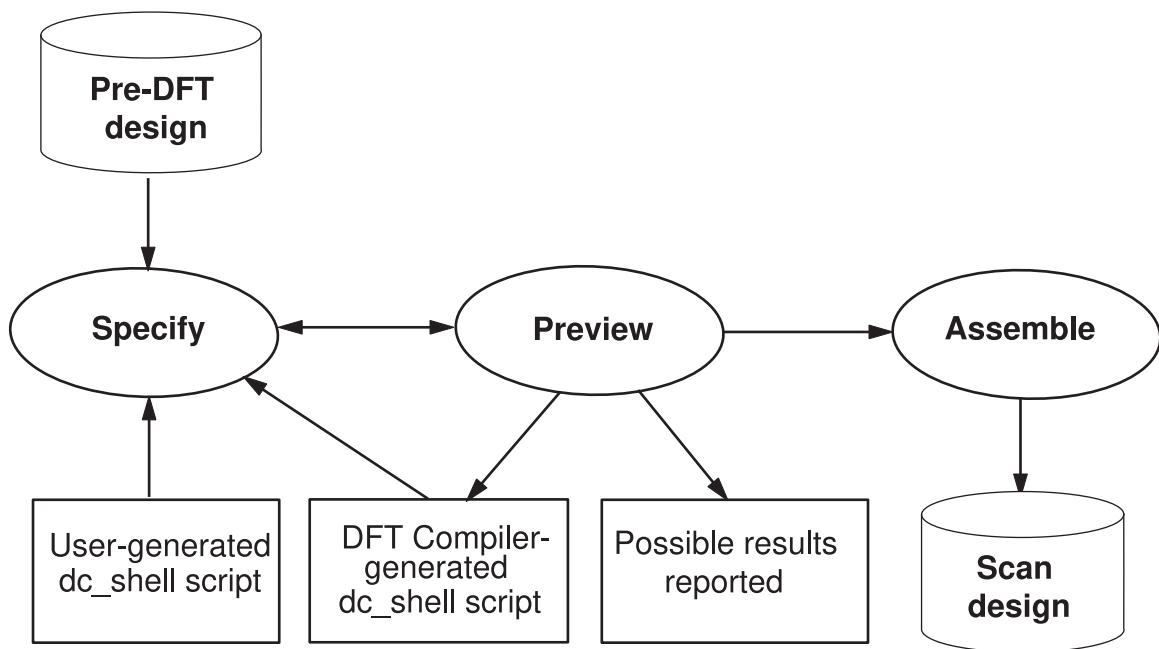
- [Configuring Scan Insertion](#)
- [Previewing Scan Insertion](#)
- [Performing Pre-DFT Test DRC](#)
- [Inserting the DFT Logic](#)

This topic briefly describes each of these phases.

For a complete description of the scan insertion process, see [Chapter 9, Architecting Your Test Design.](#)

[Figure 32](#) illustrates the scan insertion flow.

Figure 32 Scan Insertion Flow



Configuring Scan Insertion

To configure scan insertion, you can specify test ports, define test modes, and identify and mark any cells that you do not want to have scanned. You can set many of these configuration parameters by using commands such as `set_scan_configuration`, `set_dft_signal`, or `set_scan_element`.

The following example shows some typical DFT configuration commands:

```
dc_shell> set_scan_configuration -chain_count 4
dc_shell> set_dft_signal -view spec \
           -type ScanDataIn -port TEST_SI
dc_shell> set_dft_signal -view spec \
           -type ScanDataOut -port TEST_SO
dc_shell> set_dft_signal -view spec \
           -type ScanEnable -port TEST_SE
```

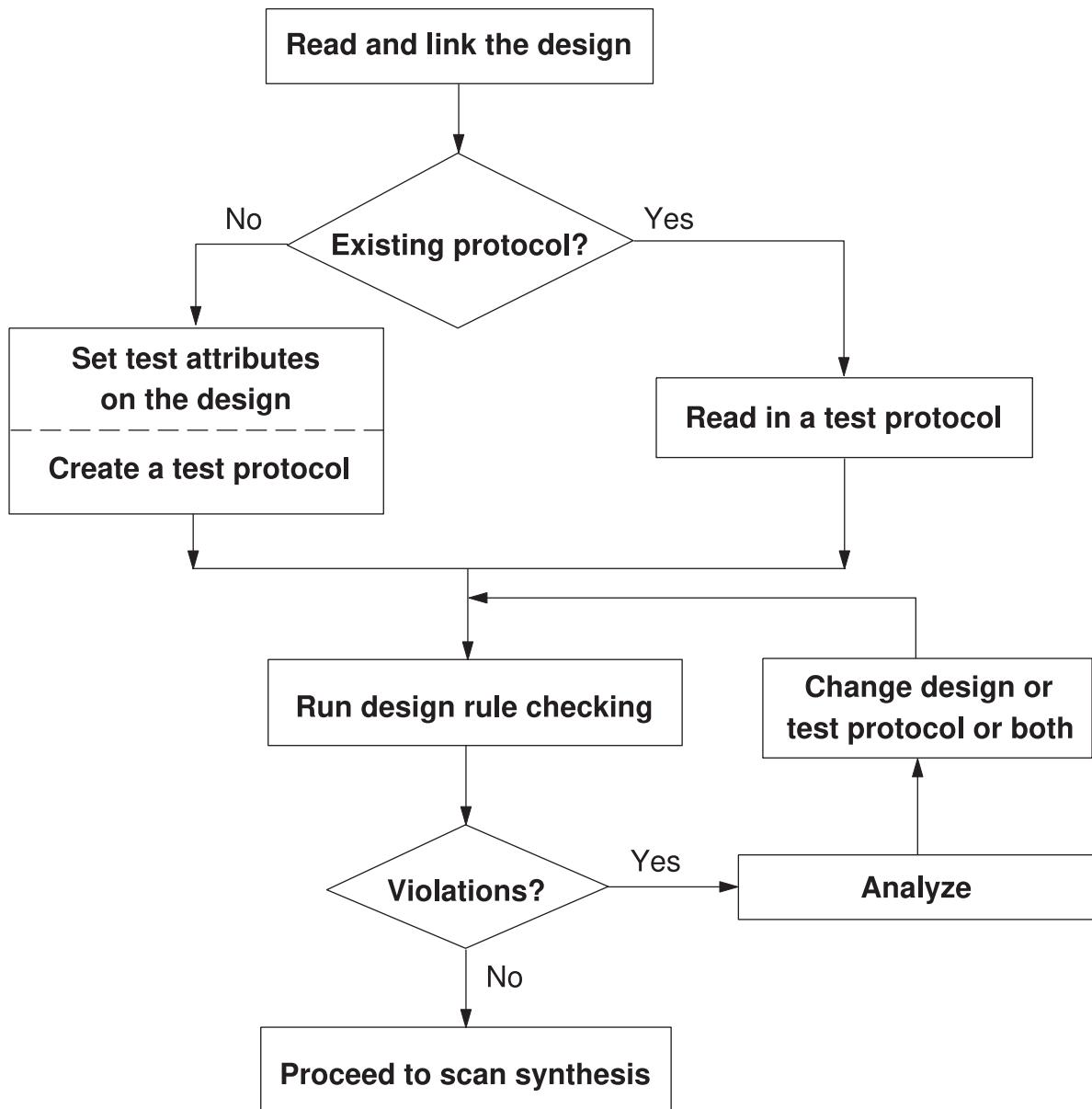
See Also

- [Chapter 9, Architecting Your Test Design](#) for more information about configuring scan insertion

Performing Pre-DFT Test DRC

Pre-DFT test design rule checking (DRC) process analyzes your unrouted scan design, based on a set of constraints applicable to your selected scan style, then outputs a set of violations. Based on the violations, you make changes to your design to prepare it for DFT insertion. [Figure 33](#) illustrates the pre-DFT test DRC flow.

Figure 33 Pre-DFT Test DRC Flow



To perform pre-DFT test design rule checking,

1. Create the test protocol.

You can use the test protocol definition you previously used for RTL test DRC, but you must still regenerate it.

```
dc_shell> create_test_protocol
```

2. Run pre-DFT test DRC.

```
dc_shell> dft_drc
```

3. Check for violations, then do one of the following:

- If violations are reported, make changes to the design or test protocol and repeat steps 1 and 2.
- If no violations are reported, proceed to scan insertion.

See Also

- [Chapter 13, Pre-DFT Test Design Rule Checking](#) for more information about checking for test design rule violations before DFT insertion

Previewing Scan Insertion

Before performing scan insertion, you can preview your scan design by running the `preview_dft` command. This command generates a scan chain design that satisfies scan specifications on your current design and displays the scan chain design. This allows you to preview your scan chain design without synthesizing it and change your specifications as necessary. The following example shows how to specify the `preview_dft` command:

```
dc_shell> preview_dft
```

[Example 1](#) shows an example of the report generated by the `preview_dft` command.

Example 1 Report 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
```

See Also

- [Previewing the DFT Logic on page 603](#) for more information about previewing scan insertion

Inserting the DFT Logic

After configuring and previewing your design, assemble the scan chains by using the `insert_dft` command:

```
dc_shell> insert_dft
```

See Also

- [Inserting the DFT Logic on page 610](#) for more information about how DFT insertion performs scan replacement, scan stitching, and signal routing

Performing Post-DFT Optimization

Post-DFT optimization is gate-level optimization performed after inserting and mapping the new DFT structures. It performs optimizations such as selecting scan-out signal connections (Q or QN) to minimize constraint violations. This reduces the scan-related overhead on timing performance and area, and it eliminates synthesis design rule errors.

The `insert_dft` command creates scan chains that are functional under zero-delay assumptions using the scan clock waveforms described by the `set_dft_signal` command. However, post-DFT optimization uses the clock waveforms described by the `create_clock` command; it does not use the scan clock timing values described by the `set_dft_signal` command.

To include scan clock timing in post-DFT optimization, you can use multicorner-multimode optimization in Design Compiler Graphical to define a mode where the scan clocks are defined with the `create_clock` command. For more information, see “Optimizing Multicorner-Multimode Designs” in the *Design Compiler User Guide*.

The post-DFT optimization flow depends on the Design Compiler mode.

Post-DFT Optimization in Design Compiler Wire Load Mode

In Design Compiler wire load mode, the `insert_dft` command automatically performs basic gate-level post-DFT optimization by default. If needed, you can disable it with the following command:

```
dc_shell> set_dft_insertion_configuration \
           -synthesis_optimization none
```

In this case, you can still manually perform a post-DFT incremental compile if you disable automatic post-DFT synthesis optimization:

```
dc_shell> # for a DC Expert compile
dc_shell> insert_dft
dc_shell> compile -scan -incremental

dc_shell> # for a DC Ultra compile
```

```
dc_shell> insert_dft
dc_shell> compile_ultra -scan -incremental
```

Post-DFT Optimization in Design Compiler Topographical Mode

In Design Compiler topographical mode, the `insert_dft` command maps new logic, but does not perform post-DFT optimization. In this mode, you perform post-DFT optimization by manually running an incremental post-DFT topographical compile after the `insert_dft` command completes. For example,

```
dc_shell-topo> insert_dft
dc_shell-topo> compile_ultra -scan -incremental ;# topographical
```

Design Compiler synthesis optimizes the newly inserted DFT logic, and it optimizes the design to accommodate the additional area and timing overhead of the DFT logic.

The tool issues a warning message if you attempt to enable post-DFT optimization in topographical mode:

```
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.

This incremental compile behavior also applies to Design Compiler Graphical, which runs from within Design Compiler in topographical mode. However, the incremental compile command also requires the `-spg` option:

```
dc_shell-topo> insert_dft
dc_shell-topo> compile_ultra -scan -spg -incremental
```

Analyzing Your Post-DFT Design

After you perform DFT insertion, you should perform design rule checking again to ensure that no new violations have been introduced into your design:

```
dc_shell> dft_drc
```

This is called *post-DFT DRC*. DFT Compiler checks for and describes potential problems with the testability of your design. These checks are more comprehensive than those in pre-DFT DRC, and they check for the correct operation of the scan chain. After you correct all the reported violations, you can proceed with the next step. Failure to correct the reported violations typically results in lower fault coverage.

Note:

Some features and flows do not support post-DFT DRC, as noted in [Post-DFT DRC Limitations on page 627](#). In such cases, use DRC in the TestMAX ATPG tool to validate the DFT-inserted design.

To analyze your post-DFT design:

1. Save your design and test protocol.

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

dc_shell> write_test_protocol \
           -output my_design_final.spf
```

2. Run post-DFT test DRC.

```
dc_shell> dft_drc
```

Note:

Some features and flows do not support post-DFT DRC in DFT Compiler; you must perform DRC in the TestMAX ATPG tool.

3. Report the scan structures.

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

dc_shell> report_scan_path -view existing_dft \
           -cell all
```

See Also

- [Post-DFT Insertion Test Design Rule Checking on page 612](#) for more information about analyzing your post-DFT design

Reporting

All DFT specification commands have corresponding reporting commands. To report what exists in the design, use the `-view existing_dft` option of the reporting command. To report what you have specified for insertion, use the `-view spec` option, which is also the default.

Example 2 DFT Configuration Report

```
dc_shell> report_dft_configuration
*****
*****
```

```

Report : DFT configuration
Design : test
Version: 2003.12
Date : Fri Aug 22 16:10:05 2003
*****
DFT Structures          Status
-----
Scan:                  Enable
Autofix:                Enable
Test point:             Disable
Wrapper:                Disable
Integration:            Disable
Boundary scan:          Disable

```

Example 3 Scan Configuration Report

```

dc_shell> report_scan_configuration
*****
Report : Scan configuration
Design : SYNCH
Version: 2003.12
Date : Fri Aug 22 15:48:24 2003
*****

=====
TEST MODE: Internal_scan
VIEW : Specification
=====

Chain count:           Undefined
Scan Style:            Multiplexed flip-flop
Maximum scan chain length: Undefined
Preserve multibit segments: True
Clock mixing:           Not defined
Internal clocks:        False
Add lockup:              True
Insert terminal lockup: False
Create dedicated scan out ports: False
Shared scan in:          0
Bidirectional mode:      No bidirectional type

```

Example 4 DFT Signal Report

```

dc_shell> report_dft_signal -view existing_dft
*****
Report : DFT signals
Design : SYNCH
Version: 2003.12
Date : Fri Aug 22 15:48:51 2003

```

```
*****
=====
TEST MODE: Internal_scan
VIEW : Existing DFT
=====
Port   SignalType      Act Hkup Timing
----  -----  ---  ---  ---
hrst_L Reset          0   -    P 100.0 R 55.0 F 45.0
mrxc ScanMasterClock 1   -    P 100.0 R 45.0 F 55.0
mrxc MasterClock      1   -    P 100.0 R 45.0 F 55.0
clk3 ScanMasterClock 1   -    P 100.0 R 45.0 F 55.0
clk3 MasterClock      1   -    P 100.0 R 45.0 F 55.0
clk2 ScanMasterClock 1   -    P 100.0 R 45.0 F 55.0
clk2 MasterClock      1   -    P 100.0 R 45.0 F 55.0

dc_shell> report_dft_signal -view spec
*****
Report : DFT signals
Design : SYNCH
Version: 2003.12
Date : Fri Aug 22 16:25:11 2003
*****


=====
TEST MODE: Internal_scan
VIEW : Specification
=====
Port SignalType Active Hookup Timing
---- -----  ---  -----
SI1 ScanDataIn  -     -    Delay 5.0
```

Example 5 Report on a User-Specified Scan Path

```
dc_shell> report_scan_path -view spec -chain all
*****
Report : Scan path
Design : SYNCH
Version: 2003.12
Date : Fri Aug 22 15:50:07 2003
*****


=====
TEST MODE: Internal_scan
VIEW : Specification
=====
Scan_path ScanDataIn (h) ScanDataOut(h) ScanEnable (h)
-----  -----  -----  -----
chain1   -       -       -
```

Example 6 AutoFix Configuration Report

```
dc_shell> report_autofix_configuration
*****
Report : Autofix configuration
Design : example
Version: V-2004.06-SP1
Date : Fri Jul 2 12:11:19 2004
*****
=====
TEST MODE: all_dft
VIEW : Specification
=====

Fix type: Set
Fix method: Mux
Fix latches: Disable
Fix type: Clock
Fix latches: Disable
Fix clocks used as data: Disable
Fix type: Internal_bus
Fix method: Enable_one
Fix type: External_bus
Fix method: Disable_all
```

Designing Block by Block

If you develop your designs on a block-by-block basis, you can use pre-DFT DRC (or RTL DRC) to check the test design rules of any particular block without also performing DFT insertion in that block. This allows you to assess the testability as you develop each block of your design, which helps you identify and fix testability problems at an early stage.

To do this, define the existing DFT signals in the block, such as scan clocks, then run pre-DFT 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.

See Also

- [Chapter 13, Pre-DFT Test Design Rule Checking](#) for more information about checking for test design rule violations before DFT insertion

Performing Scan Extraction

Scan extraction is the process of reading in an ASCII netlist that lacks test attributes, then analyzing the netlist to determine the scan chain structures. After identifying the scan structures, you can write out test model and test protocol files for design reuse.

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, define your test structures in the existing DFT view (by using the `-view existing_dft` option) because they already exist in your design.

Scan extraction only supports standard scan designs. You cannot extract scan structures for compressed scan designs.

To perform scan extraction:

1. Perform the steps described in [Preparing to Run DFT Compiler on page 105](#): read in and link the design, configure the scan style, and define the basic DFT signals.
2. Specify that the design contains existing scan structures:

```
dc_shell> set_scan_state scan_existing
```

3. Define the scan input and scan output for each existing scan chain using the `set_scan_path` and `set_dft_signal` commands, as shown in the following example:

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanDataIn -port {TEST_SI1 TEST_SI2}
```

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanDataOut -port {TEST_SO1 TEST_SO2}
```

```
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_SI1 \
    -scan_data_out TEST_SO1
```

```
dc_shell> set_scan_path chain2 \
    -view existing_dft \
    -scan_data_in TEST_SI2 \
    -scan_data_out TEST_SO2
```

4. Create the test protocol by using the `create_test_protocol` command:

```
dc_shell> create_test_protocol
```

5. Extract the scan chains by using the `dft_drc` command:

```
dc_shell> dft_drc
```

6. Report the extracted scan chains by using the `report_scan_path` commands, and ensure that the reports are as expected:

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

```
dc_shell> report_scan_path -view existing_dft \
           -cell all
```

Hierarchical Scan Synthesis

In hierarchical scan synthesis, you perform DFT insertion at a lower level of hierarchy, then incorporate those completed scan structures into DFT insertion at a higher level of hierarchy. Test models represent DFT-inserted blocks during DFT operations, which improves tool performance and capacity for multi-million-gate designs.

Hierarchical scan synthesis is described in the following topics:

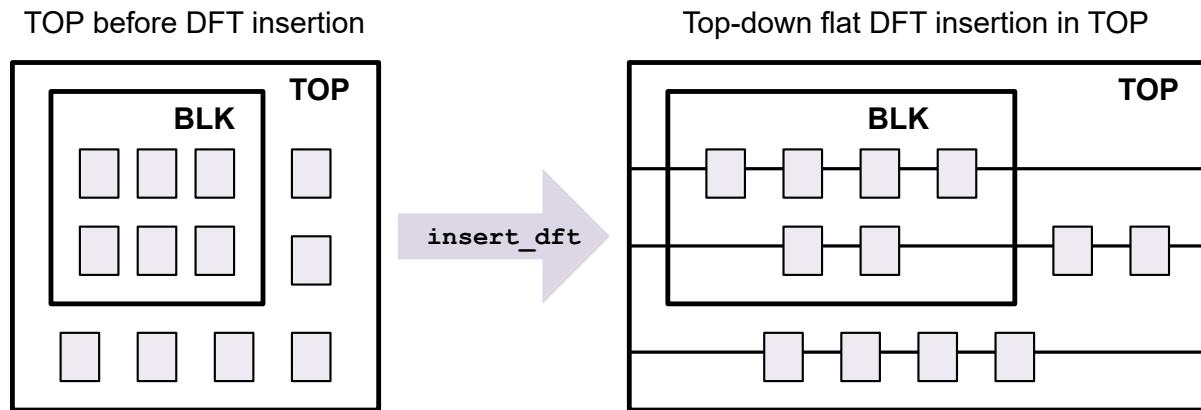
- [Top-Down Flat Versus Bottom-Up Hierarchical](#)
- [Introduction to Test Models](#)
- [Writing Out a CTL Model at the Core Level](#)
- [Reading In and Using CTL Models at the Top Level](#)
- [Checking Connectivity to Cores at the Top Level](#)
- [Using Advanced Clock Feedthrough Analysis](#)
- [Connecting the Scan-Enable Pins of Cores](#)
- [Hierarchical Synthesis, DFT Insertion, and Layout Flows](#)
- [Linking Test Models to Library Cells](#)
- [Checking Library Cells for CTL Model Information](#)

Top-Down Flat Versus Bottom-Up Hierarchical

In *top-down flat* scan synthesis, you perform a single DFT insertion operation at the top level of your design. See [Figure 34](#). This flow is simple, but it requires that DFT

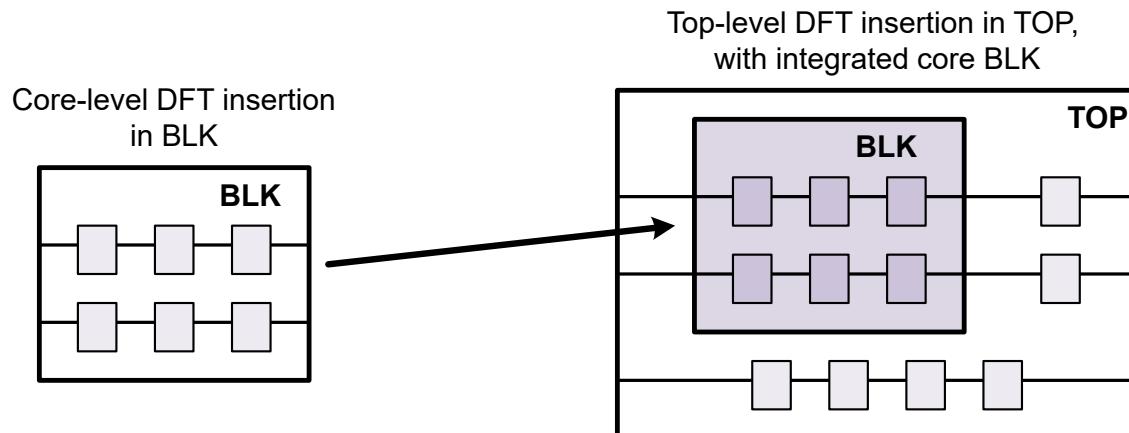
insertion be repeated for the entire design if any part of the design changes, which is time-consuming for large designs.

Figure 34 Top-Down Flat Scan Synthesis



In *bottom-up hierarchical* scan synthesis, you perform DFT insertion at a lower level of hierarchy, then incorporate those completed scan structures into DFT insertion at a higher level of hierarchy. See [Figure 35](#). This is also simply called a *hierarchical scan synthesis* (HSS) flow.

Figure 35 Bottom-Up Hierarchical Scan Synthesis



Hierarchical scan synthesis uses test models to represent core designs during top-level DFT operations, which improves tool performance and capacity for multimillion-gate designs. This is useful when

- You have a very large design that is not suitable for a single top-down DFT insertion
- You want to be able to rearchitect DFT structures in a block independently of its surrounding design logic
- You want to create a DFT-inserted block that can be reused in future designs

A lower-level block that is DFT-inserted is called a *core*. When a core is incorporated into scan structures at a higher level of DFT insertion, the core is said to be *integrated* at that level. The level of hierarchy where a core is integrated is sometimes referred to as the *top level* (as opposed to the core level), although this DFT-inserted top level can itself become a core in an even higher level of hierarchy.

Note:

In the DFT documentation, the term “block” refers to any hierarchical design, while the term “core” refers specifically to DFT-inserted blocks with CTL model information.

After a core is created, its scan structures are fixed. However, core-level scan chains become scan segments that are incorporated into top-level scan chain balancing.

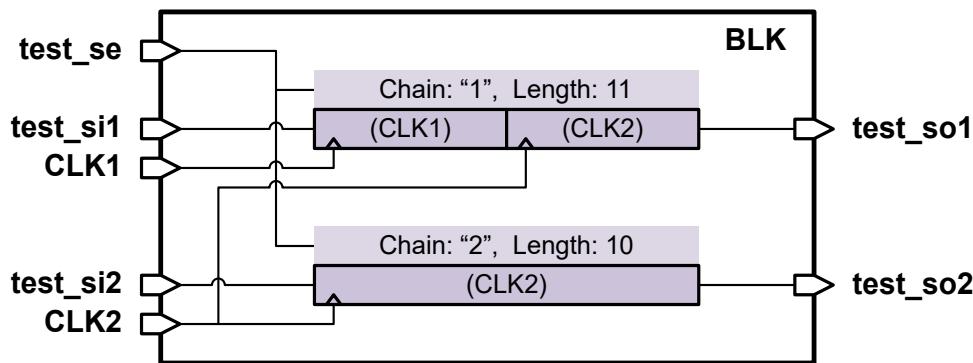
Introduction to Test Models

In a hierarchical scan synthesis flow, the tool creates a test model for a DFT-inserted core that describes only the information needed to integrate the core at a higher level of design hierarchy. Test models store information about

- Port names
- Port directions (input, output, bidirectional)
- Scan structures (such as scan chains, pipeline registers, and terminal lockup latches)
- Scan clock signals
- Asynchronous sets and reset signals
- Three-state disable signals
- Other test-related signals
- Multiple test modes
- Test protocol information, such as initialization, sequencing, and clock waveforms

[Figure 36](#) shows a test model representation of a design example.

Figure 36 Test Model Representation of a Design Example



Test models are described using Core Test Language (CTL); thus, test models are typically called *CTL models*. CTL models can be written and read in three file formats:

- .ddc format

This is a binary file that is typically written out in a synthesis flow. When you write out a design in .ddc format after performing DFT insertion, it automatically includes the binary CTL model description of that design.

- .ctl format

This is an ASCII file that contains the ASCII Core Test Language description of a core.

- .ctlddc format

This is a binary file that can be optionally written out in a synthesis flow. This format includes only the binary CTL model description of the design. It can be read in and linked like a .ddc file.

A CTL model does not include SCANDEF information. However, a .ddc file can contain both CTL model information and SCANDEF information.

CTL models are not adequate for the generation of test patterns; you must use an actual netlist representation for automatic test pattern generation (ATPG).

Writing Out a CTL Model at the Core Level

To create a CTL model for a core, you perform DFT insertion for the core design, then you write out its CTL model information.

Cores can be written in the file formats shown in [Table 20](#). Different formats contain different information.

Table 20 Design Compiler Commands for Writing Out CTL-Modeled Designs

Format	Command	Output data description
.ddc ¹	write -format ddc -hierarchy -output core.ddc	Complete binary database for the current design; includes netlist, constraints, attributes, and CTL model information
.ctl	write_test_model -format ctl -output core.ctl	ASCII CTL model representation of the current design; does not include netlist, constraints, or attributes information
.ctlddc	write_test_model -format ddc -output core.ctlddc	Binary CTL model representation of the current design; does not include netlist, constraints, or attributes information

It might be useful to write out the design in multiple formats. For example, you can write out a .ddc file containing the CTL model information for use by the tool at the integration level, along with an ASCII CTL model for reference.

[Example 7](#) shows a simple core-level script.

[Example 7 Core-Level Script Example](#)

```
# configure libraries
set link_library {* my_lib.db}
set target_library {my_lib.db}

# read and link the design
read_verilog RTL/core.v
current_design core
link

# perform test-ready compile
compile -scan

# configure DFT
set_dft_signal -view existing_dft -type ScanClock \
    -port clk -timing [list 45 55]

# preview and insert DFT
create_test_protocol
dft_drc
set_scan_configuration -chain_count 2
preview_dft
insert_dft

# write out core-level design information
write -format ddc -hierarchy -output DDC/core.ddc      ;# full design
```

1. *Includes block abstractions*

```
write_test_model -format ctl -output CTL/core.ctl      ;# ASCII CTL model
write -format verilog -hierarchy -output GATES/core.vg ;# ATPG netlist
write_test_protocol -output SPF/core.spf             ;# ATPG protocol
```

Reading In and Using CTL Models at the Top Level

At the top level of your design, you read in CTL-modeled core designs along with the other design logic, then you perform scan insertion. The tool automatically incorporates the core-level scan structures into the top-level scan structures.

You can read CTL-modeled core designs into a top-level design in multiple formats. Different formats provide different types of information, as shown in [Table 21](#). For multiple cores, you can read in a mix of formats.

Table 21 Commands for Reading In CTL-Modeled Designs

Format	Commands	Input data description
.ddc ²	read_ddc core.ddc	Complete binary database for the core; includes netlist, constraints, attributes, and CTL model information
.ctl	read_test_model -format ctl -design core core.ctl	ASCII CTL model of the core; does not include netlist, constraints, or attributes information
.ctl (with netlist)	read_verilog core.v read_test_model -format ctl -design core core.ctl	ASCII CTL model of the core with netlist; does not include constraints or attributes information
.ctlddc ^a	read_test_model -format ddc core.ctlddc	Binary CTL model of the core; does not include netlist, constraints, or attributes information
.ctlddc (with netlist)	read_verilog core.v read_test_model -format ddc core.ctlddc	Binary CTL model of the core with netlist; does not include constraints or attributes information

The `read_test_model` command attaches CTL model information to a design. If the design already has CTL model information, the newly read CTL model replaces the existing model. If the design does not exist, the command creates a black-box design using the port information in the CTL model information.

When you link the top-level design, the link process looks for .ddc files in the search path to resolve any unresolved references. It does not consider .ctl or .ctlddc files, although it does consider .ctlddc files that have a .ddc extension.

2. For this case, you can include the .ddc or .ctlddc files in the `link_library` list instead of explicitly reading it in, or you can rely on the link process to read them in (for files that have a .ddc extension).

You can use the `list_test_models` command, before or after linking, to see what designs have CTL test models attached to them. For example,

```
dc_shell> list_test_models
core1          /home/hier_dft/top.db
core2          /home/hier_dft/top.db
```

For CTL-modeled cores, DFT and synthesis commands use different types of information:

- DFT commands

When DFT commands (such as the `preview_dft` and `insert_dft` commands) are run, they use the CTL model information attached to CTL-modeled cores in place of any netlist information. They also use test attributes, if present.

If the `use_test_model -false` command is applied to a core, the CTL model information is ignored and the netlist information is used instead.

- Synthesis reporting and optimization commands

When synthesis reporting and optimization commands (such as `report_timing` and `compile`) are run, they use only the netlist, constraints, and attributes information for CTL-modeled cores. Synthesis commands do not use CTL model information at all.

Caution:

If a CTL-modeled core contains no netlist information, it is treated as a black box. Use this representation only in a flow that performs DFT insertion without synthesis optimization.

When using Design Compiler topographical mode, you can use test models from DFT Compiler together with block abstractions from Design Compiler topographical mode or IC Compiler. For more information, see “Compile Flows in Topographical Mode” in the *Design Compiler User Guide*.

Note that the post-insertion incremental optimization performed by the `insert_dft` command in Design Compiler wire load mode is a synthesis operation, not a DFT operation, and it uses netlist information if available.

Example 8 shows a simple top-level script that integrates two cores: one saved in .ddc format, and another saved in Verilog format with an ASCII CTL model.

Example 8 Top-Level Core Integration Script Example

```
# configure libraries
set link_library {* my_lib.db}
set target_library {my_lib.db}

# read two cores and top-level netlist, then link the design
read_ddc DDC/core.ddc ;# contains netlist and CTL model information
```

```

read_verilog GATES/IPBLK.vg      ;# Verilog netlist for IP block
read_test_model -format ctl \
    -design IPBLK CTL/IPBLK.ctl ;# attach CTL model to Verilog design

read_verilog RTL/top.v

current_design top
link

# perform test-ready compile
compile -scan

# configure DFT
set_dft_signal -view existing_dft -type ScanClock \
    -port clk -timing [list 45 55]

# preview and insert DFT
create_test_protocol
dft_drc
set_scan_configuration -chain_count 2
preview_dft
insert_dft

# write out top-level design information
write -format ddc -hierarchy -output DDC/top.ddc      ;# full design
write -format verilog -hierarchy -output GATES/top.vg ;# ATPG netlist
write_test_protocol -output SPF/top.spf               ;# ATPG protocol

```

Checking Connectivity to Cores at the Top Level

In a core integration flow, you must ensure that the TestMode and Constant signals entering a block match what is required to shift the scan chains through the block. If conditions do not match, scan blockages can result.

You can use design rule checking (DRC) to confirm that these conditions are met before DFT insertion. Before running the `dft_drc` command, set the following variable:

```
dc_shell> set_app_var test_validate_test_model_connectivity true
```

The `dft_drc` command simulates the `test_setup` procedure and reports any mismatches between actual and expected values on the TestMode and Constant signals of instances represented by the test models.

If a mismatch is detected, the scan segments represented in the test model are not stitched onto the scan chains.

Using Advanced Clock Feedthrough Analysis

A *clock feedthrough* is a logic path in the design where the output pin of the path has the same clock behavior as the input pin of the path. The tool uses feedthrough path information during DRC. When the tool writes out CTL test model information for the design, the model includes any feedthrough paths that span from input to output.

By default, DFT Compiler performs basic analysis of clock network logic to determine clock propagation behavior. However, for complex clock network logic, you can enable advanced clock feedthrough analysis by setting the following variable:

```
dc_shell> set_app_var test_fast_feedthrough_analysis true
```

This variable setting enables the DRC engine (which the `dft_drc` command runs internally) to detect clock feedthrough paths in the logic of the current design.

Enabling fast feedthrough analysis can help during both core creation and core integration:

- During core creation, feedthroughs from input port through complex logic to output port can be detected and included in the CTL model of the core.
- During core integration, if you have CTL-modeled cores that contain unidentified feedthrough paths, you can also set the `test_simulation_library` variable to configure the DRC engine to use a netlist simulation model for that core.

If you created the core with advanced feedthrough analysis, the CTL model should include any feedthrough paths through complex logic and you should not need to reanalyze the logic using a netlist simulation model. However, verify that the CTL model includes all expected feedthrough paths in this case.

Note:

Although feedthroughs that vary on a per-test-mode basis can be understood by DFT DRC and insertion of the current design, they cannot be described in the resulting CTL model.

Connecting the Scan-Enable Pins of Cores

When you insert DFT at a top level that contains cores, the cores already contain complete scan-enable networks. Instead of connecting the top-level scan-enable signal to target pins inside the core, DFT Compiler must connect to scan-enable signal pins at the core boundary.

You can use the `-usage` option of the `set_dft_signal` command to give DFT Compiler more information about the intended use of the scan-enable signal. Signals with a usage of `scan` enable scan cells during scan shift, while signals with a usage of `clock_gating` enable clock-gating cells during scan shift.

When scan-enable signals at the core and/or top level are defined with the `-usage` option, DFT Compiler attempts to determine which top-level signal should drive each core-level signal, using the priorities shown in [Table 22](#). The column headers along the top denote various core-level scan-enable usages. For each usage, that column shows the priorities used to determine how top-level signals are connected to that core-level signal.

Table 22 Core-Level and Top-Level Scan-Enable Usages With Priorities

Core Top	scan	clock_gating	scan plus clock_gating	No usage specified
scan	1		2	1
clock_gating		1		
scan plus clock_gating	2	2	1	2
No usage specified	3	3	3	3
New port created (test_se)	4^3	4^3	4^3	4^3

These connection priorities propagate signal usages upward through the hierarchy while preserving the original usage intent as much as possible. You can use the `set_dft_signal -connect_to` command and related options to specify specific source-to-pin signal connections that override these default signal connection behaviors.

See Also

- [Connecting the Scan-Enable Signal in Hierarchical Flows on page 232](#) for more information about how you can connect the scan-enable signal to cores

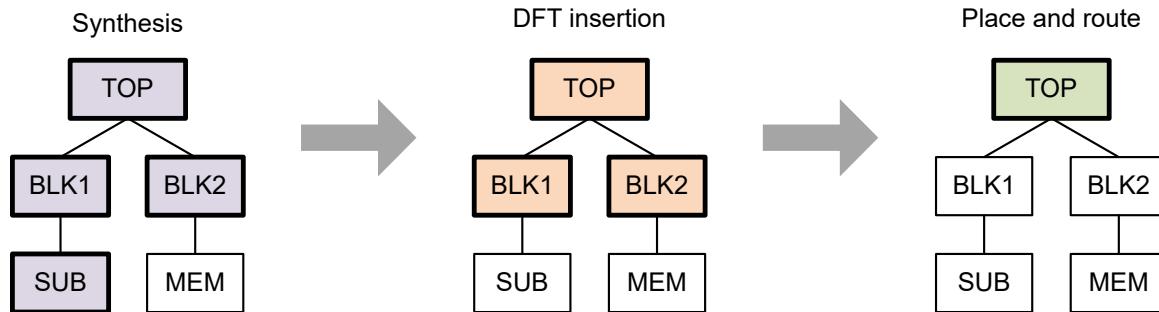
Hierarchical Synthesis, DFT Insertion, and Layout Flows

When designing your hierarchical flow, remember that synthesis, DFT insertion, and layout tool domains each have their own independently configurable hierarchical flows. Not all synthesized blocks require DFT insertion, and not all synthesized or DFT-inserted blocks require layout.

In [Figure 37](#), BLK1 uses bottom-up synthesis, top-down DFT insertion, and is placed and routed only as part of a higher level design.

3. *Only one new port is created for all core pin configurations requiring a new port; this new port will behave like a scan-enable signal with no usage specified for all further DFT integration purposes.*

Figure 37 Independent Hierarchical Synthesis, DFT, and Layout Flows



In addition, different blocks can use different tool features:

- For synthesis, some blocks can use wire load synthesis, some blocks can use DC Ultra synthesis, and some blocks can use synthesis in Design Compiler topological mode.
- For DFT, some blocks can use standard scan insertion, while other blocks can use compressed scan insertion.

For each tool domain, create a hierarchical flow that considers the needs of each design block. Also, consider interactions between the tool domains, such as

- Ensuring that mapped designs are provided to DFT insertion and layout
- Providing test-ready designs to DFT insertion for better results
- Including SCANDEF information when writing out DFT-inserted core designs to enable scan chain reordering in layout

When you write out a synthesis block abstraction of a DFT-inserted core in .ddc format, the .ddc file contains CTL model information for use by DFT insertion.

- For more information about hierarchical flows in Design Compiler topographical mode, see “Compile Flows in Topographical Mode” in the Design Compiler User Guide.
- For more information about hierarchical synthesis flows, see “Using Hierarchical Models” in the Design Compiler User Guide.

See Also

- [Generating SCANDEF Information in Hierarchical DFT Flows on page 640](#) For more information about using SCANDEF information in hierarchical flows

Linking Test Models to Library Cells

When complex library cells have built-in scan chains, Core Test Language (CTL) test models must be linked to the library cells for DFT Compiler to connect the scan chains at the top level.

If you have the original library source file, you can use the `read_lib -test_model` command to annotate test model information to a library cell when the library is read in. After the library has been read and annotated with the test model, a library .db file containing the test model can be written out. For example,

```
read_lib lib_file.lib -test_model [lib_cell_name:]model_file.ctl
write_lib lib -output lib_file.db
```

If the CTL model name differs from the intended library cell name, an optional library cell name prefix can be used to specify the model's intended library cell.

To link multiple test models to multiple cells within a single library, supply the model files to the `-test_model` option as a list:

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

Note:

The `read_lib` command is a Library Compiler command. To use it in a DFT flow, you must link the Design Compiler and Library Compiler tools together. For more information, see the *Synthesis Tools Installation Notes* and *Library Compiler Installation Notes*.

If you only have a compiled library .db file, you can use the `read_test_model` command to link a test model to an existing library cell or design in memory. Specify the library cell or design name with the `-design` option. The library cell name can be provided with or without the logic library name prefix. For example,

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

read_test_model -format ctl \
-design lib_cell_name model_file.ctl

read_test_model -format ctl \
-design logic_lib/lib_cell_name model_file.ctl
```

If you have subdesigns that are modeled using extracted timing models (ETMs), you can also link CTL test models to the ETM library cells just as you would with logic library cells.

You can use the `report_lib` command to determine if a library cell has CTL test model information applied to it, using either the `read_lib -test_model` or `read_test_model` command.

Note:

When you link a test model to a library cell with the `read_test_model` command, it takes precedence over any existing test model information present in the library.

Checking Library Cells for CTL Model Information

To determine if a library cell has CTL model information attached to it, read in the library, then run the `report_lib` command. If a library cell in the library has CTL model information attached, the report will indicate `ctl` in the attributes column, as shown in [Example 9](#).

Example 9 Simple report_lib Output for Library Cell With Test Model

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

Physical DFT Features in Design Compiler

When you use Design Compiler with physical libraries and design information, several DFT features directly use physical information to improve the DFT implementation.

[Table 23](#) shows the DFT features and their supported physical synthesis flows.

Table 23 DFT Features That Use Physical Information

DFT feature	Design Compiler in topographical mode?	Design Compiler Graphical?
Scan chain reordering Scan cells are ordered by physical proximity to minimize wire length. See Physical Reordering and Repartitioning on page 209 .	Yes	Yes
DFTMAX reduced-congestion codec Codecs are implemented with a logic structure that minimizes congestion. See Performing Congestion Optimization on Compressed Scan Designs on page 706 .	No	Yes
Test point insertion Test points are grouped by physical proximity to share registers. See Sharing Test Point Registers on page 303 .	Yes	Yes
Pipelined scan-enable signals Scan cells are grouped by physical proximity to share pipeline registers. See Implementation Considerations for Pipelined Scan-Enable Signals on page 353 .	Yes	Yes

See Also

- [Invoking the Synthesis Tool on page 105](#) for more information about invoking Design Compiler in topographical mode and Design Compiler Graphical

DFT Flows in DC Explorer

DC Explorer can improve your productivity during development of RTL and constraints by enabling fast synthesis in the early design stages. During this exploratory phase, you can also use DC Explorer to evaluate the DFT DRC readiness of the design.

In general, DC Explorer accepts all DFT specification commands that can be applied before DFT insertion. This includes (but is not limited to)

- Standard scan and compressed scan configuration commands
- Scan path and scan group definition commands
- Test-mode creation commands
- OCC controller configuration commands
- DFT partition commands
- Test point configuration commands
- The `create_test_protocol` command
- The `read_test_model` command
- The `read_test_protocol` command
- The `report_scan_path` command (when used to report user-specified scan structures)
- The `dft_drc` command
- The `write_test_protocol` command

Note that although some of these DFT specification commands do not affect pre-DFT DRC, they are accepted without warning or error. This allows typical pre-DFT DRC scripts to execute cleanly.

DC Explorer does not support the DFT preview or insertion commands or DFT commands that are run after DFT insertion. This includes

- The `preview_dft` and `insert_dft` commands
- The `report_scan_path` command (when used to report DFT-inserted scan structures)
- The `write_scan_def` command
- The `write_test_model` command

If you attempt to use an unsupported command that would not significantly affect the structure of the design, DC Explorer issues a warning:

```
de_shell> preview_dft
Warning: Command 'preview_dft' is not supported in DC Explorer. The
command is ignored. (DESH-009)
0
```

If you attempt to use an unsupported command that would significantly affect the structure of the design, DC Explorer issues an error:

```
de_shell> insert_dft
Error: Command 'insert_dft' is not supported in DC Explorer. (DESH-008)
0
```

IEEE Std 1149.1 and IEEE Std 1149.6 boundary-scan configuration and insertion, which is provided by the TestMAX DFT tool, is not supported in DC Explorer.

See Also

- [Invoking the Synthesis Tool on page 105](#) for more information about invoking DC Explorer

7

Running the Test DRC Debugger

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

Design Vision 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 GUI provides the debug environment for pre-DFT DRC violations, post-DFT DRC violations, and Core Test Language (CTL) models.

This chapter includes the following topics:

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

Starting and Exiting the Graphical User Interface

To invoke Design Vision and view test DRC results, you need to

- Execute the `design_vision` command or, for topographical mode, the `design_vision -topographical_mode` command 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 the `dft_drc` command on the Design Vision 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 Linux shell.

To invoke Design Vision directly from dc_shell, enter

```
dc_shell> gui_start
```

To use options with this command, see the *Design Vision User Guide* for further information.

Note:

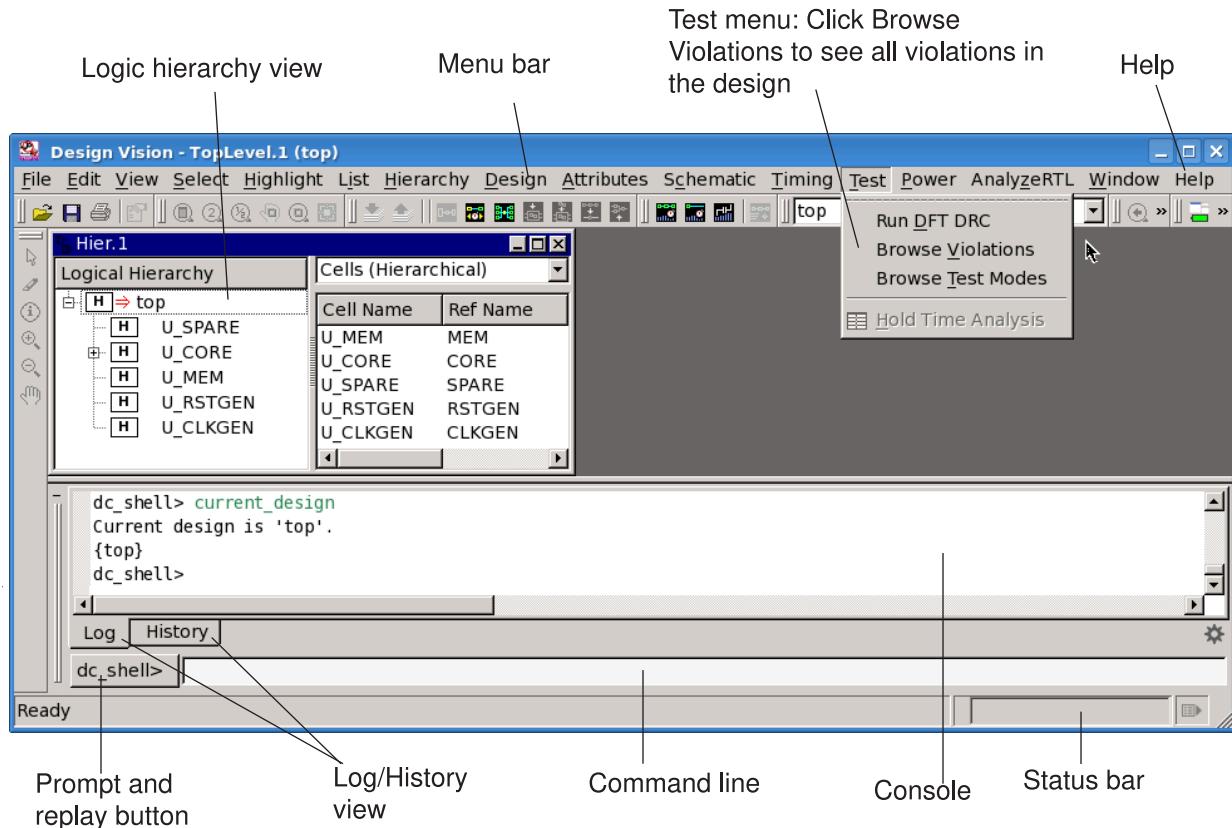
Before invoking Design Vision or opening the GUI, make sure you have correctly set your display environment variable. See the *Design Compiler User Guide* for information on setting this variable.

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. The GUI functions as a visual analysis tool to help you to visualize and analyze the violations in your design.

[Figure 38](#) shows the Design Vision window running in the Design Vision foreground.

Figure 38 The Design Vision 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

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 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 GUI provides an HTML-based browser 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 to use the GUI. Choose commands on the Test menu to view design violations and to open 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.

Examining DRC Violations

You can use the DRC violation browser to search for and view information about DRC violations in the current design.

To open the violation browser and view violations,

- Choose Test > Browse Violations

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

See Also

- [Viewing Design Violations on page 145](#) for more information about viewing and debugging DRC violations in the GUI

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,

1. 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.
 2. Select a test protocol name in the Test Modes list.
-

Viewing Design Violations

This topic covers the following:

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

Reporting DRC Violations

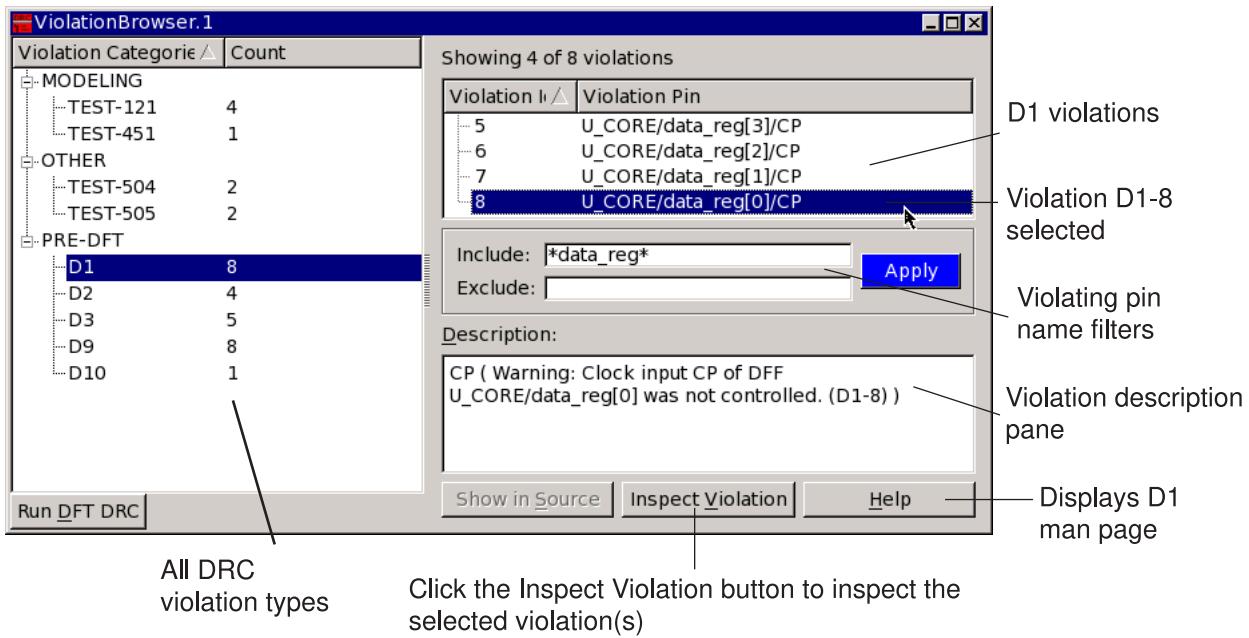
The violation browser lets you examine detailed information about violations and also provides a variety of tools for viewing the different aspects of a violation. The violation browser groups the warning and error messages into categories that help you find the problems you are concerned about.

To report the DRC violations,

- Choose Test > Browse Violations.

This opens the violation browser view window, as shown in [Figure 39](#).

Figure 39 Violation Browser View 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.

To see the violations:

1. 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.

2. Select a DRC violation type in the left pane.

A list of violating pins appears in the right pane.

3. (Optional) To filter the violating pins list, enter pin names or name patterns in the Include box, the Exclude box, or both, and click Apply.

- To show only violating pins that match the names or name patterns, enter them in the Include box.
- To suppress violating pins that match the names or name patterns, enter them in the Exclude box.

You can use the ? and * wildcard characters to create name patterns. Separate multiple names or name patterns with blank spaces.

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 DRC Violations on page 147](#).

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

Inspecting DRC Violations

You can analyze and debug DRC violations by using the violation inspector window. You can inspect multiple violations of the same type together. Use the schematic view to inspect the logic structure of the DRC violation, including pin data. You can also display waveforms for test_setup pin data.

This topic covers the following:

- [Viewing a Violation](#)
- [Viewing Multiple Violations Together](#)
- [Viewing CTL Model Scan Chain Information](#)
- [Viewing test_setup Pin Data Waveforms](#)

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.

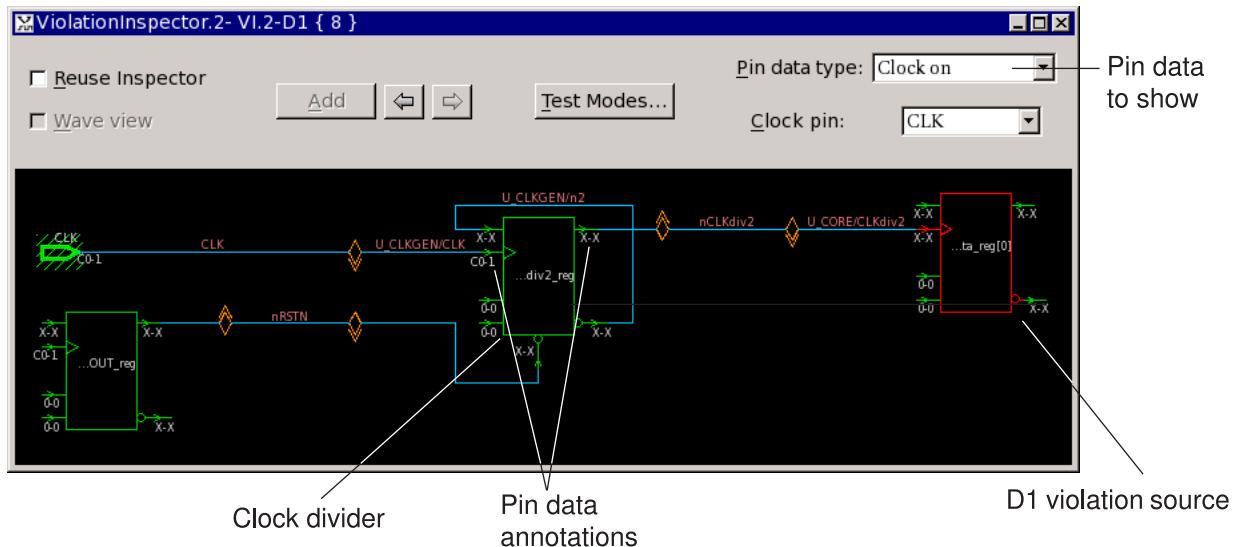
To open the violation schematic,

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

This opens the violation inspector window, as shown in [Figure 40](#).

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.

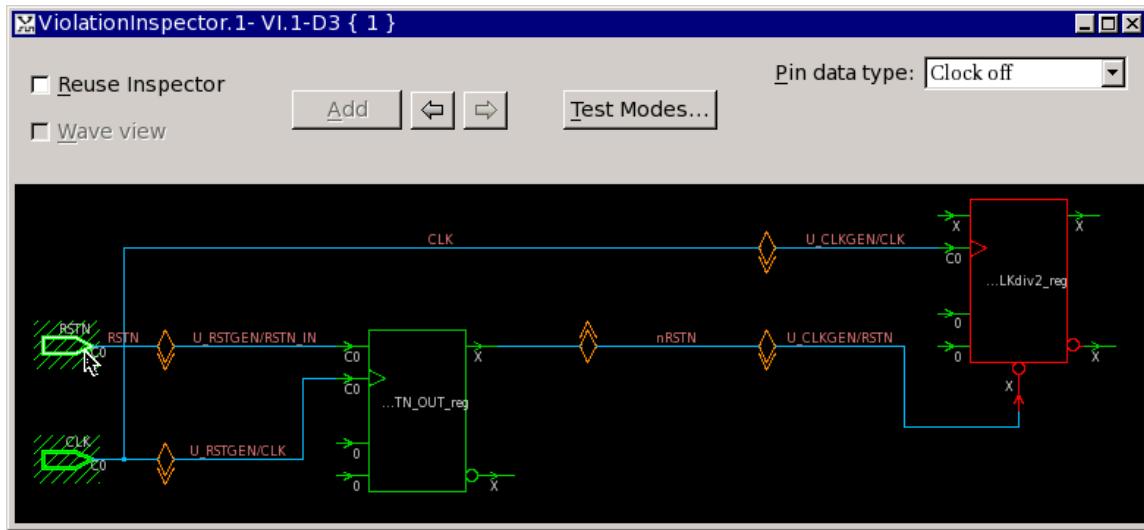
Figure 40 Violation Inspector: Viewing a Violation



- (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.

- If you define a DFT signal with the `set_dft_signal` command, the signal source is highlighted with a hatched fill pattern, as shown in [Figure 41](#).

[Figure 41](#) *Highlighted DFT Signal Sources*



Viewing Multiple Violations Together

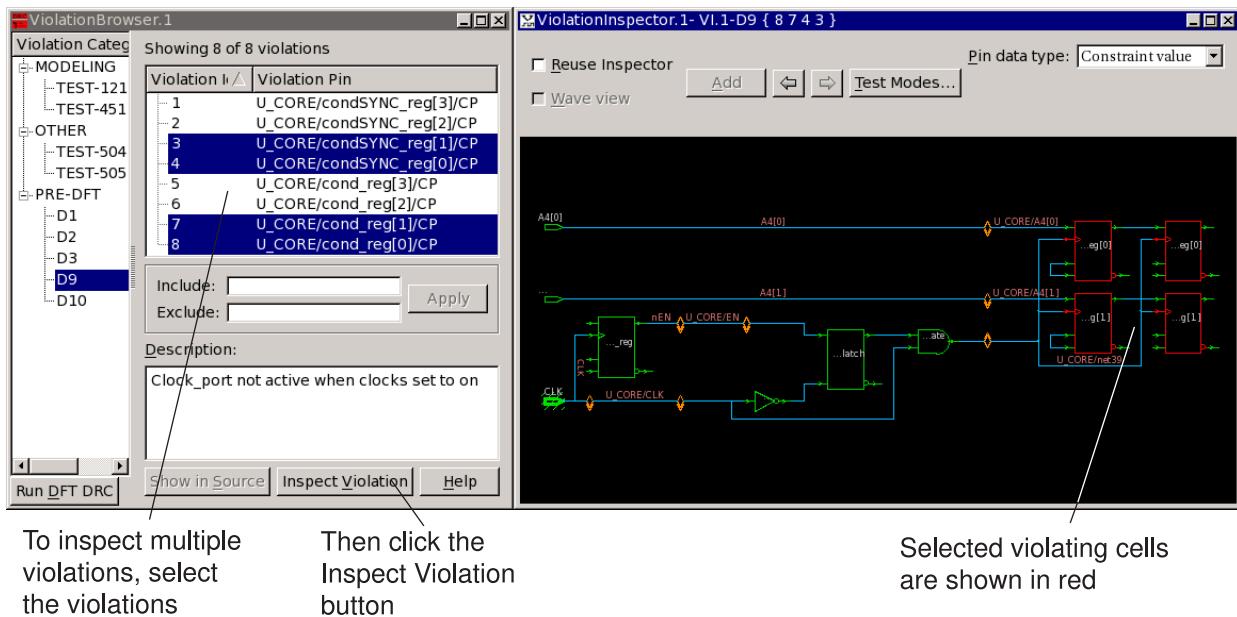
To view the path schematics for multiple violations,

- Shift-click to select multiple violation IDs in the violation browser.
- 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.

[Figure 42](#) shows the selection of multiple violations.

Figure 42 Viewing Multiple Violations



Viewing CTL Model Scan Chain Information

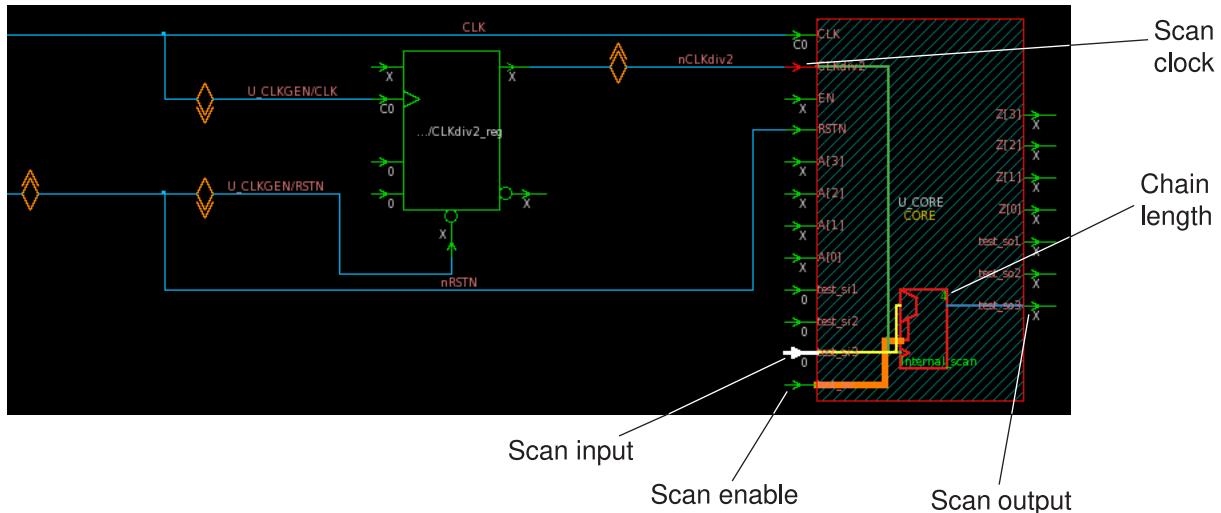
A CTL model provides information about scan cells and the test modes in which they are active. It also describes characteristics of each scan chain, such as the chain length and the scan-in, scan-out, scan clock, and scan-enable pins.

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

When you click on a scan-in or scan-out pin of a CTL-modeled cell, the tool displays an abstract representation of that scan chain with following information, as shown in Figure 43:

- Scan input
- Scan output
- Scan enable
- Scan clock

Figure 43 Displaying CTL Model Scan Chain Information



Clicking on scan clock or scan-enable pins of a CTL-modeled cell does not display any scan chain information.

Viewing test_setup Pin Data Waveforms

In the Violation Inspector window, the “Pin data type” menu lets you choose what data to annotate on pins in the design schematic. Most pin data types are one or three characters (test cycles) long. However, pin data from the test_setup procedure is arbitrarily long and cannot be annotated on the schematic.

When you select “Test setup” in the “Pin data type” list, a waveform viewer appears that displays pin data waveforms for one or more pins. You can add, remove, and group pins together. The waveform viewer is integrated with the schematic display, so you can explore the logic cone and add additional pins of interest as needed.

To display test_setup waveforms for pins,

1. Select one or more pin names in the violation browser.
2. Click the Inspect Violation button.
3. 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 44](#). You can adjust the relative heights of these views by dragging the split bar up or down.

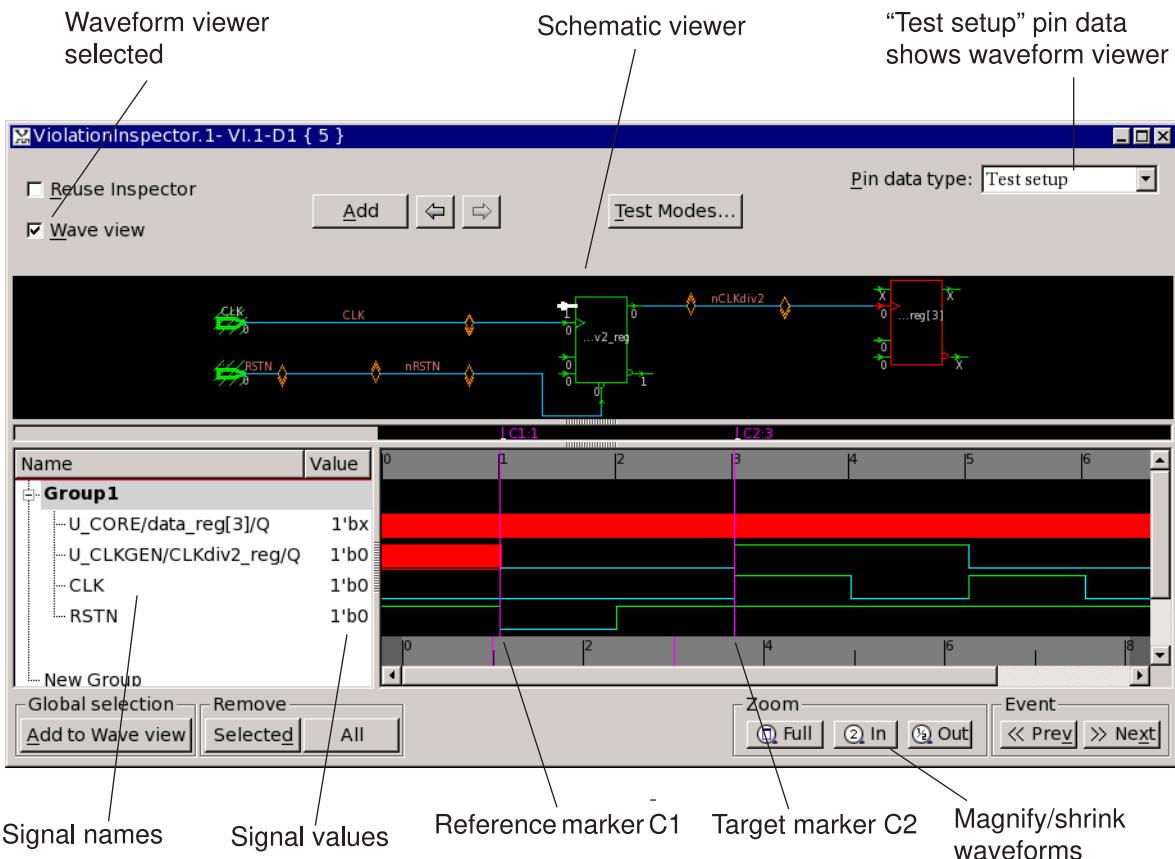
The waveform view consists of two panes: an expandable signal list on the left and the waveform viewer on the right. You can adjust the relative widths of the panes by dragging the split bars left or right.

4. Select one or more objects (pins, cells, nets, or buses) for the signals that you want to inspect.

5. 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 44 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 timescale.

- To move C1, left-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 Tools in the GUI

Detailed descriptions of the DFT-specific commands and options in the GUI are listed in this topic.

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
```

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

The `gui_wave_add_signal` 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]
```

To add a port object `i_rd`, for example, use the following syntax:

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

To add selected objects, use the following syntax:

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

guiViolationSchematicAddObjects

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

8

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 overhead of the scan structures on the design.

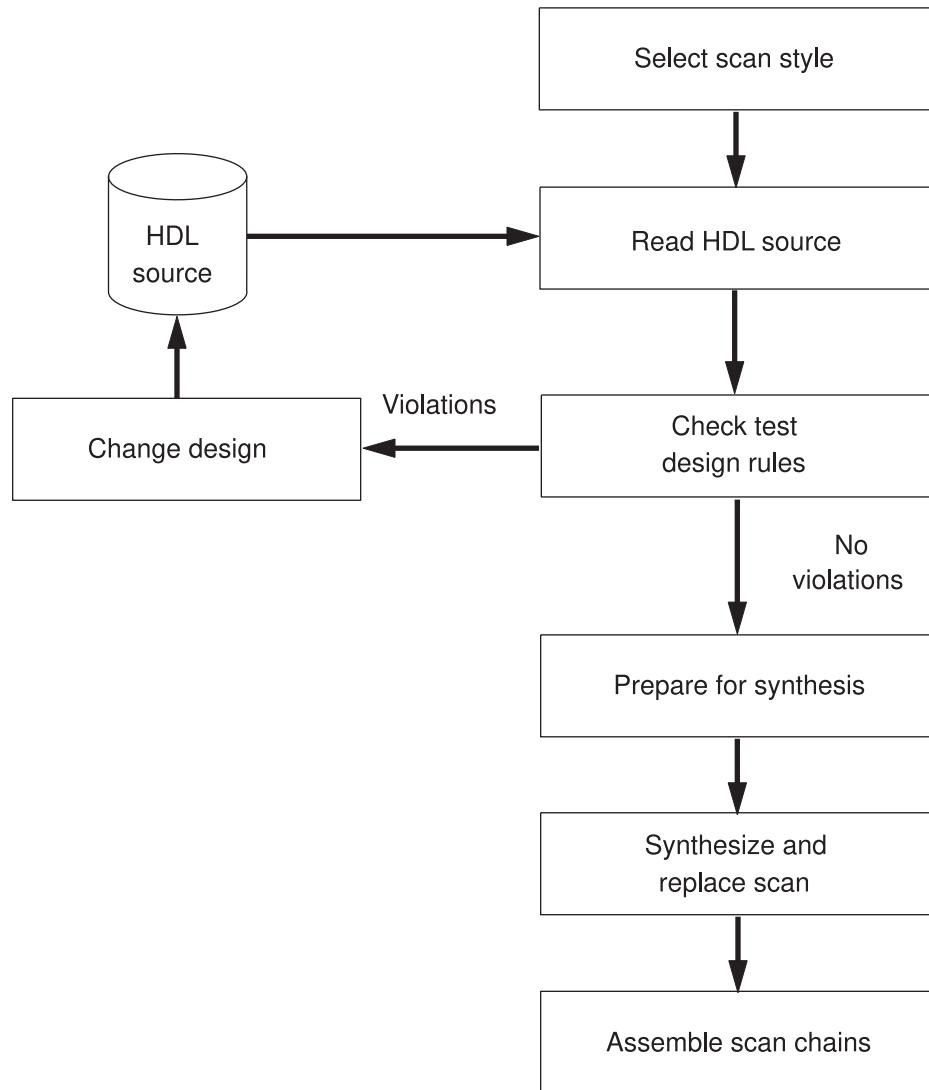
This chapter includes the following topics:

- [Scan Replacement Flow](#)
- [Preparing for Scan Replacement](#)
- [Specifying a Scan Style](#)
- [Verifying Scan Equivalents in the Logic Library](#)
- [Scan Cell Replacement Strategies](#)
- [Test-Ready Compilation](#)
- [Validating Your Netlist](#)
- [Performing Constraint-Optimized Scan Insertion](#)

Scan Replacement Flow

[Figure 45](#) shows the flow for the scan replacement process. This flow 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 190).

Figure 45 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 overhead of the scan structures on the design.

Preparing for Scan Replacement

This topic discusses what to consider before starting the scan replacement process, and it covers the following:

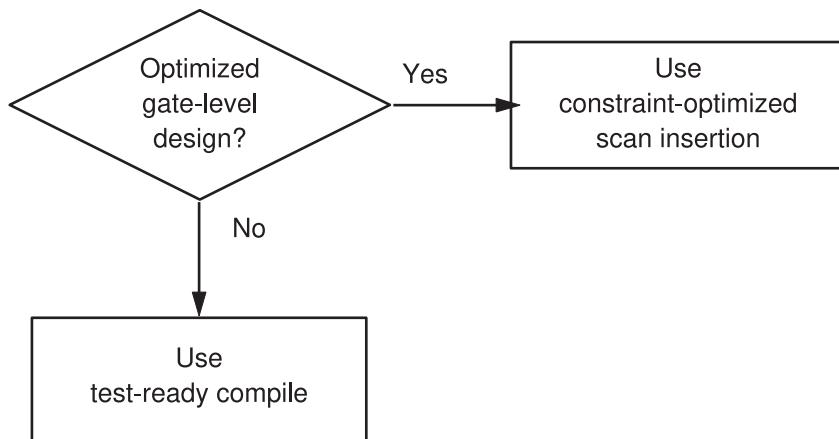
- [Selecting a Scan Replacement Strategy](#)
- [Identifying Barriers to Scan Replacement](#)
- [Preventing Scan Replacement](#)

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 46](#) shows how to determine the appropriate scan replacement strategy.

Figure 46 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.

See Also

- [Test-Ready Compilation on page 178](#) for more information about test-ready compilation
- [Performing Constraint-Optimized Scan Insertion on page 187](#) for more information about constraint-optimized scan insertion

Identifying Barriers to Scan Replacement

You should perform pre-DFT DRC by running the `dft_drc` command to identify conditions that prevent scan replacement. The following topics cover DRC violations that prevent scan replacement:

- [Logic Library Does Not Contain Appropriate Scan Cells](#)
- [Support for Different Types of Sequential Cells and Violations](#)
- [Attributes That Can Prevent Scan Replacement](#)
- [Invalid Clock Nets](#)
- [Invalid Asynchronous Pins](#)

See Also

- [Chapter 13, Pre-DFT Test Design Rule Checking](#) for more information about checking for test design rule violations prior to DFT insertion

Logic 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 warning 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 logic library does not contain scan cells.
- The logic library contains scan cells, but it does not provide a scan equivalent for the indicated nonscan cell.
- The logic library contains scan cells, but it incorrectly models the scan equivalent for the nonscan cell.
- The logic library contains scan cells, but all possible scan equivalents have the `dont_use` attribute applied.

If DFT Compiler cannot find scan equivalents for any sequential cell in the logic library, it generates the following warning message:

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

If you see this warning, check with your library vendor to see if the vendor provides a logic library that supports scan synthesis.

If DFT Compiler finds a scan cell in the logic 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 exact scan equivalent does not exist in the logic library or it has the `dont_use` attribute applied.
- The logic library has a problem. In that case, contact the library vendor for more information.

Support for Different Types of Sequential Cells and Violations

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 warning 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 warning 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 warning 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.

Cells that violate this requirement cause DFT Compiler to generate one of these warning 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)

Note that the cell might use different clocks for functional and test operations.

Note:

Your design will contain unsupported sequential cells only if you explicitly instantiate them. DFT Compiler does not insert unsupported sequential cells.

Attributes That Can Prevent Scan Replacement

The following attributes affect scan replacement:

- `scan_element == false`

The `scan_element` attribute is applied by the `set_scan_element` command. When set to `false`, it excludes sequential cells from scan replacement and scan stitching. The behavior depends on the type of cell the attribute is applied to:

- If the `scan_element` attribute is set to `false` on an unmapped sequential cell or a nonscan cell, the cell is never scan-replaced or scan-stitched.
- If the `scan_element` attribute is set to `false` on a test-ready cell,
 - Subsequent test-ready compile commands do not unscan it.
 - In wire load mode, the `insert_dft` command unscans it (unless the `set_dft_insertion_configuration -synthesis_optimization` option is set to `none`) and does not stitch it into scan chains.
 - In topographical mode, the `insert_dft` command keeps the cell scan-replaced (to minimize layout disturbance) but does not stitch it into scan chains.
- `dont_touch == true`

The `dont_touch` attribute is applied by the `set_dont_touch` command. When set to `true`, it prevents the cell type from being changed, which indirectly affects scan replacement. The behavior depends on the type of cell the attribute is applied to:

- Test-ready cell: If the `dont_touch` attribute is set to `true` on a test-ready cell, the cell is kept as a scan-replaced cell. If the cell is a valid scan cell, it is stitched into scan chains. If not, due to other directives such as `set_scan_element false`, it remains as an unstitched test-ready cell.
- Nonscan cell: If the `dont_touch` attribute is set to `true` on a nonscan cell, the cell is kept as a nonscan cell. It is not scan-replaced or stitched into scan chains. Pre-DFT DRC notes such cells with the following information message:

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

Nonscan cells identified as shift register elements can be stitched into scan chains.

- Unmapped sequential cell: If the `dont_touch` attribute is set to `true` on an unmapped sequential cell before the initial compile, the attribute prevents the cell from being mapped. As a result, DFT insertion fails with the following error message:

Error: DFT insertion isn't supported on designs with unmapped cells. (TEST-269)

The `dont_touch` attribute is ignored when an identified shift register is split by the scan architect; the head scan flip-flops of any new shift register segments are scan-replaced even if they have the `dont_touch` attribute applied.

A `dont_touch` attribute on the top-level design does not affect scan replacement of the design.

Note:

Although the `-exclude_elements` option of the `set_scan_configuration` excludes cells from scan stitching, it does not prevent scan replacement, and it does not cause excluded test-ready cells to be unscanned. To prevent cells from being scan-replaced, use the `set_scan_element false` command.

Nonscan sequential cells generally reduce the fault coverage results for full-scan designs. If you do not want to exclude affected cells from scan replacement, remove the script commands that apply the attributes, then rerun the script.

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.

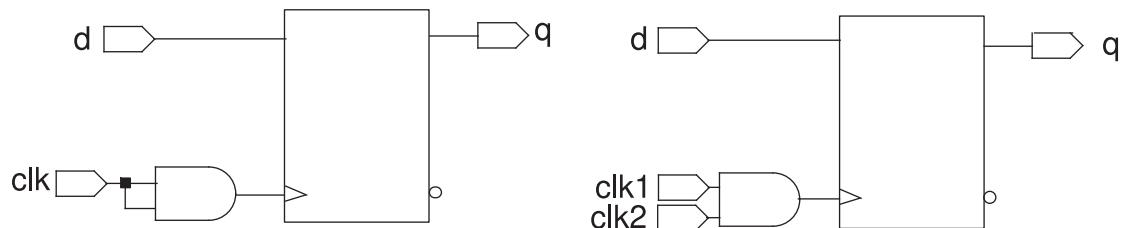
DFT Compiler does not support the use of组合ally combined clock signals, even if the same port drives the signal.

Note:

If the same port drives the combinationally combined clock signal, as shown in the design on the left in [Figure 47](#) or the design in [Figure 48](#), DFT Compiler does not detect the problem in nonscan or unrouted scan designs.

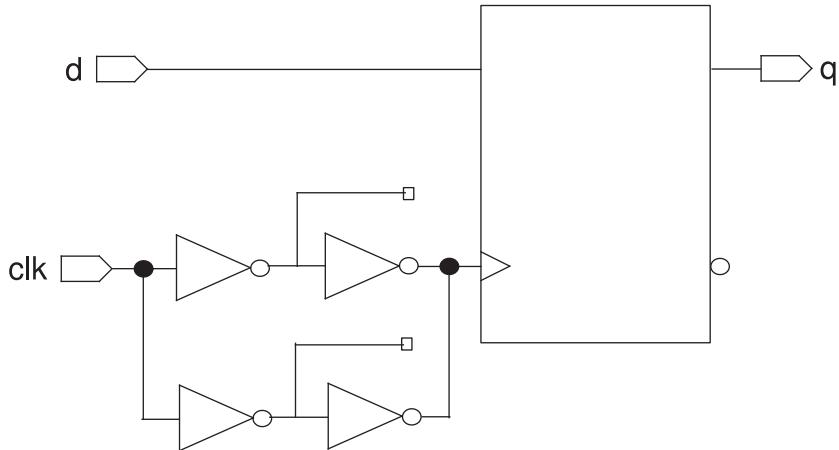
[Figure 47](#) shows design examples that use combinationally combined clocks. When multiple clock signals drive a clock net, DFT Compiler forces the net to X and cells clocked by the net become black-box sequential cells.

Figure 47 Examples of Combinationally Combined Clock Nets



DFT Compiler supports the use of reconvergent clocks, such as clocks driven by parallel clock buffers. [Figure 48](#) shows a design example that uses a reconvergent clock net.

Figure 48 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 combinational 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 warning 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 gated test 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 warning 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.

See Also

- [Using AutoFix on page 333](#) for more information about fixing uncontrollable clocks with AutoFix

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 warning 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

You can use the following attributes to prevent scan replacement during test-ready compile and DFT insertion:

- `scan_element == false`
- `dont_touch == true`

Setting the `scan_element` attribute to `false` prevents a cell from being scan-replaced and scan stitched. Setting the `dont_touch` attribute to `true` on a nonscan cell prevents it from being scan replaced and scan stitched.

See Also

- [Attributes That Can Prevent Scan Replacement on page 162](#) for more information about synthesis and test attributes that can affect scan replacement

Specifying a Scan Style

This topic explains the process for selecting and specifying a scan style for your design. It covers the following:

- [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 topics:

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

Note:

The LSSD scan style includes the LSSD and clocked LSSD scan styles.

This topic briefly describes each scan style.

See Also

- [Chapter 4, Scan Styles](#) for more information about scan styles

Multiplexed Flip-Flop Scan Style

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

- Multiplexed flip-flop is the most widely known and understood scan style.
- Multiplexed flip-flop scan cells are easy to design and characterize, as they consist of a conventional flip-flop plus a data selection MUX.

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 flip-flops and 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 flip-flops, master-slave flip-flops, and 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 scan replacement of 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-Enabled LSSD Scan Style

DFT Compiler supports scan-enabled level-sensitive scan design (LSSD) equivalents for D flip-flops. This scan style is similar to the LSSD scan style, except that a global scan-enable signal is used to repurpose the functional clock as the slave test clock in test mode. 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 scan-enabled 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 only supports the scan replacement of flip-flops.

When you use the scan-enabled 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 logic library?

To make it possible to implement internal scan structures in the scan style you select, appropriate scan cells must be present in the logic 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 Logic Library on page 170](#).

- What is your design style?

If your design is predominantly edge-triggered, use the multiplexed flip-flop, clocked scan, clocked LSSD, or scan-enabled 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 logic library?

The quality and accuracy of the scan and nonscan sequential cell models in the Synopsys logic 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 Library Compiler sequential modeling syntax supports only complete functional modeling for multiplexed flip-flop scan cells.

When the logic 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 logic library you are using, see your ASIC vendor. For information on creating logic 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

DFT Compiler uses the selected scan style to perform scan synthesis. A scan style dictates the appropriate scan cells to insert during optimization. This scan style is used on all modules of your design.

There are four types of scan styles available in DFT Compiler, shown in [Table 24](#).

Table 24 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>
Scan-enabled level-sensitive scan design	<code>scan_enabled_lssd</code>

The default style is multiplexed flip-flop. To specify another scan style, use the `-style` option of the `set_scan_configuration` command. For example,

```
dc_shell> set_scan_configuration -style clocked_scan
```

Verifying Scan Equivalents in the Logic Library

Before starting scan synthesis, you need to confirm that your logic library contains scan cells and then verify that the scan cells are suitable for the selected scan style.

This topic covers the following:

- [Checking the Logic Library for Scan Cells](#)
- [Checking for Scan Equivalents](#)

Checking the Logic Library for Scan Cells

You can determine whether the logic 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 logic 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 logic library. A scan output pin can have a `signal_type` attribute of either `test_scan_out` or `test_scan_out_inverted` in the logic 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_pins 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 logic library. If the logic libraries identified in the `target_library` variable do not contain scan cells, the `dft_drc` command generates the following warning message:

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

You must a design loaded and linked before you run the `dft_drc` command.

If your logic library does not contain scan cells, check with your semiconductor vendor to see if the vendor provides a logic library that supports test synthesis.

Checking for Scan Equivalents

To verify that the logic 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 logic library does not contain a scan equivalent for a sequential cell in a nonscan design or the scan equivalent has the `dont_use` attribute applied, the `dft_drc` command generates the following warning 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 logic library contains scan equivalents only for JK flip-flops. [Example 10](#) shows the warning message issued by the `dft_drc` command, along with the scan equivalent mappings to the available scan cells.

Example 10 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 topic describes how to select the set of scan cells and multibit components to use in your scan replacement strategy. It covers the following:

- [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 topic covers the following:

- [Restricting the List of Available Scan Cells](#)
- [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` value is the list of scan cells that the `compile -scan` command 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` value 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 the `insert_dft` command or by subsequent synthesis operations such as the `compile -incremental` command. If the `-exact` option 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 the `-exact` option is not used, the optimization algorithm is free to use any scan cell in the target library.

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 the `compile -scan` command 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 use the `-scan` option with 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, the Design Compiler and DFT Compiler tools 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. 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 topic covers the following:

- [What Are Multibit Components?](#)
- [How DFT Compiler Creates Multibit Components](#)
- [Controlling Multibit Test Synthesis](#)

- [Performing Multibit Component Scan Replacement](#)
- [Disabling Multibit Component Support](#)

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 synthesis. 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. 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. For further information, see the *Design Compiler User Guide*.

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.

For more information about multibit inference from RTL, see HDL Compiler for Verilog User Guide.

See Also

- [“Multibit Register Synthesis and Physical Implementation Application Note”](#) for detailed information on multibit cells and flows across multiple tools

How DFT Compiler Creates 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 groups 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.
- 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 topic 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.

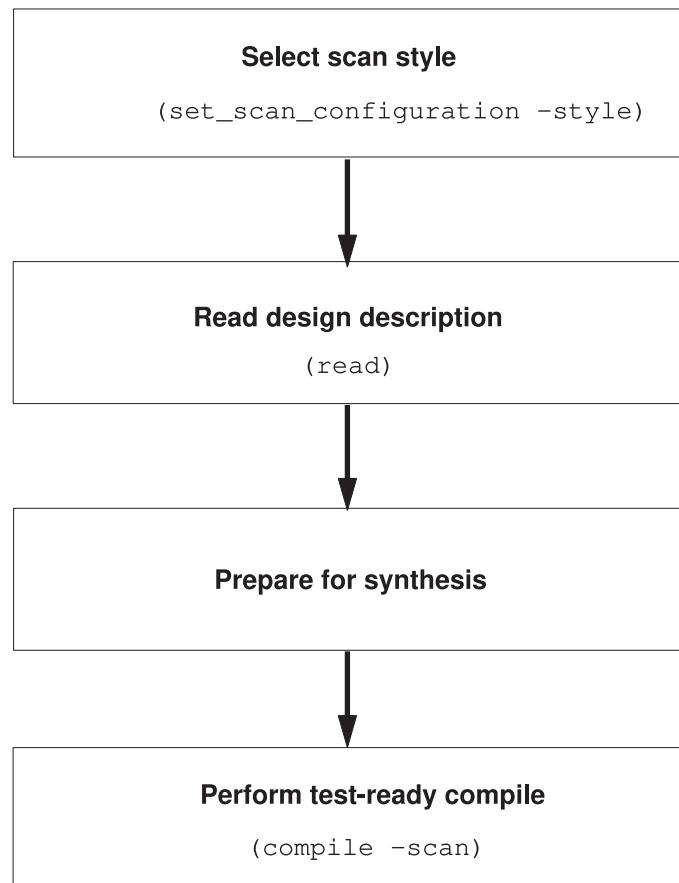
Typically, the input to test-ready compile is an HDL design description. You can also perform test-ready compile on a nonscan gate-level netlist that requires optimization. For example, a gate-level netlist resulting from technology translation usually requires logic

optimization to meet constraints. In such a case, use test-ready compile to perform scan replacement.

The Test-Ready Compile Flow

Figure 49 shows the test-ready compile flow and the commands required to complete this flow.

Figure 49 *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 167](#).

- Prepare for logic synthesis

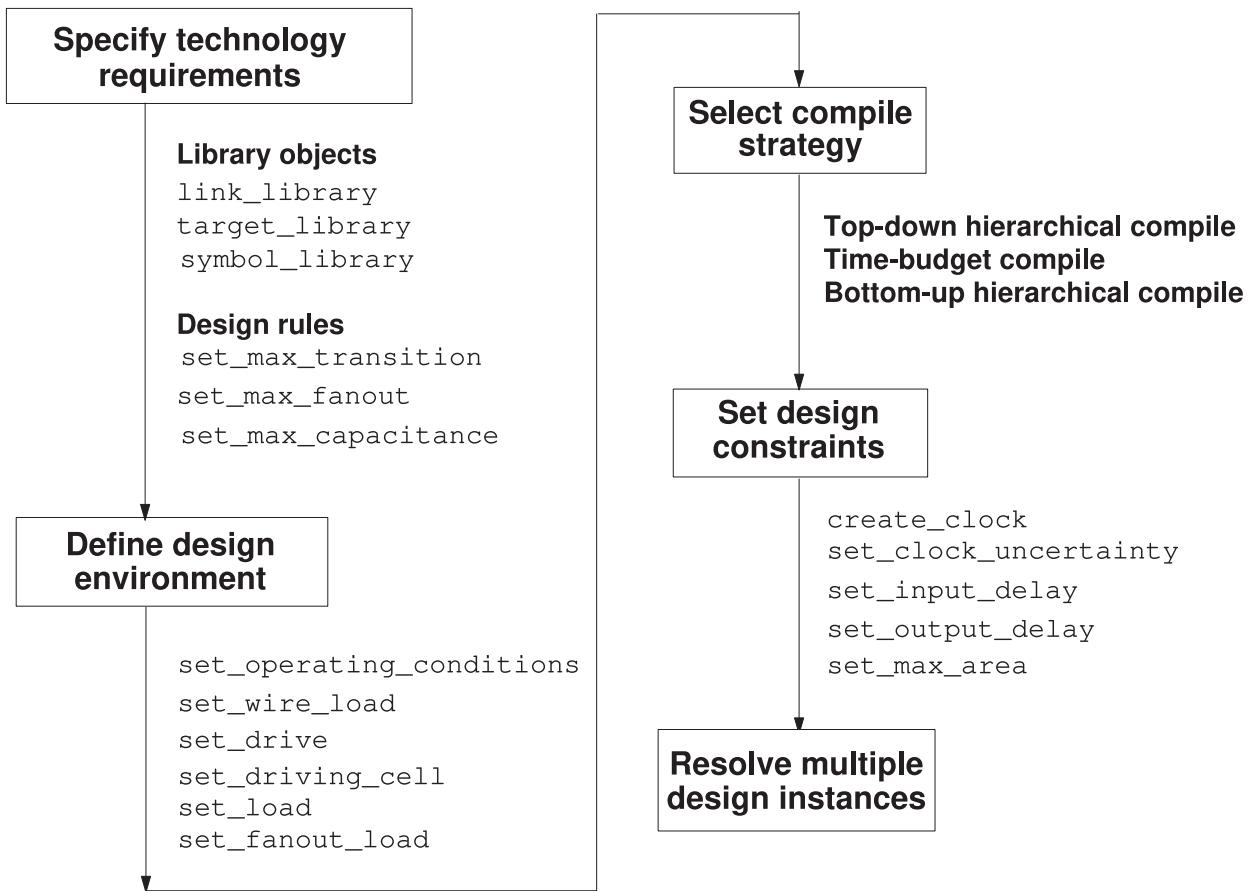
For information about preparing for logic synthesis, see [Preparing for Test-Ready Compile on page 180](#).

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 50](#) shows the synthesis preparation steps. For more information about these steps, see the *Design Compiler User Guide*.

Figure 50 Synthesis Preparation Steps



Performing Test-Ready Compile in the Logic Domain

The `compile -scan` command invokes test-ready compile. You must enter this command from the `dc_shell` command line; the Design Analyzer menus do not support the `-scan` option.

```
dc_shell> compile -scan
```

For details of how to constrain and compile your design, see Design Compiler User Guide.

Controlling Test-Ready Compile

You can use the following variable and commands to control scan implementation by `compile -scan`:

- `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 -preserve_multibit_segment`

The `set_scan_configuration -style` command determines which scan style the `compile -scan` command uses for scan implementation.

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 -architecture rtl switch
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 173](#).

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 11](#) describes a small design containing two flip-flops: one a simple D flip-flop and one a flip-flop with a multiplexed data input.

Example 11 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 12](#) shows the VHDL equivalent to the Verilog design example provided in [Example 11](#).

Example 12 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;
```

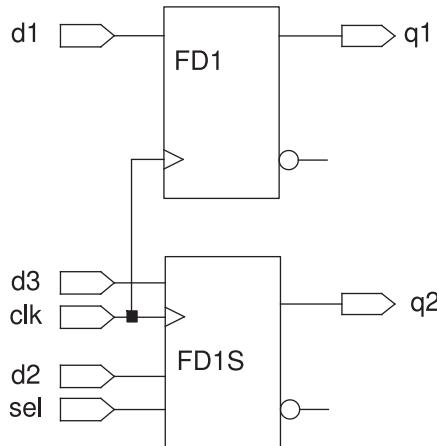
```
    end process;
end RTL;
```

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 51](#) shows the result of the default compile process on the design example. Design Compiler synthesis uses the D flip-flop (FD1) and the multiplexed flip-flop scan cell (FD1S) from the class logic library to implement the specified functional logic.

Figure 51 Gate-Level Design: Default Compile



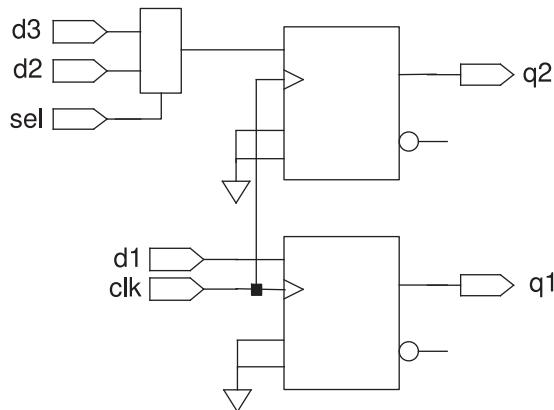
Using default compile increases the scan replacement runtime and can result in sequential cells that do not have scan equivalents.

To invoke test-ready compile, specify the scan style before optimization and use the `-scan` option of the `compile` command:

```
dc_shell> set_scan_configuration -style multiplexed_flip_flop
dc_shell> compile -scan
```

[Figure 52](#) shows the result of the test-ready compile process on the design example.

Figure 52 Gate-Level Design: Test-Ready Compile



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 so that the functional data input pins are active
- Ties the inactive scan input pins (SI) to logic 0

During scan routing, DFT Compiler replaces the temporary scan connections with the final scan connections.

A scan equivalent might not exist for the exact implementation defined, such as for the simple D flip-flop in the previous example. In that case, DFT Compiler might use a scan cell that can be logically modified to meet 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 13](#) 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 13 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 topic 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.

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, see man page for the `link` command.
- For more information about missing references and link errors, see 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 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.

Constraint-optimized scan insertion is covered in the following topics:

- [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.

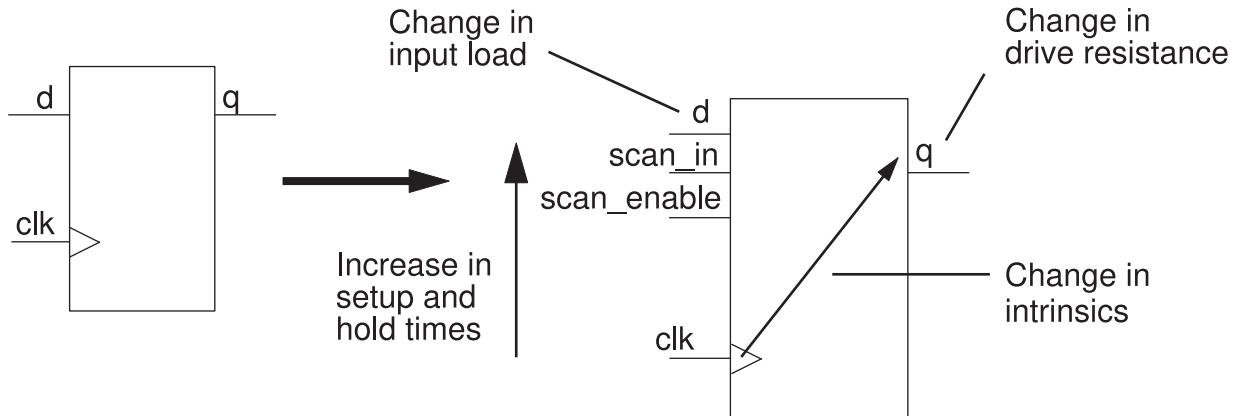
Locating Scan Equivalents

To perform scan replacement, constraint-optimized scan insertion first locates simple scan equivalents by using the identical-function method. If this method does not achieve scan replacement, then sequential-mapping-based scan replacement is used.

Like test-ready compile, constraint-optimized scan insertion supports degeneration of scan cells to create the required scan equivalent functionality.

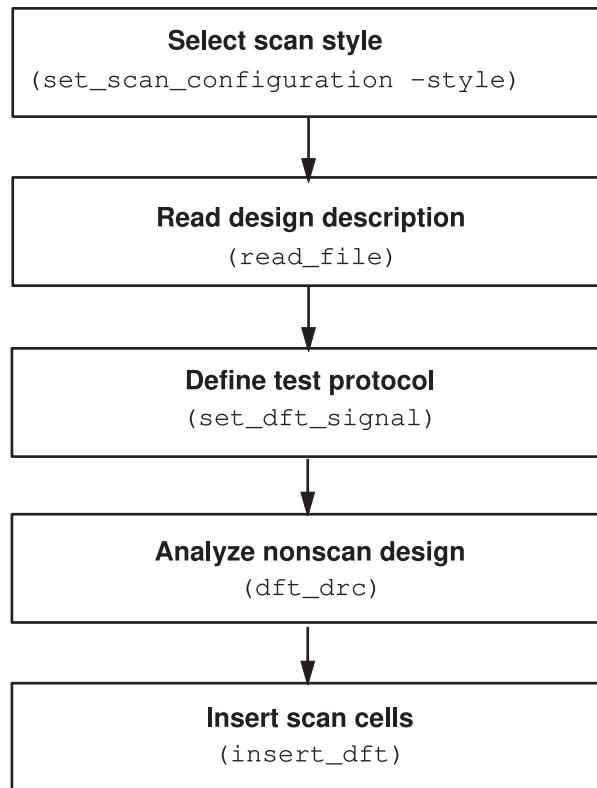
Replacing sequential cells with their scan equivalents modifies the design timing, as shown in [Figure 53](#). DFT Compiler performs scan-specific optimizations to reduce the timing overhead of scan replacement. By using focused techniques, constraint-optimized scan insertion optimizes the scan logic faster than the incremental compile process could.

Figure 53 Timing Changes Due to Scan Replacement



[Figure 54](#) shows the flow used to insert scan cells with constraint-optimized scan insertion and the commands required to complete this flow.

Figure 54 Constraint-Optimized Scan Insertion Flow (Scan Replacement Only)



For details about scan replacement methods, see “Performing Test-Ready Compile” in the Design Compiler User Guide.

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 timing 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 runtime can occur.

- Select a scan style.
- Identify barriers to scan replacement.

Scan Insertion

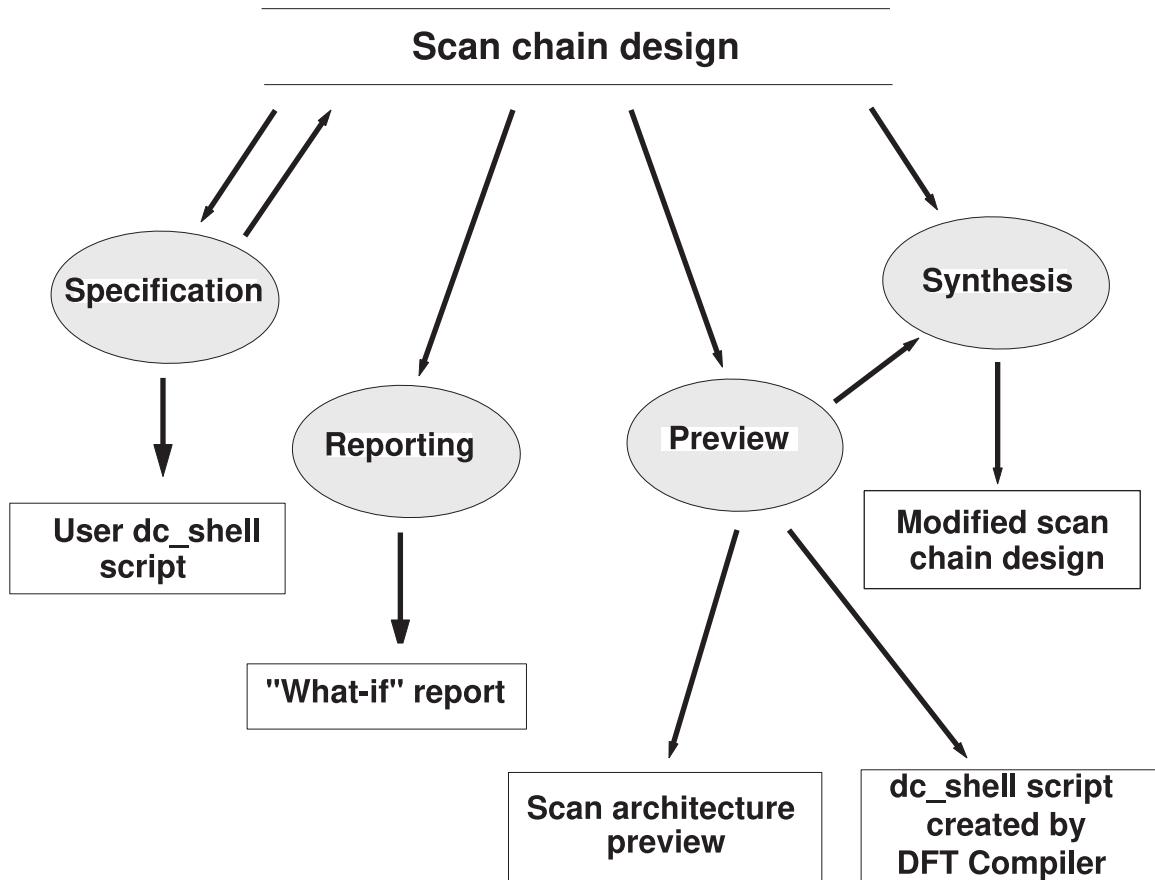
To alter a default scan design, you must specify changes to the scan configuration. You can make specifications at any point before scan synthesis. This topic describes the specification commands you can use.

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 55](#) shows this specify-preview-synthesize process.

Figure 55 The Scan Insertion Process



[Example 14](#) is a basic scan insertion script.

Example 14 Basic Scan Insertion Script

```
current_design Top
set_dft_configuration -fix_set enable -fix_reset enable
set_scan_configuration -chain_count ...
create_test_protocol -infer_clock -infer_asynch
dft_drc
preview_dft
insert_dft
dft_drc
report_scan_path -view existing_dft -chain all
report_constraints -all_violators
```

In this example, the following DFT configuration command enables AutoFix. For more information, see [Using AutoFix on page 333](#).

```
dc_shell> set_dft_configuration -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 topics 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 the `insert_dft` command 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 `preview_dft` command performs the following 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.
- 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 the `preview_dft` command, you can use the `-script` option to 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.

Caution:

The `preview_dft` command does not annotate the design database with test information. 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 one time before executing the `insert_dft` command. Executing the `dft_drc` command provides information on circuit testability before inserting scan into your design.

9

Architecting Your Test Design

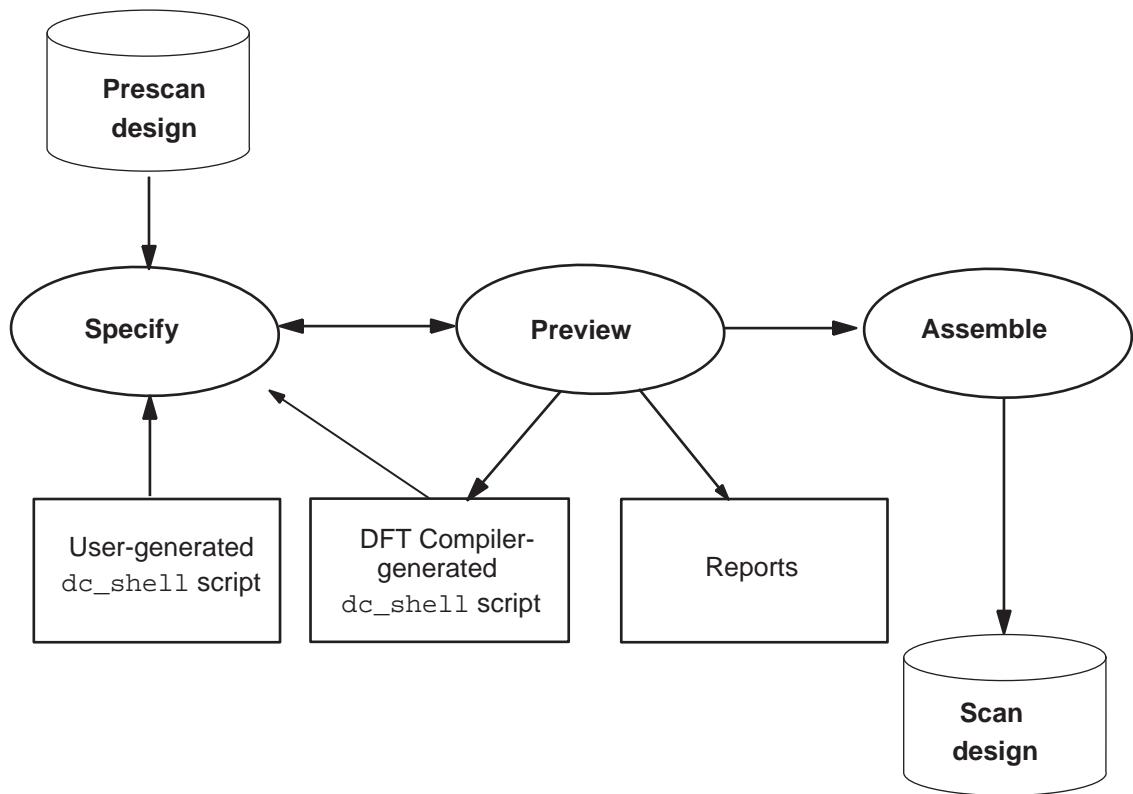
This chapter describes the basic processes involved in configuring and architecting your test design for scan insertion.

This chapter includes the following topics:

- [Configuring Your DFT Architecture](#)
- [Architecting Scan Chains](#)
- [Architecting Scan Signals](#)
- [Architecting Test Clocks](#)
- [Configuring Clock-Gating Cells](#)
- [Specifying a Location for DFT Logic Insertion](#)
- [Partitioning a Scan Design With DFT Partitions](#)
- [Modifying Your Scan Architecture](#)

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 56](#) shows the basic flow of the scan chain generation process.

Figure 56 Scan Chain Generation Process



Configuring Your DFT Architecture

Before you run scan insertion, you need to configure your DFT architecture. This topic includes the following topics related to the configuration process:

- [Defining Your Scan Architecture](#)
- [Specifying Individual Scan Paths](#)

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 pre-DFT DRC test design rules
dft_drc

# preview the scan structures
preview_dft

# assemble the scan structures
insert_dft

# check post-DFT DRC test design rules
dft_drc
```

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 options 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 before running the `insert_dft` command 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 Synopsys Timing Constraints and Optimization User Guide.

Defining Constant Input Ports During Scan

If your design requires a signal to be held constant to enable DFT logic or satisfy test design rules, use the `set_dft_signal` command to define a constant or test-mode signal.

See Also

- [Chapter 13, Pre-DFT Test Design Rule Checking](#) for more information about running test design rule checking prior to DFT insertion

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 from scan replacement. By default, all nonviolating sequential cells with equivalent scan cells are scan-replaced. You can specify leaf cells, hierarchical cells, references, library cells, and designs.

Use the `set_scan_element` command sparingly. For best results, use the command only on leaf or hierarchical cells.

- `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`

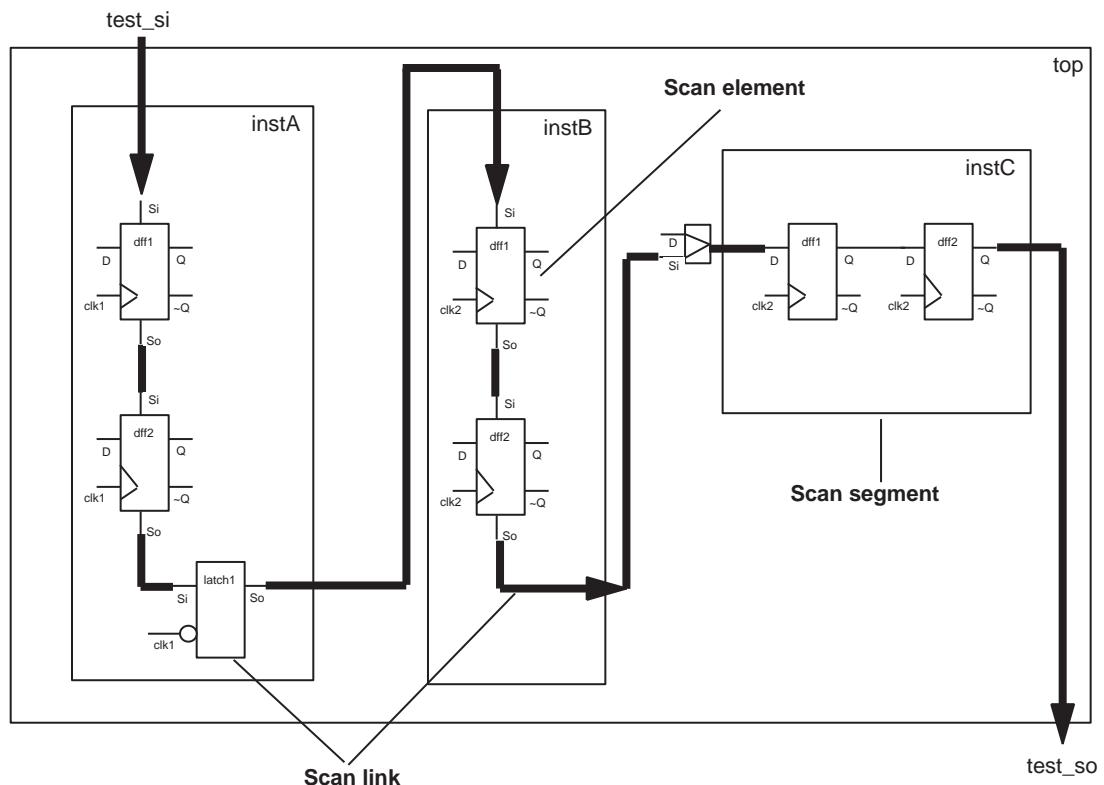
Use this command to specify desired port connections and scan chain assignments for test signals.

- `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 57](#) illustrates the scan path components.

Figure 57 Scan Path Components



[Figure 57](#) 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 topics in this chapter discuss some of the situations you might encounter during scan specification. See [Chapter 8, Performing Scan Replacement](#), for scan style selection considerations.

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 topic covers the following topics related to architecting scan chains:

- [Controlling the Scan Chain Length](#)
- [Determining the Scan Chain Count](#)
- [Defining Individual Scan Chain Characteristics](#)
- [Balancing Scan Chains](#)
- [Physical Reordering and Repartitioning](#)
- [Controlling the Routing Order](#)
- [Retiming Scan-Ins and Scan-Outs to the Leading Clock Edge](#)
- [Routing Scan Chains and Global Signals](#)
- [Rerouting Scan Chains](#)
- [Stitching Scan Chains Without Optimization](#)
- [Scan Stitching Only Scan-Replaced Cells](#)
- [Using Existing Subdesign Scan Chains](#)
- [Uniquifying Your Design](#)
- [Reporting Scan Path Information on the Current Design](#)

Controlling the Scan Chain Length

You can globally specify 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 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, the `insert_dft` command 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 the scan chain lengths 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.

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 standard scan configured for multiple test modes. It is not currently supported with DFTMAX compressed scan modes. Do not use this feature with DFTMAX scan compression.

Note the following properties of the `-exact_length` option:

- This option disables scan chain balancing.
- The `-exact_length` option should not be used with the `-max_length` or `-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 results cannot be guaranteed when this option is used on designs containing complex segments.

Determining the Scan Chain Count

You can specify the number of scan chains. DFT Compiler attempts to create the specified number of scan chains while minimizing the longest scan chain length. 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` and `-chain_count` options are mutually exclusive. If you use both options, the `-max_length` option takes precedence over the `-chain_count` option.

Defining Individual Scan Chain Characteristics

Typically, scan chains are configured using global chain count or chain length settings. Use the `set_scan_path` command to specify one or more additional requirements for individual scan chains in the current design.

The `set_scan_path` command enables you to

- Specify a name for a scan chain
- Allocate scan cells, scan segments, 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 individual exact scan chain lengths
- 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`) apply the most recent command.

For example, the following command sets the length of an individual scan chain:

```
dc_shell> set_scan_path C1 -exact_length 40
```

Balancing Scan Chains

The `insert_dft` command always attempts to balance the number of cells in each scan chain. However, some scan chain requirements can limit or disable balancing, such as

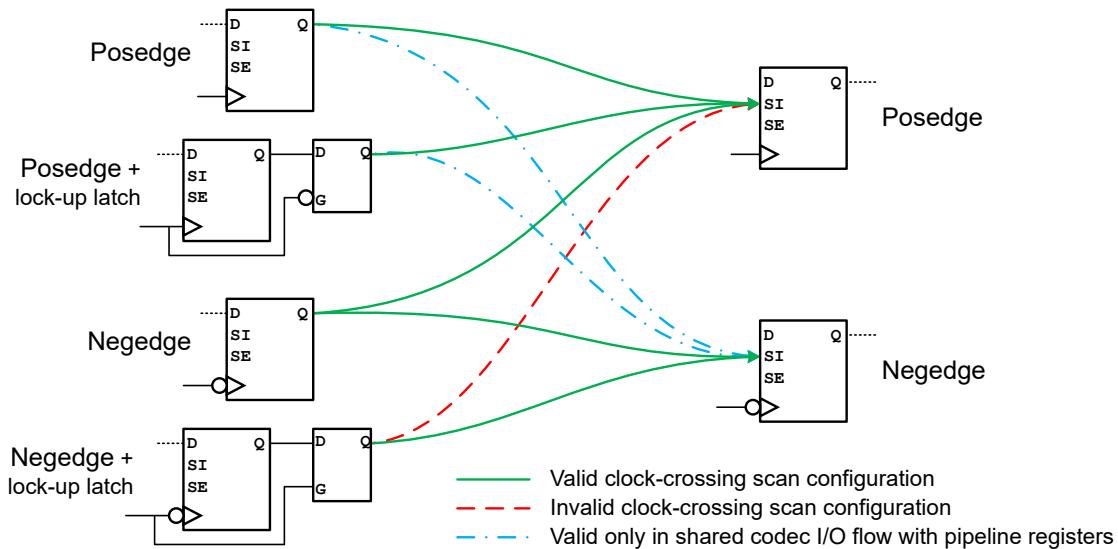
- Disabling clock mixing or clock edge mixing
- Defining scan chains with the `set_scan_path` command
- Using test models
- Using a hierarchical DFT insertion flow
- Specifying a global scan chain exact length

When overriding the default behavior, always use the `preview_dft` command to verify that the result meets your requirements.

Concatenating Scan Cells and Segments

When concatenating scan cells or segments to form scan chains, the `insert_dft` command can join them using only the scan cell sequences shown in [Figure 58](#). The tool might insert additional synchronization elements, such as lock-up latches or retiming flip-flops (not shown), to create the sequences.

Figure 58 Valid Scan Cell and Segment Concatenation Sequences



The irregularly dashed blue lines indicate sequences allowed only in the specific scenario described in [SolvNet article 1656177, "Why Does `insert_dft` Add Extra Retiming Registers in a Shared Codec I/O Flow?"](#)

This figure is relevant within a single clock domain or between clock domains with the same test clock waveforms. Between test clocks with differing waveforms, the tool determines the valid scan cell sequences using the waveform timing.

Multiple Clock Domains

The *clock edge* of a scan cell represents both the clock identity and the active clock edge of the cell. For multiplexed flip-flop designs, DFT Compiler allocates cells to scan chains based on clock edges by default. You can override this default behavior by using the `set_scan_configuration -clock_mixing` command.

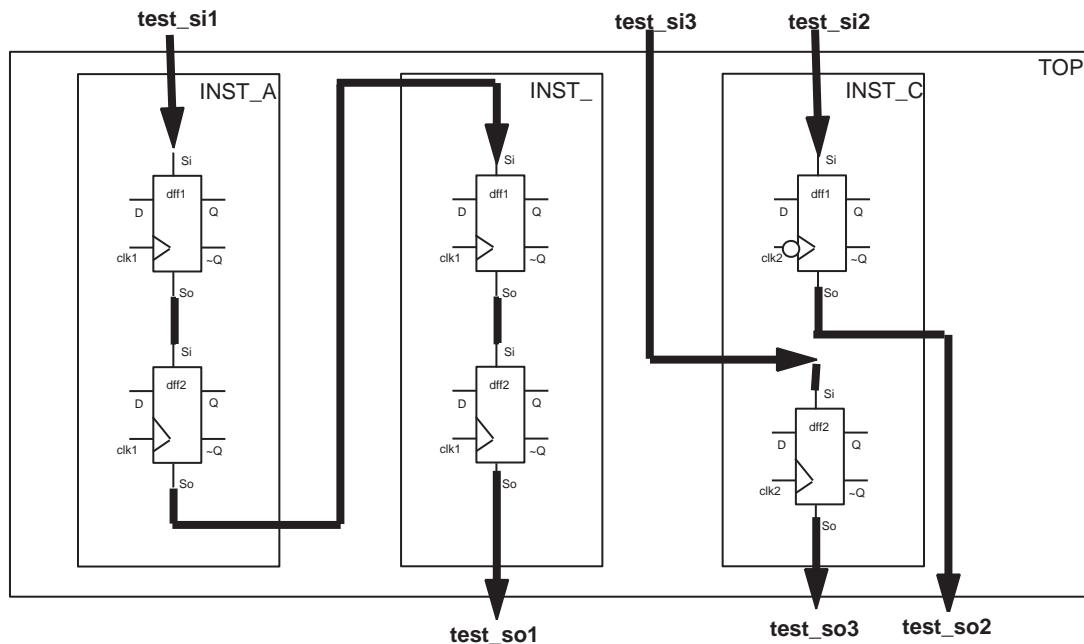
For example, assume that you have a design with three clock domains and your desired scan architecture contains two balanced scan chains.

```
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 59](#), DFT Compiler overrides your request for two chains and generates three scan chains, one for each clock edge (clk1, positive-edge clk2, negative-edge clk2). Because the clock domains contain unequal numbers of cells, DFT Compiler generates unbalanced scan chains.

Figure 59 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.

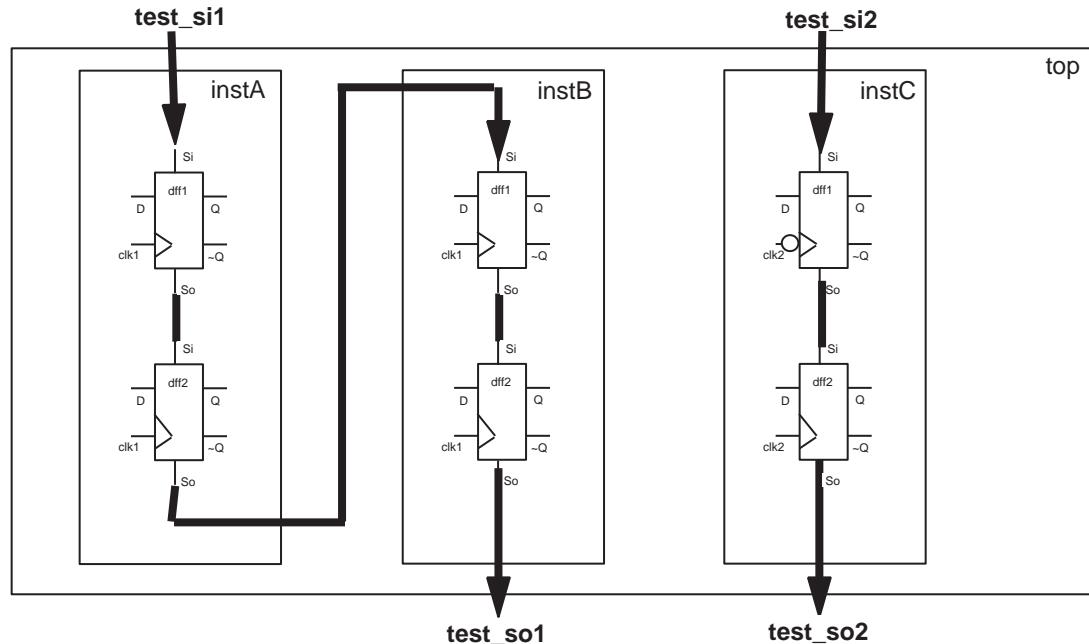
```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port {clk1 clk2}

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 60](#) shows the scan architecture when you allow edge mixing.

Figure 60 Better Balancing With Mixed Clock Edges



You can balance the scan chains by mixing clocks:

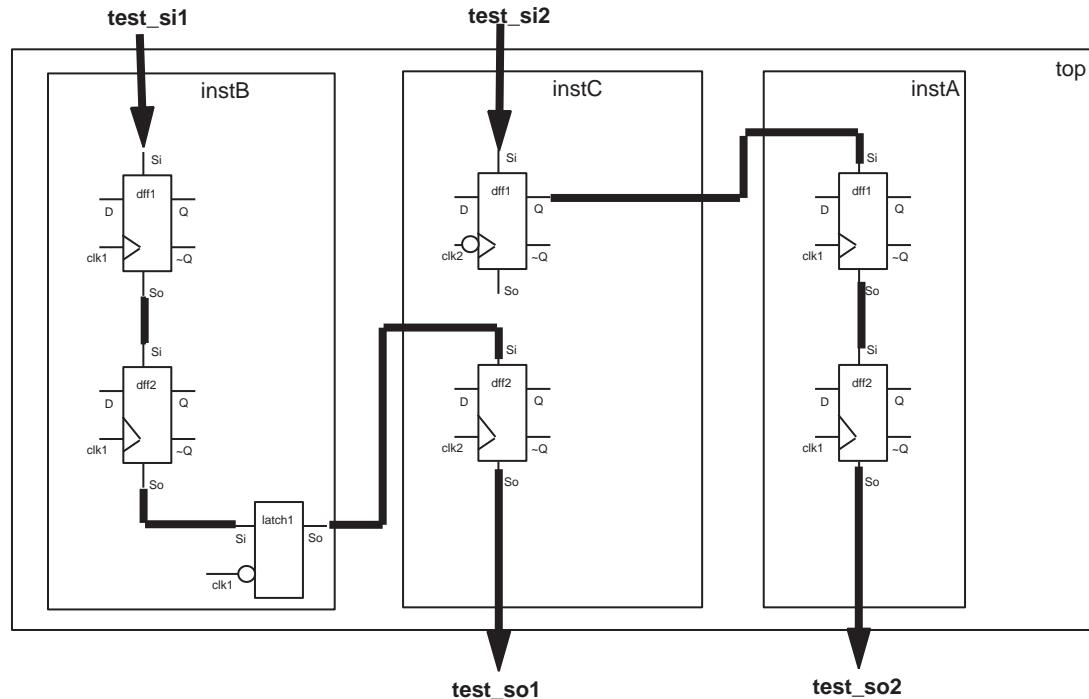
```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port {clk1 clk2}

dc_shell> set_scan_configuration -chain_count 2

dc_shell> set_scan_configuration -clock_mixing mix_clocks
```

Directly mixing clock edges in a single scan chain could produce a large timing risk. To reduce this risk, DFT Compiler adds lock-up latches to the scan path wherever clock skew issues might occur. [Figure 61](#) shows the resulting scan architecture.

Figure 61 Balanced Scan Chains With Mixed Clocks



See Also

- [Architecting Test Clocks on page 244](#) for details about scan lock-up latches and other clock-mixing considerations

Multibit Components and Scan Chains

A multibit component is a group of cells with identical functionality, inferred from RTL code or created by the `create_multibit` command. Depending on tool configuration and design constraints, synthesis implements a multibit component using multibit cells or single-bit cells.

By default, the `insert_dft` command treats the cells inside a multibit component as discrete sequential cells that can be reordered, split up, and rebalanced across scan chains as needed.

If you want to treat each sequential multibit component as a scan segment to retain cell grouping and order, use the following command:

```
dc_shell> set_scan_configuration -preserve_multibit_segment true
```

In this case, the cells inside a multibit component cannot be separated for length-balancing purposes. You can report the multibit scan segments that will be used by using the `preview_dft -show {segments}` command.

The `-preserve_multibit_segment` option applies globally to the current design and overrides any specification on subdesigns.

Physical Reordering and Repartitioning

In Design Compiler in topographical mode and in Design Compiler Graphical, the tool orders scan cells by physical proximity to minimize wire length. To use this feature, you must use both the `-spg` and `-scan` options in the initial *and* incremental `compile_ultra` commands. See [Example 15](#).

Example 15 Performing Physical Reordering and Repartitioning of Scan Chains

```
compile_ultra -scan ;# initial compile
# ...apply DFT configuration settings...
preview_dft
insert_dft
compile_ultra -scan -incremental ;# incremental post-DFT compile
```

Note:

This feature requires a license corresponding to the type of scan inserted in the design—a DFT Compiler license for standard scan designs and a DFTMAX or TestMAX DFT license for compressed scan designs.

After DFT insertion, when you incrementally optimize the design, the tool reorders and repartitions the scan elements as needed to further reduce congestion. In this case, the `compile_ultra` command issues the following message:

Information: Performing scan chain reordering in the SPG flow. (SPG-126)

Repartitioning uses the `multi_directional` repartitioning method by default. You can use the `set_optimize_dft_options` command to configure scan repartitioning. For example,

```
dc_shell> set_optimize_dft_options \
           -repartitioning_method single_directional
```

To perform reordering but not repartitioning, use the following command:

```
dc_shell> set_optimize_dft_options -repartitioning_method none
```

To disable both reordering and repartitioning, set the following variable:

```
dc_shell> set_app_var \
           test_enable_scan_reordering_in_compile_incremental false
```

Reordering and repartitioning follow the same rules used to generate the SCANDEF information that drives reordering and repartitioning in layout.

See Also

- [Introduction to SCANDEF on page 635](#) for more information about reordering and repartitioning

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 error 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 edgeup latch.

If your multiplexed flip-flop design violates this requirement, DFT Compiler reorders the invalid mixed-clock scan chains and triggers the following warning 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 warning 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 warning 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.

Retiming Scan-Ins and Scan-Outs to the Leading Clock Edge

In some cases, hierarchical blocks can contain scan chains with a mix of positive edge-triggered and negative edge-triggered flip-flops. When these block-level scan chains are combined at a higher level in the design hierarchy to form longer scan chains, half-cycle paths across clock edges might be created between the scan chains. Meeting timing for these half-cycle scan chain paths can be challenging at higher frequencies, especially for chips with long top-level routes between blocks.

To avoid these half-cycle paths when block-level scan chains are combined, use the `-add_test_retimings_flops` option of the `set_scan_configuration` command. For example,

```
dc_shell> set_scan_configuration -add_test_retimings_flops begin_and_end
```

When this option is specified for block-level scan chain insertion, flip-flops triggering on the leading edge of the test clock are added as needed to any scan chains that begin or end with trailing-edge-triggered flip-flops. For return-to-zero clocks, rising-edge flip-flops are used to retime to the leading edge. For return-to-one clocks, falling-edge flip-flops are used to retime to the leading edge. The tool automatically chooses the edge-triggered retiming cell from the target library.

Valid keywords for the `-add_test_retimings_flops` option are `begin_and_end`, `begin_only`, `end_only`, and `none`. The default is `none`, which disables retiming register insertion.

[Table 25](#) shows the retiming behaviors provided by the `-add_test_retimings_flops` option.

Table 25 Retiming Behaviors Provided by the -add_test_retimings_flops Option

If the scan chain	<code>begin_only</code>	<code>end_only</code>	<code>begin_and_end</code>
Begins with a leading-edge flip-flop			
Begins with a trailing-edge flip-flop	Adds retiming flip-flop to beginning of scan chain		Adds retiming flip-flop to beginning of scan chain
Ends with a leading-edge flip-flop			
Ends with a trailing-edge flip-flop		Adds retiming flip-flop to end of scan chain	Adds retiming flip-flop to end of scan chain

To report the locations where these retiming flip-flops are to be added, use the `preview_dft` command. For details on this command, see [Previewing the DFT Logic on page 603](#).

In addition, when this feature is enabled (with any value other than `none`), the following are clocked on the leading edge:

- Head pipeline registers
- DFT-inserted clock chains
- Shift-power control (SPC) chains
- DFTMAX Ultra decompressor registers
- DFTMAX serializer decompressor registers

If you generate a SCANDEF file for a design with retiming flip-flops, the retiming flip-flops are not included between the START and STOP points in the SCANDEF file. Only the original design scan cells are included between the START and STOP points.

This feature controls only one aspect of retiming flip-flop insertion. The tool can also insert retiming flip-flops as described in [Codec I/O Sharing and Standard Scan Chains on page 803](#).

See Also

- [Mixed Edges on page 92](#) for more information about the scan chain timing requirements of mixed edges

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 logic 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 logic 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 logic 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 logic 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 following 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.

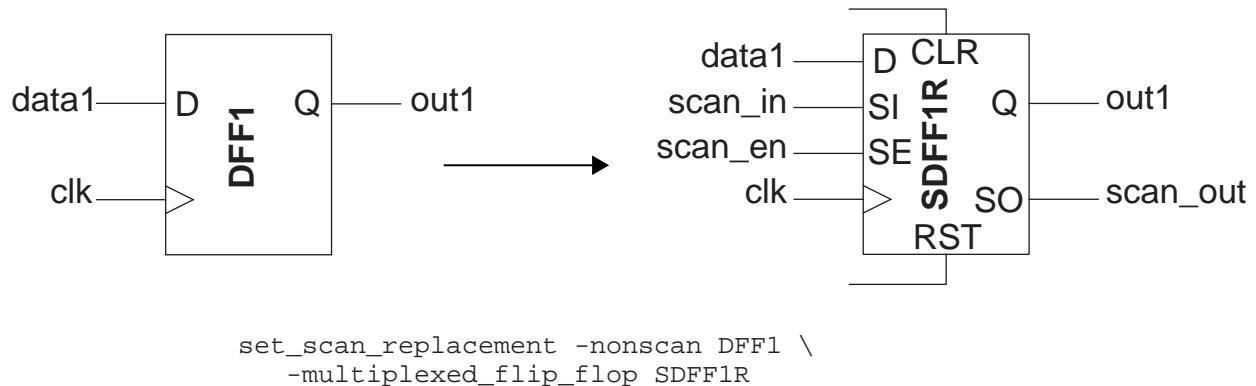
For example, the scan cell DFFS1 is a direct mapping of the nonscan cell DFFD1, but with scan pins. To specify the mapping of the DFFD1 nonscan cell to the DFFS1 scan cell, use the following command:

```
dc_shell> set_scan_replacement -nonscan DFFD1 \
    -multiplexed_flip_flop DFFS1
```

Few-Pins-to-Many-Pins Scan Cell Replacement Situation

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 62 Few-to-Many Scenario (Accepted)



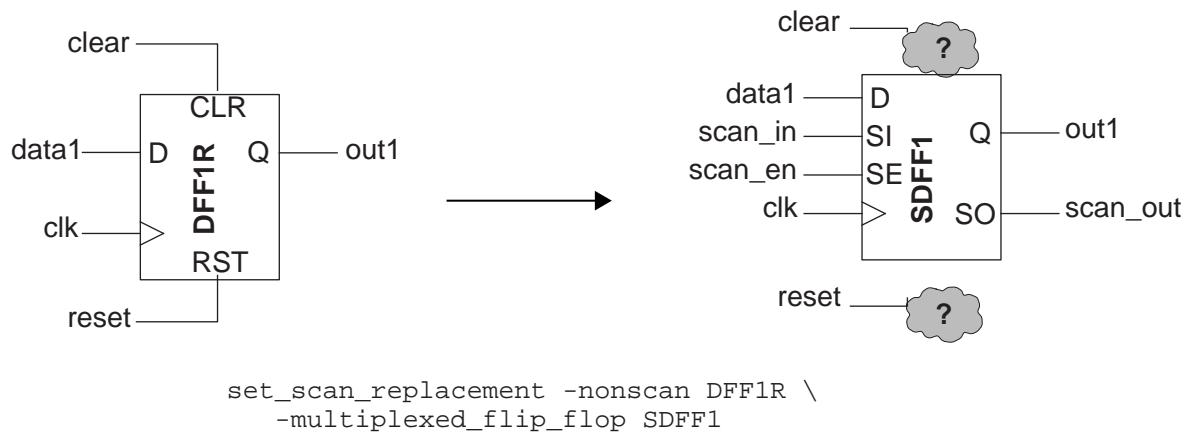
In [Figure 62](#), the replacement cell has more inputs and outputs than required by the nonscan cell. The unused pins of the scan cell are left unconnected.

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 incorrect 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 63](#), for example, the clear and reset pins do not exist on the scan cell. They are left unconnected, causing incorrect logic.

Figure 63 Many-to-Few Scenario (Rejected)



Criteria for Conversion Between Nonscan and Scan Cells

This topic 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_elements` to prevent flip-flops from being stitched into the scan chains. The difference between using `set_scan_configuration -exclude_elements` 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`

Scan Stitching Only Scan-Replaced Cells

By default, the `insert_dft` command performs scan replacement for all cells that are not scan replaced, but have scan equivalents and do not violate DRC.

If you have a design that is already scan-replaced and you do not want the `insert_dft` command to perform scan replacement of nonscan cells, specify the following command before running the `insert_dft` command:

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

With this setting, DRC evaluates only scan-replaced cells for inclusion in scan chains; nonscan cells are left as-is.

If you read in a .ddc file for a test-ready design, you do not need to specify the `set_scan_configuration -replace false` command. The design database contains the test attributes needed for the tool to recognize the scan-replacement results.

Using Existing Subdesign Scan Chains

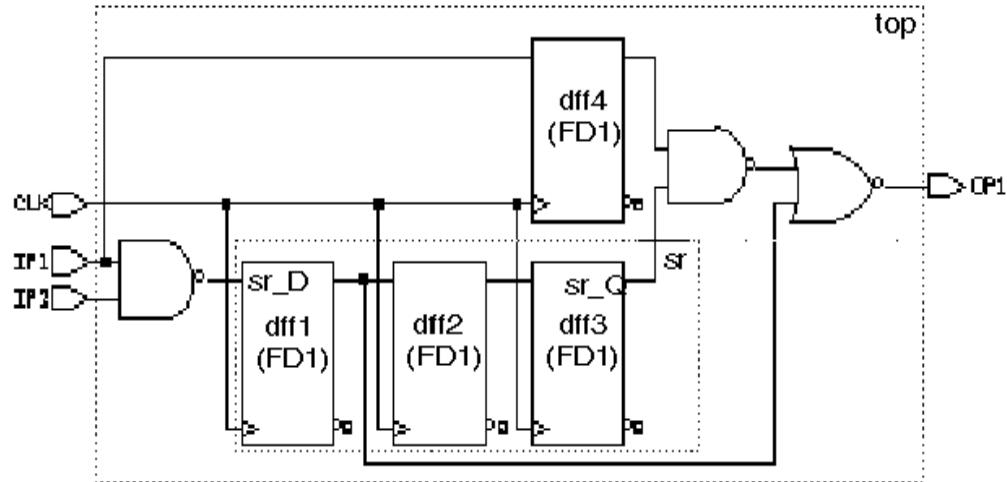
A subdesign scan chain uses subdesign ports for all test signals. DFT Compiler can infer subdesign scan chains during test design rule checking.

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 64](#) 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 64 Subdesign Scan Chain Example Before Scan Insertion



Use the following command sequence to infer the subdesign scan chain in module sr:

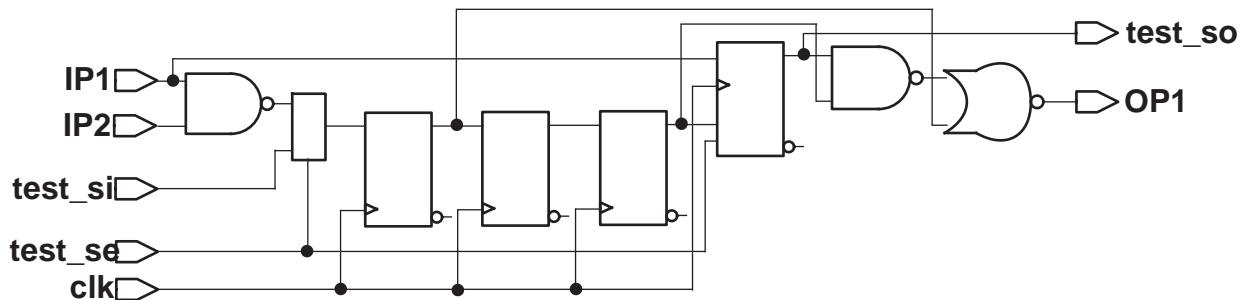
```
dc_shell> current_design sr
dc_shell> set_dft_signal -view spec -port sr_D -type ScanDataIn
dc_shell> set_dft_signal -view spec -port sr_Q -type ScanDataOut
dc_shell> create_test_protocol
dc_shell> dft_drc
```

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 65](#) 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 65 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:

```
dc_shell> set_app_var insert_test_design_naming_style name
```

In the previous example, the default name is `design_name_test_counter`.

Note:

To prevent unqualification 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 16](#) shows the type of information displayed by the `report_scan_path -chain all` command.

Example 16 Scan Path Information Displayed by the report_scan_path Command

Scan_path	Len	ScanDataIn	ScanDataOut	ScanEnable	MasterClock	SlaveClock
I_1	22	test_si1	test_so1	test_se	CLK	-
I_2	21	test_si2	test_so2	test_se	CLK	-
I_3	21	test_si3	test_so3	test_se	CLK	-

For more information, see man page for the `report_scan_path` command.

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 the test ports. If you are using your own placeholder test ports, you must set these attributes with the `set_dft_signal` command. If the `insert_dft` command creates any needed ports, 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
- Controlling Scan-Enable Connections to DFT Logic
- Controlling Buffering for DFT Signals
- Suppressing Replacement of Sequential Cells
- Changing the Scan State of a Design
- Removing Scan Configurations
- Keeping Specifications Consistent
- Synthesizing Three-State Disabling Logic
- Configuring Three-State Buses
- Handling Bidirectional Ports
- 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 26](#) provides a list of `signal_type` attribute values.

Table 26 signal_type Attribute Values for Test Signals

Test I/O port signal	signal_type value	Valid on input	Valid on output	Valid on three-state output	Valid on bidirectional input/output
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	4

4. Not recommended; complex methodologies required

Test I/O port signal	signal_type value	Valid on input	Valid on output	Valid on three-state output	Valid on bidirectional input/output
Asynchronous control ports	Reset	Yes	No	No	Yes

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:

```
Accepted dft signal specification for modes: all_dft
```

In the preceding example, the `-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 27](#).

Table 27 Port Naming Style Variables

Name	Default
<code>test_scan_in_port_naming_style</code>	<code>test_si%s%</code>
<code>test_scan_out_port_naming_style</code>	<code>test_so%s%</code>
<code>test_scan_enable_port_naming_style</code>	<code>test_se%</code>
<code>test_scan_enable_inverted_port_naming_style</code>	<code>test_sei%</code>
<code>test_clock_port_naming_style</code>	<code>test_c%</code>
<code>test_scan_clock_port_naming_style</code>	<code>test_sc%</code>
<code>test_scan_clock_a_port_naming_style</code>	<code>test_sca%</code>
<code>test_scan_clock_b_port_naming_style</code>	<code>test_scb%</code>
<code>test_mode</code>	<code>test_mode%</code>
<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 spec` and `-view existing_dft` 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.

To construct this dual modality, DFT Compiler needs to know what already exists in the design. You use the `-view existing_dft` option with the `set_dft_signal` command to provide such information. The tool then uses this information to perform pre-insertion design rule checking (DRC) to determine the elements that can be incorporated into scan chains.

Typical examples that use the `-view existing_dft` option include

- Clock signals:

```
set_dft_signal -view existing_dft -type ScanClock -port \
    clk -timing {45 55}
```

- Asynchronous set and reset signals:

```
set_dft_signal -view existing_dft -type Reset -port rst \
    -active_state 0
```

By default, DFT Compiler creates new ports in the design if they are needed. You can specify which existing ports the tool uses to build the DFT structures by using the `-view spec` option with the `set_dft_signal` command.

Typically, the `-view spec` option is 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_dft` option.
- If the information is needed to build DFT structures, then it should be specified by using the `-view spec` option.

Allocating Scan Ports

Ports that are defined to be scan-in and scan-out data ports are used in the order specified by the commands. For example, suppose that you identify three scan-in data ports and three scan-out data ports as follows:

```
set_dft_signal -type ScanDataIn -port [list SIN1 SIN2 SIN3]
set_dft_signal -type ScanDataOut -port [list SOUT1 SOUT2 SOUT3]
```

DFT Compiler allocates the listed ports to scan-in and scan-out functions as follows:

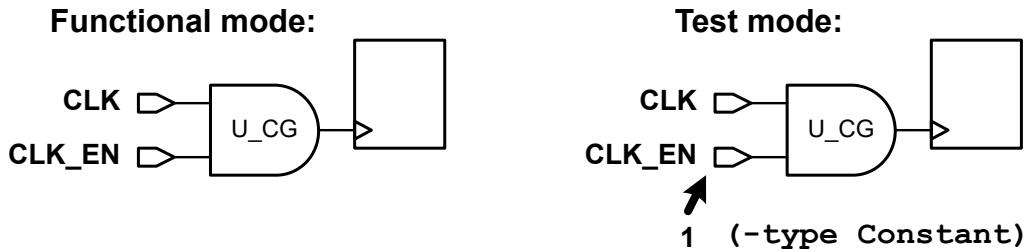
- If you specify two standard scan chains, the tool uses only the first two of the listed scan-in ports, SIN1 and SIN2, and only the first two listed scan-out ports, SOUT1 and SOUT2. SIN3 and SOUT3 are not used for scan chain connection purposes.
- If you specify three standard scan chains, the tool uses all of the listed scan-in and scan-out ports: SIN1, SIN2, SIN3, SOUT1, SOUT2, and SOUT3.
- If you specify four standard scan chains, the tool first uses the three designated scan-in and three designated scan-out ports. For the fourth chain, it creates an additional dedicated scan-in port and an additional scan-out port. The scan-out port can be an existing output port connected to the output of a flip-flop, which can be reused as a scan-out port, or it can be a new dedicated output port.

Using -type Constant versus Using -type TestMode

The difference between Constant and TestMode signal definitions is as follows:

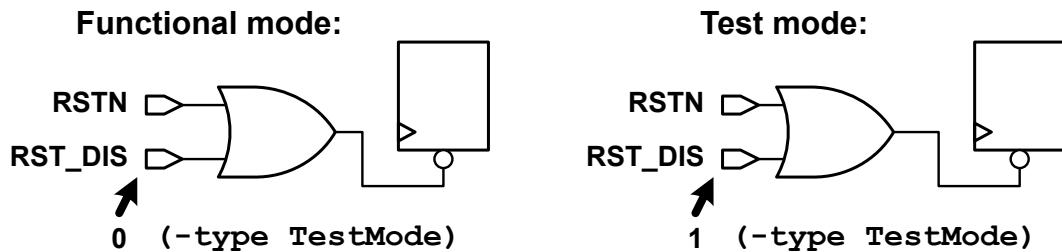
- Constant signal definitions specify the value of the signal in test mode, but they do not specify anything about the signal's value in functional mode. [Figure 66](#) shows the behavior of a Constant signal in the functional and test modes.

Figure 66 Constant Signal Specification



- TestMode signal definitions specify the value of the signal in both test mode and functional mode. [Figure 67](#) shows the behavior of a Constant signal in the functional and test modes.

Figure 67 Design With a Controlled Clock Signal



This difference affects the generation of verification setup files, which are specified with the `set_svf` command. Verification setup files contain information about the functional mode values for any defined DFT signal. This allows Formality equivalence checking to disable the test logic added by DFT Compiler when the `synopsys_auto_setup` variable is set to true.

This difference affects verification setup file generation for the preceding examples as follows:

- The Constant signal definition does not result in any verification setup file directives because it contains no information about the functional mode value of CLK_EN.
- The TestMode signal definition specifies that the functional mode value for the RST_DIS signal is 1. This results in the following verification setup file directive:

```
guide_scan_input \
    -design { top } \
    -disable_value 0 \
    -ports { RST_DIS }
```

Selecting Test Ports

By default, DFT Compiler creates dedicated test ports as needed, but it also 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.

You can also share ports between test and normal operation, which minimizes the number of dedicated test ports required for internal scan. 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.

The following topics describe how to select and define existing ports in your design as test ports:

- [Defining Existing Unconnected Ports as Scan Ports](#)
- [Sharing a Scan Input With a Functional Port](#)
- [Sharing a Scan Output With a Functional Port](#)
- [Controlling Subdesign Scan Output Ports](#)

Defining Existing Unconnected Ports as Scan Ports

You can define existing unconnected ports in your RTL description for use as test ports. These are known as placeholder scan ports or dummy scan ports. This approach allows you to use the same testbench for the RTL and gate-level implementations of your design.

Use the `set_dft_signal` command to instruct DFT Compiler to use these ports:

```
dc_shell> set_dft_signal -type ScanDataIn -view spec \
    -port SI1

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

```
dc_shell> set_dft_signal -type ScanDataOut -view spec \
    -port SO
```

Sharing a Scan Input With a Functional Port

By default, DFT Compiler always creates a dedicated scan input port. To share a scan input port with a specified existing functional port, use the `set_dft_signal` command.

```
dc_shell> set_dft_signal -type ScanDataIn -view spec \
    -port DATA_in[0]
```

If you select a bidirectional port as the scan input port, DFT Compiler automatically inserts the necessary bidirectional control logic to enable the input path during scan shift.

Sharing a Scan Output With a Functional Port

By default, if a scannable cell directly drives an output port in the current design, DFT Compiler automatically uses it as the last cell in the scan chain. DFT Compiler disrupts the ordering to place this cell at the end of the scan chain. If multiple scannable sequential cells directly drive output ports, DFT Compiler uses the cell that would have been stitched closest to the end of the scan chain. If the scan cell is the last cell in a scan segment, the entire scan segment is placed at the end of the scan chain. Use the `preview_dft` command to see if a cell or segment has been moved to the end of the scan chain to prevent a dedicated scan output port.

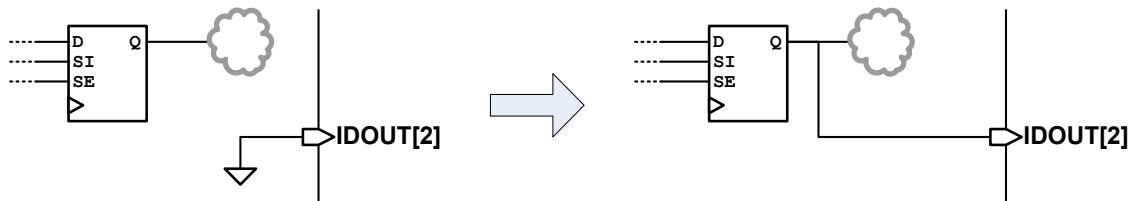
To select the functional port to be used as a scan output port, use the `set_dft_signal` command.

```
dc_shell> set_dft_signal -type ScanDataOut -view spec \
    -port DATA_out[0]
```

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 you select a bidirectional or three-state port as the scan output port, DFT Compiler automatically inserts the necessary control logic to enable the output path during scan shift.

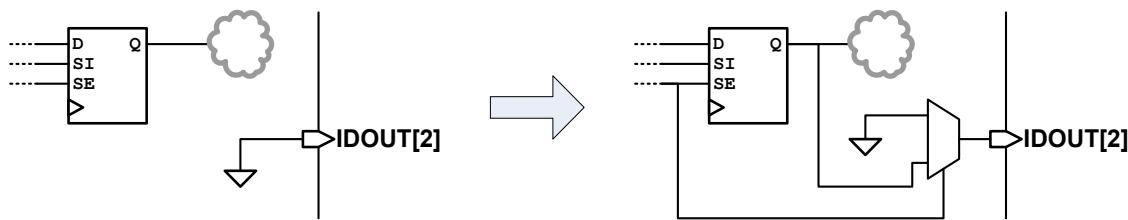
By default, if the specified port is tied to logic 0 or logic 1, DFT Compiler ignores the constant value during scan insertion and drives the port directly as a scan output, as shown in [Figure 68](#). This behavior ensures that no additional logic is added at the port if it was undriven in the RTL and tied to logic 0 during synthesis.

Figure 68 Scan Output Port With Constant MUXing Disabled



If the existing constant value driving the port is required for proper operation in functional mode, set the `test_mux_constant_so` variable to `true`. In this case, DFT Compiler multiplexes the scan-out signal with the constant value, using the scan-enable signal to control the multiplexer, as shown in Figure 69.

Figure 69 Scan Output Port With Constant MUXing Enabled



If your semiconductor vendor requires dedicated top-level scan output ports or you prefer them, use the `set_scan_configuration` command to always use dedicated scan outputs:

```
dc_shell> set_scan_configuration \
           -create_dedicated_scan_out_ports true
```

Controlling Subdesign Scan Output Ports

By default, when DFT Compiler routes scan chains through subdesigns, it uses existing subdesign output ports driven by scan cells wherever possible. This minimizes the number of new output ports added to subdesigns, and it can reduce the amount of design unification required for multiply-instantiated designs. However, it can also add the loading of external scan routes outside the subdesign to functional nets inside the subdesign.

To always create dedicated scan-out ports on subdesigns, set the following variable:

```
dc_shell> set_app_var test_dedicated_subdesign_scan_outs true
```

Note that this variable setting alone does not guarantee isolation of the functional path from external scan path loading. You must also apply the `set_fix_multiple_port_nets` command to subdesigns where the functional subdesign outputs should be isolated from external scan path loading. For more information about this command, see the man page.

Controlling Scan-Enable Connections to DFT Logic

By default, DFT Compiler uses a global scan-enable signal for DFT logic connections. In some cases, you might want to create multiple scan-enable signals and control how they connect to the DFT logic.

The following topics describe how to control scan-enable connections to DFT logic:

- [Associating Scan-Enable Ports With Specific Scan Chains](#)
- [Defining Dedicated Scan-Enable Signals for Scan Cells](#)
- [Connecting the Scan-Enable Signal in Hierarchical Flows](#)
- [Preserving Existing Scan-Enable Pin Connections](#)

Associating Scan-Enable Ports With Specific Scan Chains

To associate a specific port with specific scan chains, use the `set_dft_signal` and `set_scan_path` commands, as follows:

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

dc_shell> set_scan_path {chain_names} -view spec \
                    -scan_enable port_name
```

If the condition set with these commands cannot be met, a warning is issued during scan preview and scan insertion.

Defining Dedicated Scan-Enable Signals for Scan Cells

By default, DFT Compiler chooses an available ScanEnable signal to connect to the scan-enable pins of scan cells. However, you can also define a dedicated ScanEnable signal to use for these scan-enable pin connections.

Specifying a Global Scan-Enable Signal

You can define a global ScanEnable signal to use for the scan-enable pins of scan cells by using the `-usage scan` option when defining the signal with the `set_dft_signal` command:

```
set_dft_signal
  -type ScanEnable
  -view spec
  -usage scan
  -port port_list
```

To define a signal with the `scan usage`, the `-view` option must be set to `spec`.

When you define a ScanEnable signal with the `scan` usage, the `insert_dft` command is limited to using only that signal to connect to the scan-enable pins of scan cells. If there are insufficient ScanEnable signals for other purposes, DFT Compiler creates additional ScanEnable signals as needed.

You can use the `report_dft_signal` and `remove_dft_signal` commands for reporting and removing the specification, respectively.

Specifying Object-Specific Scan-Enable Signals

You can also define dedicated ScanEnable signals for specific parts of the design by using the `-connect_to` option and associated options of the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable
    -view spec
    -usage scan
    -port port_list
    [-connect_to object_list]
    [-connect_to_domain_rise clock_list]
    [-connect_to_domain_fall clock_list]
    [-exclude object_list]
```

The `-connect_to` option specifies a list of design objects that are to use the specified ScanEnable signal. The supported object types are

- Scan cells
- Hierarchical cells
- Designs
- Test clock ports

This allows you to make clock-domain-based signal connections. It includes scan cells clocked by the specified test clocks. The functional clock behavior is not considered.

- Scan-enable pins of CTL-modeled cores

The `-connect_to_domain_rise` and `-connect_to_domain_fall` options accept a test clock port list and work the same as the `-connect_to` option, except that they apply only to rising-edge and falling-edge scan cells, respectively.

You can also use the `-exclude` option to specify a list of scan cells, hierarchical cells, or design names to exclude from the object-specific control signal.

The following example defines two ScanEnable signals, named `SE_CLK1` and `SE_CLK2`, to connect to the scan-enable pins of test clock domains `CLK1` and `CLK2`, respectively:

```
dc_shell> set_dft_signal -type ScanClock -view existing_dft \
    -port {CLK1 CLK2} -timing {45 55}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
```

```
-port SE_CLK1 -connect_to {CLK1}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
           -port SE_CLK2 -connect_to {CLK2}
```

Note the following limitation:

- You cannot specify object-specific scan-enable specifications for pipelined scan-enable signals.

Connecting the Scan-Enable Signal in Hierarchical Flows

When you insert DFT at a top level that contains cores, which are DFT-inserted blocks represented by CTL models, the cores already contain complete scan-enable networks. Instead of connecting the top-level ScanEnable signal to target pins inside the core, DFT Compiler must connect to ScanEnable signal pins at the core boundary.

When ScanEnable signals at the core and/or top level are defined with the `-usage` option of the `set_dft_signal` command, DFT Compiler attempts to determine which top-level signal should drive each core-level signal, using the priorities shown in [Table 22 on page 133](#).

You can override the default connection behaviors for cores by using object-specific signal definitions at the top level, applied using the `set_dft_signal -connect_to` command and associated options:

```
set_dft_signal
  -type ScanEnable
  -view spec
  -usage scan | clock_gating
  -port port_list
  [-connect_to object_list]
  [-connect_to_domain_rise clock_list]
  [-connect_to_domain_fall clock_list]
  [-exclude object_list]
```

Object-specific specifications are described in [Specifying Object-Specific Scan-Enable Signals on page 231](#). However, not all object types accepted by the `object_list` argument apply to cores. The object types that apply to cores are

- Test clock ports
- This allows you to make clock-domain-based signal connections to cores. It includes core-level scan-enable pins associated with the specified test clocks. The functional clock behavior is not considered.
- Scan-enable pins of CTL-modeled cores

This allows you to make direct pin-to-pin connections from top-level signal sources to core-level pins.

Specifying Domain-Based Connections to Core Scan-Enable Pins

When you specify a top-level domain-based signal connection, DFT Compiler uses information inside a core's CTL model to determine the core scan-enable pins associated with each core clock. To ensure that this information is present in the model, the following requirements must be observed during core creation:

- Core-level ScanEnable signals must be defined using the `-usage` option of the `set_dft_signal` command. This ensures that the core's internal scan-enable signal is not used outside its intended usage.
- Core-level ScanEnable signals defined with a usage of `clock_gating` must also be defined as domain-specific signals using the `-connect_to` `clock_list` option of the `set_dft_signal` command. This ensures that clock-specific clock-gating annotations are included in the CTL model.

[Example 17](#) shows part of a core-level ASCII CTL model that contains clock domain information for a scan-enable signal defined with a usage of `scan`.

Example 17 Scan Chain Clock Information in an ASCII CTL Model

```
CTL all_dft {
    ...
    Internal {
        "SE_SCAN" {
            CaptureClock "CLK" {
                LeadingEdge;
            }
            DataType User "ScanEnableForScan" {
                ActiveState ForceUp;
            }
        }
    }
}
```

[Example 18](#) shows part of a core-level ASCII CTL model that contains clock domain information for a scan-enable signal defined with a usage of `clock_gating`. The domain-based signal specification causes the `CaptureClock` constructs to be included.

Example 18 Clock-Gating Clock Information in an ASCII CTL Model

```
CTL all_dft {
    ...
    Internal {
        "SE(CG)" {
            CaptureClock "CLK" {
                LeadingEdge;
            }
            DataType User "ScanEnableForClockGating" {
                ActiveState ForceUp;
            }
        }
    }
}
```

```

        }
    }
}
```

The CTL model information for a signal can contain multiple DataType constructs (for signals defined with multiple usages) and multiple CaptureClock constructs (for signals associated with multiple clocks).

Consider a core with two scan-enable pins, where pin SE1 is associated with CLK1 and pin SE2 is associated with CLK2. The following top-level commands connect corresponding top-level scan-enable signals TOP_SE1 and TOP_SE2 to these core-level pins indirectly using domain-based specifications:

```

dc_shell> set_dft_signal -type ScanClock -view existing_dft \
    -port {CLK1 CLK2} -timing {45 55}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
    -port TOP_SE1 -connect_to {CLK1}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
    -port TOP_SE2 -connect_to {CLK2}
```

Specifying Connections Directly to Core Scan-Enable Pins

You can specify pin-based signal connection specifications that connect any top-level ScanEnable signal to any core-level ScanEnable pin. These pin-based connection specifications override the default connection behavior, and there is no requirement for the top-level and core-level signal usages to match.

Consider a core with two scan-enable pins, where pin SE1 is associated with CLK1 and pin SE2 is associated with CLK2. The following top-level commands connect top-level scan-enable signals to corresponding core-level pins using direct core pin specifications:

```

dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
    -port TOP_SE1 -connect_to {CORE/SE_SCAN1}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
    -port TOP_SE2 -connect_to {CORE/SE_SCAN2}
```

Preserving Existing Scan-Enable Pin Connections

During DFT insertion, DFT Compiler identifies the scan-enable pins of scan cells and scan cores that should be connected to the global scan-enable signal. These are known as scan-enable target pins.

By default, if a scan-enable target pin already has a connection, DFT Compiler disconnects it to make the connection to a scan-enable signal. During DFT insertion, the `insert_dft` command issues a TEST-394 warning to note the disconnection:

Warning: Disconnecting pin 'memwrap/UMEM/SE' to route scan enable.
 (TEST-394)

To preserve existing connections to scan-enable target pins during DFT insertion, set the following variable:

```
dc_shell> set test_keep_connected_scan_en true
```

In this case, the `insert_dft` command issues a TEST-410 warning to confirm that the existing connection is kept:

```
Warning: Not disconnecting pin 'memwrap/UMEM/SE' to route scan enable.  

(TEST-410)
```

For more information, see the man page. For an example application, see [SolvNet article 034774, “How To Connect DFT Signals to Hierarchical Pins of Verilog Wrappers.”](#)

Controlling Buffering for DFT Signals

To have synthesis buffer a DFT signal, use the `set_driving_cell` command to specify the source port's drive characteristics:

```
dc_shell> set_driving_cell -lib_cell BUFX4 test_scan_enable
```

To prevent synthesis from buffering a DFT signal, use the `set_ideal_network` command to configure the source port as the driver of an ideal network:

```
dc_shell> set_ideal_network test_scan_enable
```

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 the `insert_dft` command.

For full-scan designs, the `insert_dft` command 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 Logic Scan Synthesis

In logic 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 the `set_scan_element false` command 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 19](#).

Example 19 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

Caution:

You do not need to set the scan state if you are following the recommended design flow.

Removing Scan Configurations

The `reset_scan_configuration` 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 the `remove_scan_path` command to remove the path specifications you defined with the `set_scan_path` command.

Note that the `reset_scan_configuration` command does not change your design. It merely deletes specifications you have made.

You can use the `reset_scan_configuration` command to remove explicit specifications of synthesizable segments. When you remove an explicit specification, the multibit component inherits the current implicit specification.

Note:

The `reset_scan_configuration` command does not affect the settings made with the `set_scan_register_type` command. These settings must be removed with 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 70](#) shows the simple internal three-state net used as an example throughout this section.

Figure 70 Internal Three-State Net Example

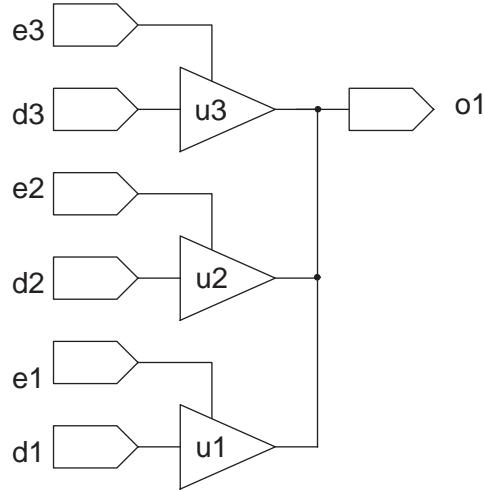
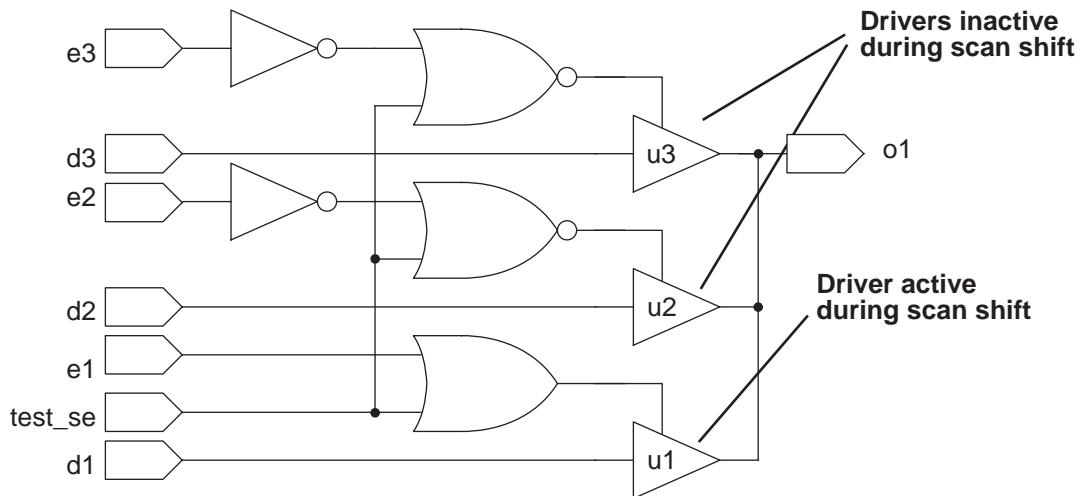


Figure 71 shows the disabling logic added by DFT Compiler during the `insert_dft` process.

Figure 71 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 prevent DFT Compiler from inserting the disabling logic:

```
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 the `insert_dft` command 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.

```
dc_shell> # subdesign command sequence
dc_shell> current_design subdesign
dc_shell> set_dft_configuration -fix_bus disable
...
dc_shell> insert_dft

dc_shell> # 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 the `set_dft_configuration` command 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 the `set_dft_configuration` command.

See Also

- [Previewing Additional Scan Chain Information on page 604](#) for more information about previewing tristate conditioning logic

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 `-method enable_one` option. If you want to ensure that all external three-state nets are disabled, use the `-method disable_all` option, which is the default behavior for the `external_tristates` type.

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_ autofix_configuration` command appropriately for each module before you run the `insert_dft` command 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 unquifying 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 on a specific list of objects, use the following command:

```
set_autofix_element \
    -type internal_bus | external_bus \
    -method input | output | no_disabling \
    object_list
```

- To set local bidirectional port specifications on a specific list of objects, use the following command:

```
set_autofix_element \
    -type bidirectional \
    -method input | output | no_disabling \
    object_list
```

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 the `preview_dft -show {bidirectionals}` command to see the bidirectional port conditioning that will be implemented for each bidirectional port in a design.

See Also

- [Previewing Additional Scan Chain Information on page 604](#) for more information about previewing bidirectional conditioning logic

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

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 defaults for the clock timing, based on the clock type, unless you explicitly specify clock timing.

This topic shows you how to set test clocks and handle multiple clock designs. It includes the following:

- [Defining Test Clocks](#)
- [Specifying a Hookup Pin for DFT-Inserted Clock Connections](#)
- [Requirements for Valid Scan Chain Ordering](#)
- [Lock-Up Latch Insertion Between Clock Domains](#)

- Automatically Creating Skew Subdomains Within Clock Domains
- Manually Creating Skew Subdomains at Associated Internal Pins
- Manually Creating Skew Subdomains With Scan Skew Groups
- Defining Scan Chains by Scan Clock
- Handling Multiple Clocks in LSSD Scan Styles

Defining Test Clocks

To explicitly define test clocks in your design, use the `set_dft_signal` command. For example,

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

Specify the clock signal type with the `-type` option. For the multiplexed flip-flop style, use the `ScanClock` type. For other scan styles, see the man page.

Define the test clock waveform with the `-timing` option. The waveform definition is a pair of values that specifies the rising-edge arrival time followed by the falling-edge arrival time. [Figure 72](#) shows a return-to-zero clock waveform definition.

Figure 72 Return-to-Zero Test Clock Waveform Definition

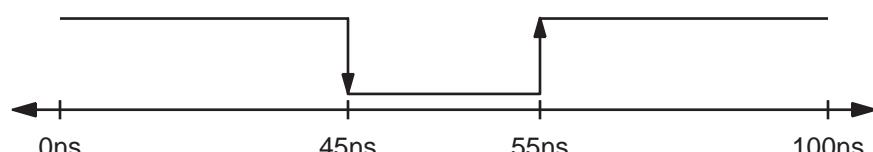
```
set_dft_signal ... -timing {45 55}
```



[Figure 73](#) shows a return-to-one clock waveform definition.

Figure 73 Return-to-One Test Clock Waveform Definition

```
set_dft_signal ... -timing {55 45}
```



If you use the `-infer_clock` option of the `create_test_protocol` command to infer test clocks in your design, the tool uses the default clock waveforms shown in [Table 27](#).

Scan clock type	First edge (ns)	Second edge (ns)
Edge-triggered (non-LSSD styles)	45.0	55.0
Master clock (LSSD styles)	30.0	40.0
Slave clock (LSSD styles)	60.0	70.0

For all test clocks, the clock period is the value defined by the `test_default_period` variable.

After defining or inferring your test clocks, you can verify their timing characteristics by using the `report_dft_signal` command.

The rise and fall clock waveform values 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 a Hookup Pin for DFT-Inserted Clock Connections

In some cases, DFT Compiler might need to make a connection to an existing scan clock network during DFT insertion. Some examples are

- Pipeline clock connections for automatically inserted pipelined scan data registers
- Test point clock connections for test points with flip-flops
- ATE clock connections for DFT-inserted OCC controllers
- Codec clock connections for serialized and streaming scan compression
- Self-test clock connections to the LogicBIST self-test controller and codec

By default, DFT Compiler makes the clock connection at the source port specified in the `-view existing_dft` signal definition. However, if you want DFT Compiler to make the clock connection at an internal pin, such as a pad cell or clock buffer output, you can specify it with the `-hookup_pin` option in a subsequent `-view spec` signal definition. For example,

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}
dc_shell> set_dft_signal -view spec -type ScanClock \
    -port CLK -hookup_pin UCLKBUF/Z
```

You do not specify the clock waveform timing for the `-view spec` signal definition, but you must specify the associated port with the `-port` option.

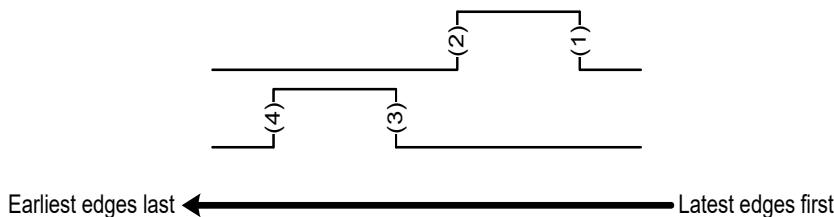
Requirements for Valid Scan Chain Ordering

This topic describes the requirements for valid scan chain ordering in the multiplexed flip-flop scan style.

DFT Compiler generates valid mixed-clock scan chains based on the ideal test clock timing. Scan chain cells are ordered by the ideal test clock edge times, as defined with the `-timing` option of the `set_dft_signal` command. Cells clocked by later clock edges are placed before cells clocked by earlier clock edges. This guarantees that all cells in the scan chain get the expected data during scan shift.

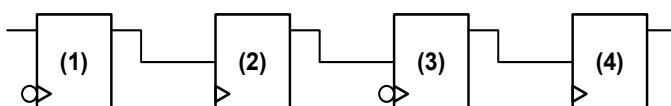
[Figure 74](#) shows the ideal test clock waveforms for two test clocks. The clock edges are numbered by their edge timing order, with the latest clock edge indicated by (1).

Figure 74 Ideal Test Clock Waveforms for Two Test Clocks



[Figure 75](#) shows how DFT Compiler constructs a scan chain containing a scan cell clocked by each clock edge. The scan cells are ordered with the cells clocked by the latest clock edges coming first.

Figure 75 Scan Chain Cells for Two Test Clocks



To maintain the validity of your scan chains, do not change the test clock timing after assembling the scan structures.

Although DFT Compiler chooses an order that ensures correct shift function under ideal clock timing, it cannot guarantee that capture problems will not occur. Capture problems are caused by your logic functionality; modify your design to correct capture problems. For more information, see [Chapter 13, Pre-DFT Test Design Rule Checking](#).

By default, when you request clock mixing within a multiplexed flip-flop scan chain, DFT Compiler inserts lock-up latches to prevent timing problems. For more information, see [Lock-Up Latch Insertion Between Clock Domains on page 248](#).

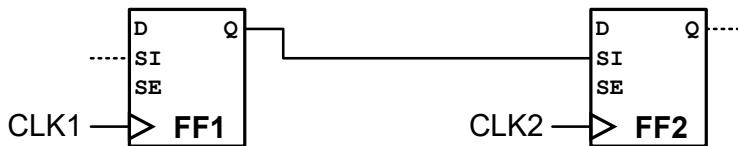
Lock-Up Latch Insertion Between Clock Domains

This topic describes how the tool adds lock-up latches between clock domains in the multiplexed flip-flop scan style.

A *scan lock-up latch* is a retiming sequential cell on a scan path that can address skew problems between adjacent scan cells when clock mixing or clock-edge mixing is enabled. DFT Compiler inserts them to prevent skew problems that might occur.

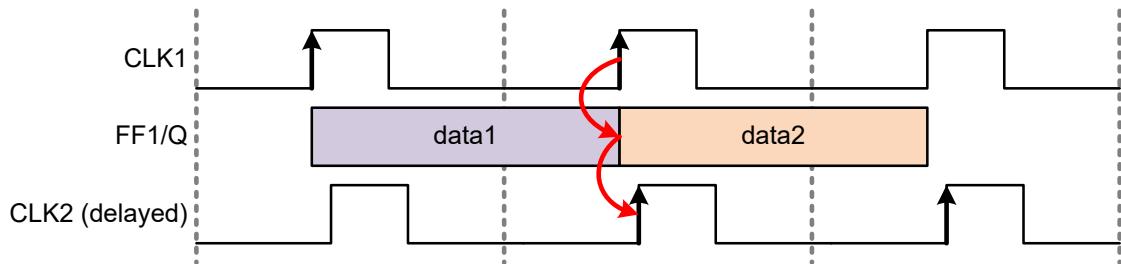
Consider the scan structure in [Figure 76](#), where a scan cell clocked by CLK1 feeds a scan cell clocked by CLK2, and both clocks are defined with the same ideal waveform definition.

Figure 76 Two Scan Cells Clocked by Two Different Clocks



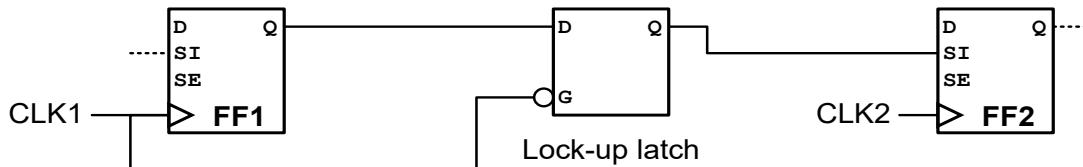
If both scan cells receive a clock edge at the same time, no timing violations occur. However, if the CLK2 waveform at FF2 is delayed, perhaps due to higher clock tree latency, a hold violation might result where FF2 incorrectly captures the current cycle's data instead of the previous cycle's data. [Figure 77](#) shows this hold violation for leading-edge scan cells.

Figure 77 Timing for Two Leading-Edge Scan Cells Clocked by Two Different Clocks



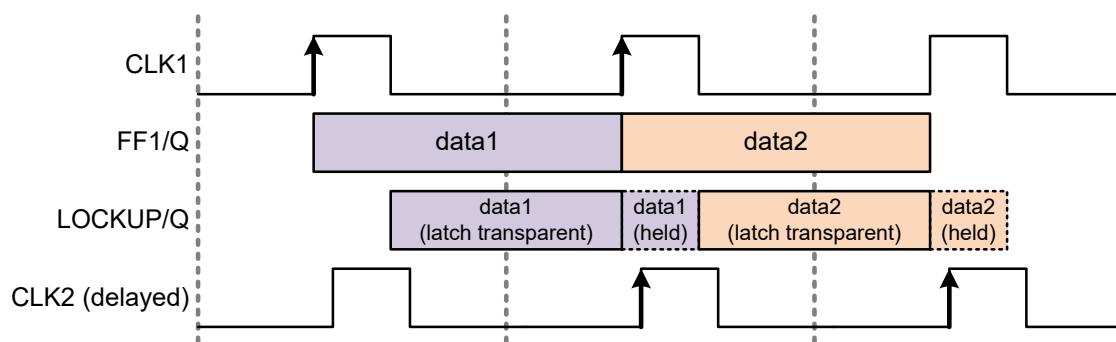
A lock-up latch prevents hold violations for scan cells that might capture data using a skewed clock edge. It is a latch cell that is inserted between two scan cells and clocked by the inversion of the previous scan cell's clock. [Figure 78](#) shows the same two scan cells with a lock-up latch added.

Figure 78 Two Scan Cells With a Lock-Up Latch



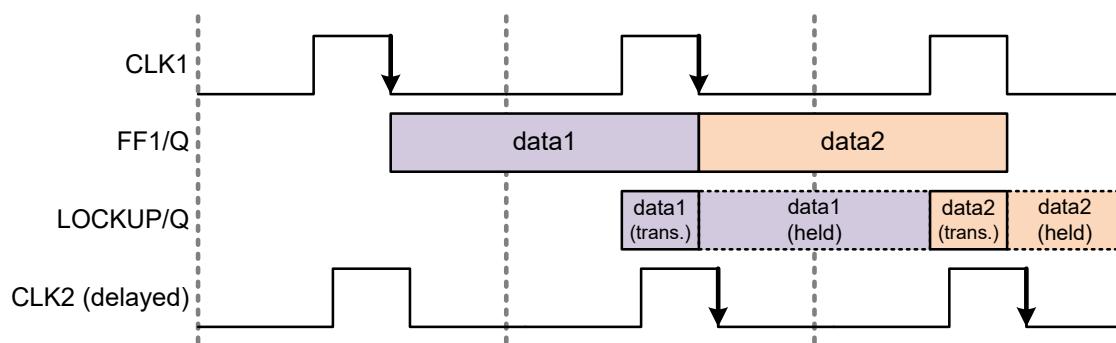
The lock-up latch cell works by holding the previous cycle's scan data while the current cycle's scan data is captured, effectively delaying the output data transition to the next edge of the source clock. [Figure 79](#) shows the lock-up timing behavior for the example. Although this example uses return-to-zero clock waveforms, lock-up latch operation is similar for return-to-one clock waveforms.

Figure 79 Timing for Two Leading-Edge Scan Cells With a Lock-Up Latch



Lock-up latch operation for trailing-edge scan cells is similar to that of leading-edge scan cells, except that the data is held into the next clock cycle as shown in [Figure 80](#).

Figure 80 Timing for Two Trailing-Edge Scan Cells With a Lock-Up Latch



By default, DFT Compiler adds scan lock-up latches as needed to multiplexed flip-flop scan chains. Scan chain cells are ordered by the ideal test clock edge times, as defined with the `-timing` option of the `set_dft_signal` command. Cells clocked by later clock

edges are placed before cells clocked by earlier clock edges. Adjacent scan chain cells clocked by different clock edges are handled as follows:

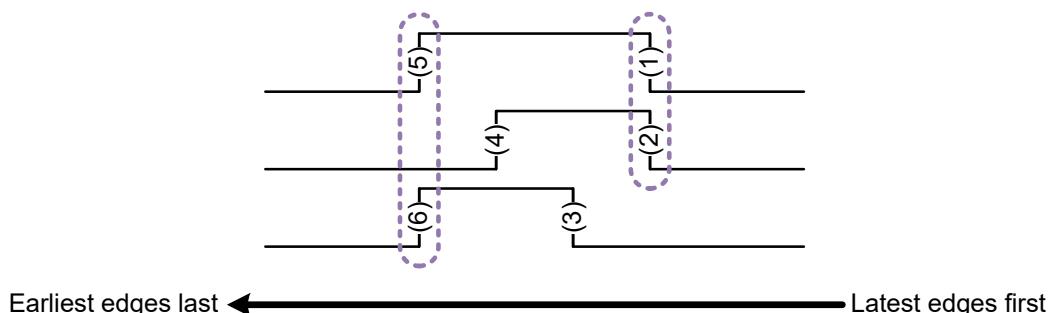
- When the scan cells are clocked by clock edges with different ideal clock edge timing, DFT Compiler does not insert a lock-up latch. The second scan cell captures data using an earlier clock edge, and DFT Compiler assumes this difference in the ideal clock edge timing is sufficient to avoid a hold time violation.
- When the scan cells are clocked by clock edges with identical ideal clock edge timing, DFT Compiler inserts a lock-up latch to avoid a potential hold violation due to clock skew.

Note:

DFT Compiler builds scan paths that meet zero-delay timing (without clock propagation delay or uncertainty). In [Figure 79](#), if CLK2 is skewed later than CLK1 by more than the active-high pulse width of CLK1, a hold violation can still occur.

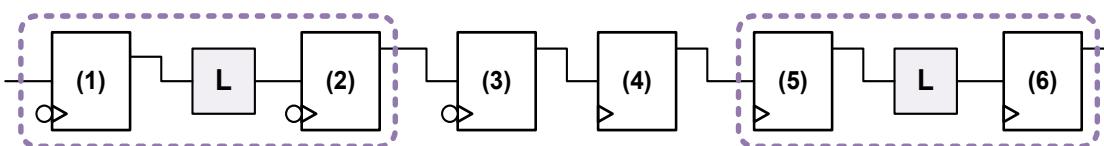
[Figure 81](#) shows a set of ideal test clock waveforms for a set of overlapping test clocks. The clock edges are numbered by their edge timing order, with the latest clock edge indicated by (1). Clock edges with identical ideal clock edge timing are highlighted.

Figure 81 Ideal Test Clock Waveforms for Overlapping Test Clocks



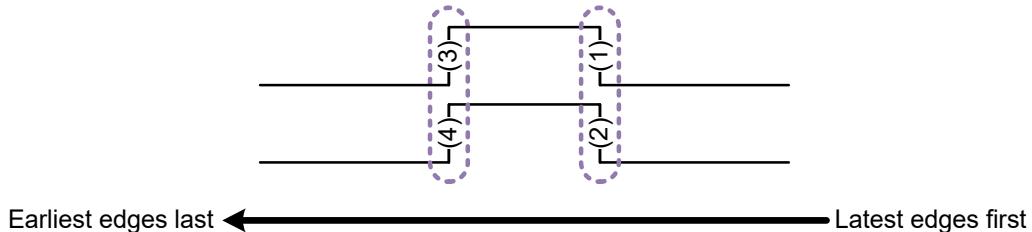
[Figure 82](#) shows how DFT Compiler constructs a scan chain containing a scan cell clocked by each clock edge. The scan cells are ordered with the cells clocked by the latest clock edges coming first. Lock-up latches are inserted between scan cells clocked by the clock edges with identical ideal clock edge timing.

Figure 82 Scan Chain Lock-Up Latches for Overlapping Test Clocks



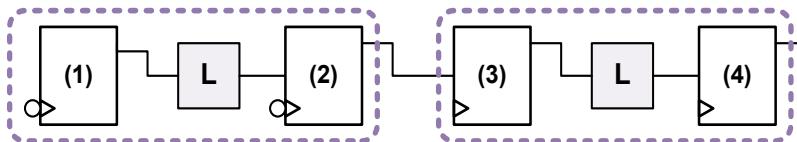
However, in most designs, all test clocks share identical return-to-zero test clock waveforms, as shown in [Figure 83](#).

Figure 83 Ideal Test Clock Waveforms for Simple Test Clocks



In this case, the ordering behavior is simplified. The scan cells are ordered with all falling-edge scan cells first and all rising-edge scan cells last, as shown in [Figure 84](#). Lock-up latches are inserted between differently-clocked scan cells within the rising-edge and falling-edge sections of the scan chain.

Figure 84 Scan Chain Lock-Up Latches for Simple Test Clocks



DFT Compiler inserts a lock-up latch at the same level of hierarchy as the scan output pin of the preceding scan element:

- If the preceding element is a CTL model or is located in a block containing CTL model information, the lock-up latch is inserted at the level of hierarchy where the CTL model exists.
- If the preceding element is a leaf scan cell (that does not exist in a CTL-modeled block), the lock-up latch is inserted at the level of hierarchy where the scan cell exists.

The `set_scan_configuration` command provides options to control lock-up latch insertion. By default, DFT Compiler performs automatic lock-up latch insertion for multiplexed flip-flop scan chains. To disable this feature, use the `-add_lockup` option of the `set_scan_configuration` command:

```
dc_shell> set_scan_configuration -add_lockup false
```

To add lock-up latches at the end of each scan chain to assist with potential block-to-block timing issues during core integration, use the `-insert_terminal_lockup` option of the `set_scan_configuration` command:

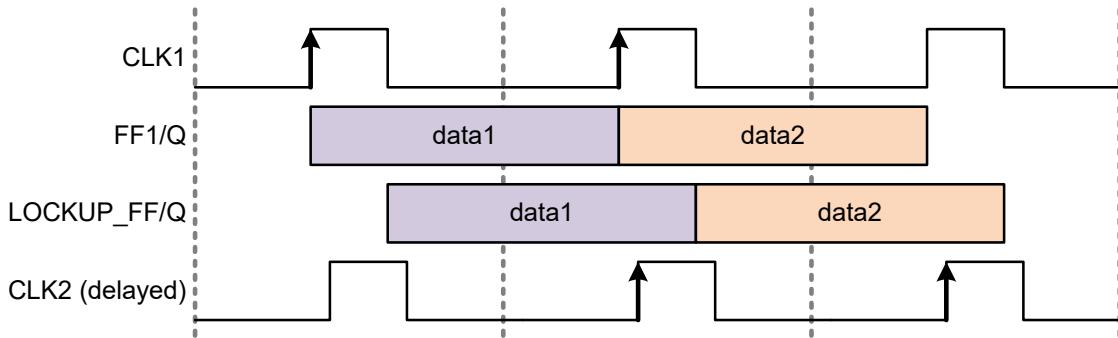
```
dc_shell> set_scan_configuration -insert_terminal_lockup true
```

The default lock-up element type is a level-sensitive lock-up latch. To use a lock-up flip-flop instead, use the `-lockup_type` option of the `set_scan_configuration` command:

```
dc_shell> set_scan_configuration -lockup_type flip_flop
```

When a lock-up flip-flop is used, the data is held as shown in [Figure 85](#).

Figure 85 Timing for Two Scan Cells With a Lock-Up Flip-Flop



Regardless of your selected scan style or configuration, you can explicitly add scan lock-up elements to your scan chain by using the `set_scan_path` command.

For successful lock-up operation, the falling edge of the current scan cell must occur after or concurrent with the rising edge of the next scan cell. This requirement is always inherently met when DFT Compiler inserts lock-up elements between scan chain cells. However, when you are manually inserting lock-up elements with the `set_scan_path` command, you must ensure that this requirement is met.

Use the `preview_dft -show cells` command to see where the `insert_dft` command will insert scan lock-up elements in your scan chain:

```
dc_shell> preview_dft
...
Scan chain '1' (test_si --> Z[3]) contains 4 cells:
Z_reg[0]                                (CLK1, 45.0, rising)
Z_reg[1] (1)                               (CLK2, 45.0, rising)
Z_reg[2]                                (CLK1, 45.0, rising)
Z_reg[3]
```

You can also use the `scan_lockup` cell attribute to locate lock-up elements:

```
dc_shell> set lockup_cells \
           [get_cells -hierarchical * -filter {scan_lockup==true}]
```

Automatically Creating Skew Subdomains Within Clock Domains

This topic describes how the tool can add lock-up latches within a clock domain between subdomains that might have higher skew between them.

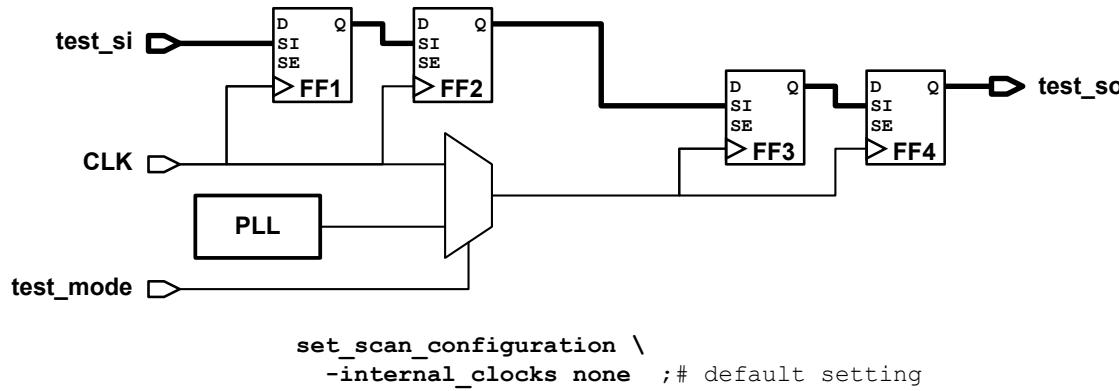
Note:

This feature is only supported for the multiplexed flip-flop scan style.

For the purpose of building scan chains, the `insert_dft` command, by default, treats the entire clock network driven by a given clock source as the same-skew clock signal.

Consider the netlist shown in [Figure 86](#), which shows a clock network structure before clock tree synthesis. By default, the `insert_dft` command treats all four flip-flops as belonging to the same-skew top-level clock signal, CLK.

Figure 86 Circuit With Same Top-Level Clock Driving Internal Clock Signals



Note that the MUX cell introduces a delay in the clock network. If clock tree synthesis balances the test mode clock latency equally to all flip-flops, the MUX cell should not cause any timing problems. However, because clock tree synthesis might not consider the test mode clock tree latencies used for scan shift, a potential scan path hold violation could occur at FF3/SI.

To avoid creating this potential hold time violation, you can treat the scan cells downstream from any multi-input cell as a different *skew subdomain* within the clock domain, driven by their own internal clock pin (such as the MUX output pin).

To do this, use the following command:

```
dc_shell> set_scan_configuration -internal_clocks multi
```

This command instructs the `insert_dft` command to

- Identify all multiple-input gates (such as MUX cells) in each clock network.

Note:

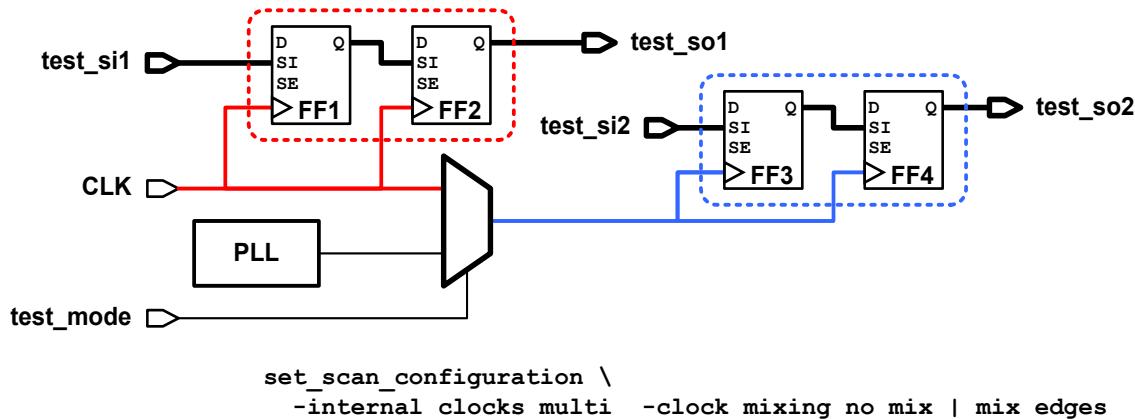
Integrated clock-gating cells, which have the `clock_gating_integrated_cell` attribute defined, are not considered; they are transparent for the determination of skew subdomains.

- Create an internal clock at the output pins of these gates.
- Treat these internal clocks as skew subdomains of the parent clock.

This feature affects only scan chain architecture; the clock network is still a single test clock domain for all other DFT operations, including writing out the test protocol.

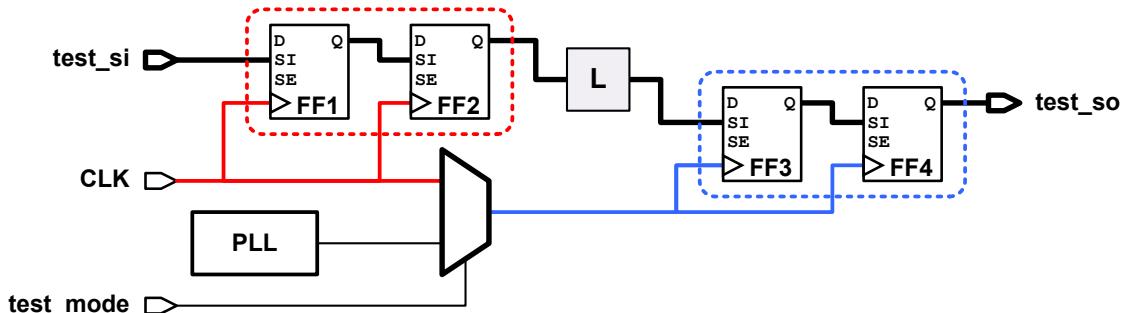
The resulting scan chains depend on the current clock-mixing setting, which is controlled by the `-clock_mixing` option of the `set_scan_configuration` command. If clock mixing is disabled by specifying the `no_mix` or `mix_edges` clock-mixing mode, the `insert_dft` command creates separate scan chains for each internal clock, as shown in [Figure 87](#).

Figure 87 Circuit With Internal Clocks With Clock Mixing Disabled



If clock mixing is enabled by specifying the `mix_clocks` or `mix_clocks_not_edges` clock-mixing mode, the `insert_dft` command can use lock-up latches to keep the scan cells from different internal clocks on the same scan chain, as shown in [Figure 88](#).

Figure 88 Circuit With Internal Clocks With Clock Mixing Enabled



```
set_scan_configuration \
    -internal_clocks multi -clock_mixing mix_clocks | mix_clocks_not_edges
```

If global clock mixing is disabled, you can still enable clock mixing for the internal clocks within each parent clock domain by using the `-mix_internal_clock_driver` option of the `set_scan_configuration` command:

```
dc_shell> set_scan_configuration \
    -internal_clocks multi \
    -clock_mixing no_mix | mix_edges \
    -mix_internal_clock_driver true
```

You can create skew subdomains within specific clock domains by using the `-internal_clocks` option of the `set_dft_signal` command. The following command tells the `insert_dft` command to create internal clocks at multiple-input cells only for the CLK domain:

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

If you set different `-internal_clocks` values using the `set_scan_configuration` and `set_dft_signal` commands, the more specific setting applied with the `set_dft_signal` command takes precedence. For example, assume that you set the following opposing `-internal_clocks` values 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 MUX cells or other multiple-input gates are treated as separate clocks. All other clocks in the design are treated according to the default configuration.

This feature is similar to the `-associated_internal_clocks` feature described in [Manually Creating Skew Subdomains at Associated Internal Pins on page 256](#), except that the internal clocks are created at all multi-input cell output pins instead of only user-specified pins.

When you use user-defined test points or test points inserted by AutoFix, testability logic might be inserted in the clock network. The `preview_dft` command does not see internal clocks created by test points, but the `insert_dft` command does. For more information about test points, see [Chapter 10, Advanced DFT Architecture Methodologies](#).

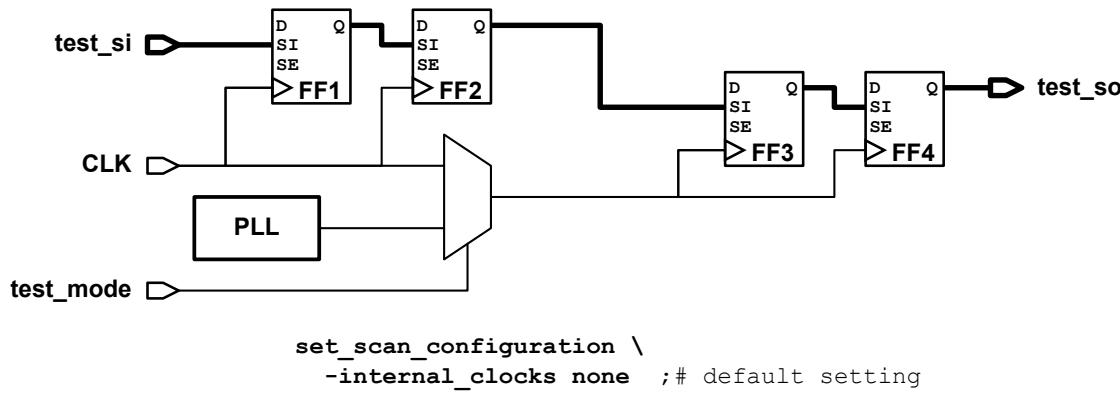
Manually Creating Skew Subdomains at Associated Internal Pins

This topic describes how you can manually define subdomains of a parent clock network that might have higher skew between them.

For the purpose of building scan chains, DFT insertion, by default, treats the entire clock network driven by a given clock source as the same-skew clock signal.

Consider the netlist shown in [Figure 89](#), which shows a clock network structure before clock tree synthesis. By default, the `insert_dft` command treats all four flip-flops as belonging to the same-skew top-level clock signal, CLK.

Figure 89 Circuit With Same Top-Level Clock Driving Internal Clock Signals



Note that the MUX cell introduces a delay in the clock network. If clock tree synthesis balances the test mode clock latency equally to all flip-flops, the MUX cell should not cause any timing problems. However, because clock tree synthesis might not consider the test mode clock tree latencies used for scan shift, a potential scan path hold violation could occur at FF3/SI.

To avoid creating this potential hold time violation, you can treat the scan cells downstream from the MUX cell as a different *skew subdomain* within the clock domain, driven by their own internal clock pin (the MUX output pin).

To do this, define the scan clock as follows:

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -associated_internal_clocks {UMUX/Z}
```

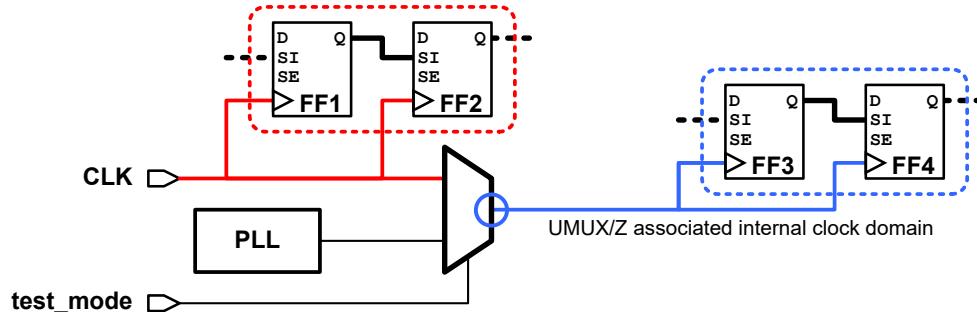
This command instructs the `insert_dft` command to

- Create an internal clock at each associated internal pin in the list
- Treat these internal clocks as skew subdomains of the parent clock

This feature affects only scan chain architecture; the clock network is still a single test clock domain for all other DFT operations, including writing out the test protocol.

In the previous example, the clock network contains a skew subdomain driven by UMUX/Z, plus the remainder of the parent clock domain, as shown in [Figure 90](#). These are treated as separate clock domains according to the `set_scan_configuration -clock_mixing` setting applied to the design.

Figure 90 Circuit With Top-Level Clock Source and Associated Internal Clock Pin



```
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port CLK -associated_internal_clocks {UMUX/Z}
```

This feature is similar to the `-internal_clocks` feature described in [Automatically Creating Skew Subdomains Within Clock Domains on page 253](#), except that the internal clocks are created only at the user-specified pins instead of all multi-input cell output pins.

Associated internal clocks take precedence over any `-internal_clocks` specifications.

Limitations

Note the following requirements and limitations:

- Associated internal clocks can be defined only on leaf pins, not hierarchical pins.
- Pre-DFT DRC drives the clock signal directly at both the clock source and the internal pins, allowing the clock signal to bypass cells that are black boxes in synthesis. However, post-DFT DRC drives the clock signal only at the clock source.

To propagate the clock through blockages during post-DFT DRC, use a custom `test_setup` procedure (if initialization vectors are needed) or the `test_simulation_library` variable (if Verilog simulation models are needed).

- This feature is used only with `-view existing_dft` clock signal definitions.
- The association is valid only when `-type` is `MasterClock`, `ScanMasterClock`, `ScanSlaveClock`, or `ScanClock`.
- To remove the internal pin associations, you must use the `remove_dft_signal` command to remove both the DFT signal and the association list.
- The `-mix_internal_clock_driver` option of the `set_scan_configuration` command does not affect associated internal clocks defined in the current design.
- The `-hookup_sense` option has no effect. You can associate only the same clock edge from a top-level clock edge to of a list of pins.
- The `report_dft_signal` command does not show the associated internal pins.

Manually Creating Skew Subdomains With Scan Skew Groups

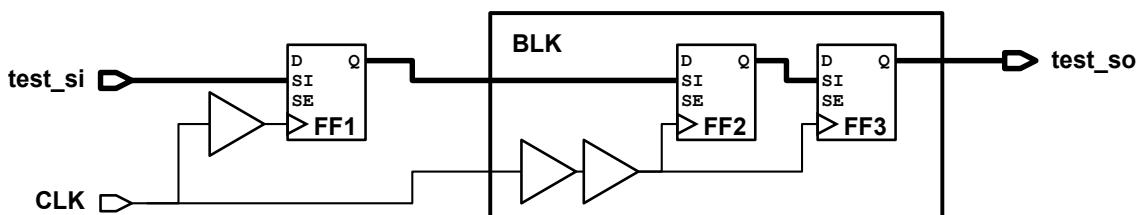
This topic describes how to define a skew subdomain on any arbitrary set of scan cells. Such a set of scan cells is called a *scan skew group*.

Note:

This feature is only supported for the multiplexed flip-flop scan style.

In some cases, you might want to provide manual guidance for lock-up latch insertion between areas of the design with potentially differing clock tree latencies. Consider [Figure 91](#), in which block BLK has a different clock latency than the top-level logic.

Figure 91 Circuit With Clock Tree Containing Multiple Latency Regions



A lock-up latch at the scan input pin of BLK would prevent hold violations along the scan path. However, the scan chain is entirely within the same scan clock domain, and there are no multi-input cells that allow the `-internal_clocks` option to be used.

You can use *scan skew groups* to provide manual guidance for lock-up latch insertion. A scan skew group is a group of scan cells that might have a different clock latency

characteristic than other parts of the design. The DFT architect treats the scan skew group as a unique *skew subdomain*.

To define a scan skew group, use the `set_scan_skew_group` command:

```
set_scan_skew_group
  group_name
  -include_elements {include_list}
```

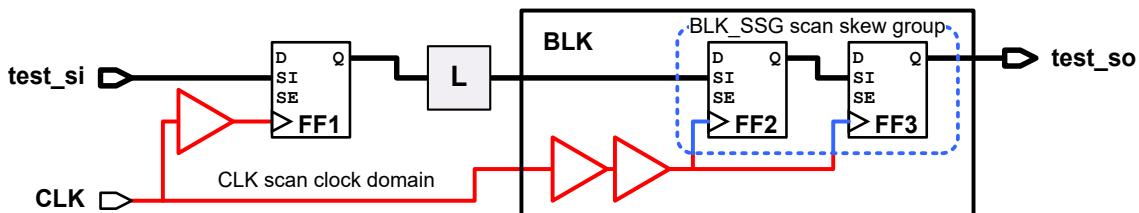
Each scan skew group has a unique name for identification. The include list can contain leaf cells, hierarchical cells, and CTL-modeled cells. Wildcards and collections are supported. You can define as many scan skew groups in your design as needed. You cannot include the same scan cell in multiple scan skew groups.

For the previous example, consider the following scan skew group definition:

```
dc_shell> set_scan_skew_group BLK_SSG -include_elements {BLK/FF*}
```

If clock mixing is enabled, DFT insertion adds a lock-up latch between the top-level and block-level scan cells, as shown in [Figure 92](#). (If clock mixing is disabled, DFT insertion keeps the top-level and block-level scan cells in separate scan chains, not shown.)

Figure 92 Circuit With Lock-Up Latch Due to Scan Skew Group Definition



Scan skew groups override the normal scan clock domain identification behaviors such as

- Scan clock name
- The `-internal_clocks` option of the `set_scan_configuration` command
- The `-mix_internal_clock_driver` option of the `set_scan_configuration` command

The `preview_dft -show {scan_clocks}` command reports the scan skew group name as the clock name. For example,

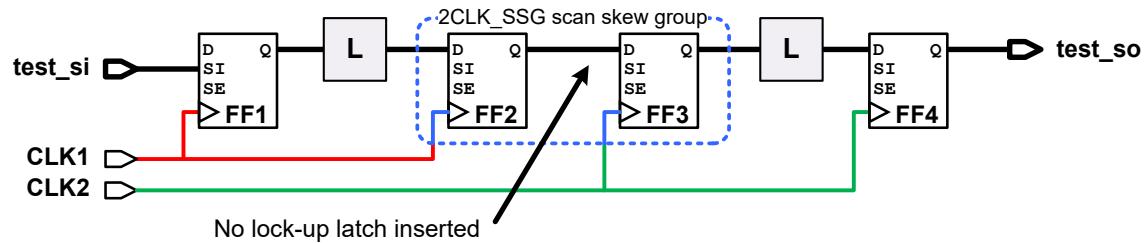
```
Scan chain '1' (test_si1 --> test_so1) contains 3 cells:
  FF1 (1)                                (CLK, 45.0, rising)
  BLK/FF2                               (BLK_SSG/Z, 45.0, rising)
  BLK/FF3
```

If you include scan cells from different scan clock domains in the same scan skew group, the `preview_dft` and `insert_dft` commands issue a warning message:

Scan skew group 2CLK_SSG contains scan cells from the following clock domains: CLK1, CLK2. (TEST-1923)

In this case, the DFT architect treats all cells in the scan skew group as if they are in the same clock domain. No lock-up latches will be inserted between them, as shown by the example in [Figure 93](#). This can occur even if clock mixing is disabled because the scan cells in the scan skew group are no longer treated as belonging to different clock domains.

Figure 93 Circuit With Scan Skew Group Spanning Multiple Clock Domains



Scan skew groups override only clock identity considerations. They do not override the clock timing considerations of scan chain architecture such as

- Scan clock waveform timing
- Scan clock edge polarity

Defining Scan Chains by Scan Clock

You might want to define a scan chain that is specific to a particular scan clock domain. To do this, use the `-scan_master_clock` option of 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 includes additional scan cells clocked by other clocks if the clock-mixing requirements allow.

If you use the `-edge` option with the `-scan_master_clock` option when defining a scan path using the `set_scan_path` command, the tool includes only the elements controlled by the specified edge of the specified clock in the scan chain. The valid arguments to the `-edge` option are `rising` and `falling`.

For example,

```
dc_shell> set_scan_path c1 -view spec \
           -scan_master_clock clk1 -edge rising
```

In this example, scan chain c1 will contain elements that are triggered by the rising edge of clock clk1.

Note the following limitations of the `-edge` option:

- When there are no elements present for the defined scan chains, the scan path name is reused.
- This option is not supported with multivoltage designs.
- This option is not supported with the other `set_scan_path` options, such as the `-head`, `-tail`, `-ordered`, and `-length` options.

If the specifications in the `set_scan_path` command cannot be met, they are not applied.

Handling Multiple Clocks in LSSD Scan Styles

This topic provides information on handling multiple clocks in level-sensitive scan designs (LSSD) scan styles.

Using Multiple Master Clocks

In LSSD scan designs, 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 20](#) accomplishes this.

Example 20 Command Sequence for Multiple Master Clocks in LSSD

```
dc_shell> set_scan_configuration -style lssd

# 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 [get_object_name [all_registers -clock cclk1]]
dc_shell> set_scan_path 1 -include_elements $cclk1_cells
dc_shell> create_clock en2 -name cclk2 -period 100
dc_shell> set cclk2_cells [get_object_name [all_registers -clock cclk2]]
dc_shell> set_scan_path 2 -include_elements $cclk2_cells
```

```
# preview scan configuration and implement
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> preview_dft -show all
dc_shell> insert_dft
```

Dedicated Test Clocks for Each Clock Domain

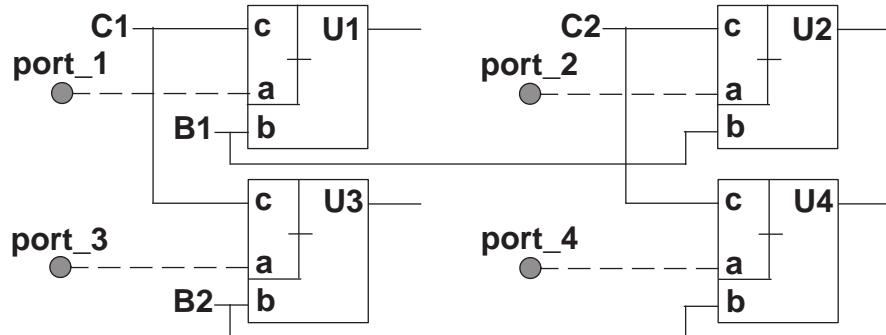
The `insert_dft` command creates clocks that are used only for test purposes when it routes scan chains by using the following scan styles:

- LSSD (which includes clocked LSSD)
- Scan-enabled 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 master and slave system clocks. For example, in [Figure 94](#), 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, resulting in four unique clock sets. As a result, the `insert_dft` command adds four test clocks, one for each unique clock set.

Figure 94 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

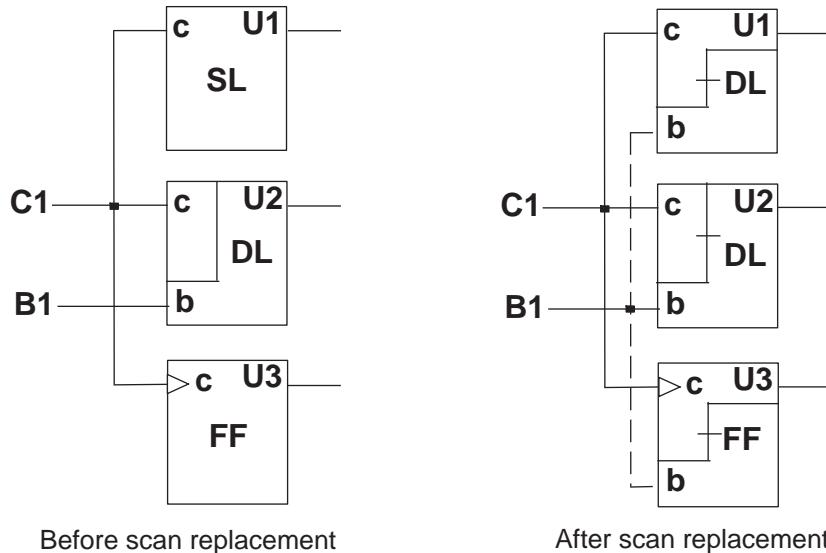
The `insert_dft` command uses the following guidelines when connecting slave clock pins of single-latch and flip-flop elements after scan replacement:

- 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 95](#).

Note:

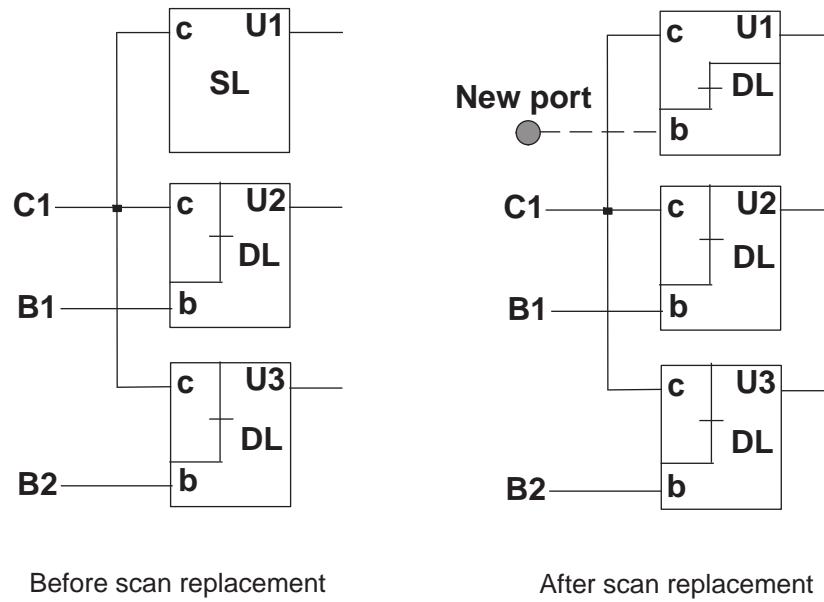
For clarity, the A clock is omitted in [Figure 95](#) through [Figure 98](#) after scan replacement.

Figure 95 Single-Latch and Double-Latch Cells With the Same System Clock



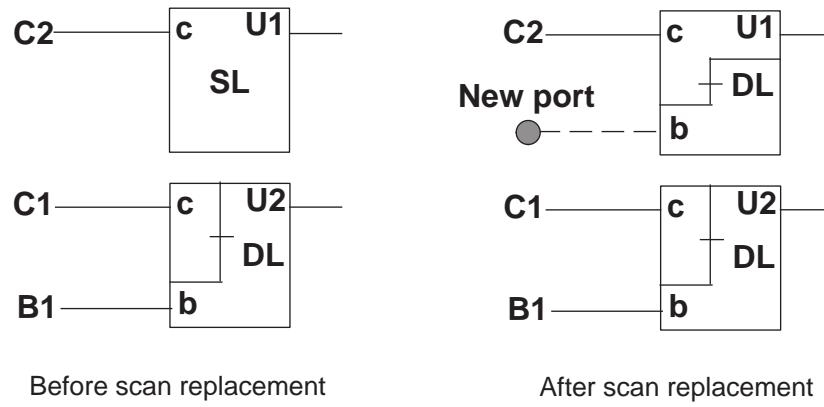
- Connect to a new slave clock, creating a new one if necessary, if a system clock drives multiple cells with different slave clocks. See [Figure 96](#).

Figure 96 Single-Latch and Double-Latch Cells Clocked by the Same System Clock



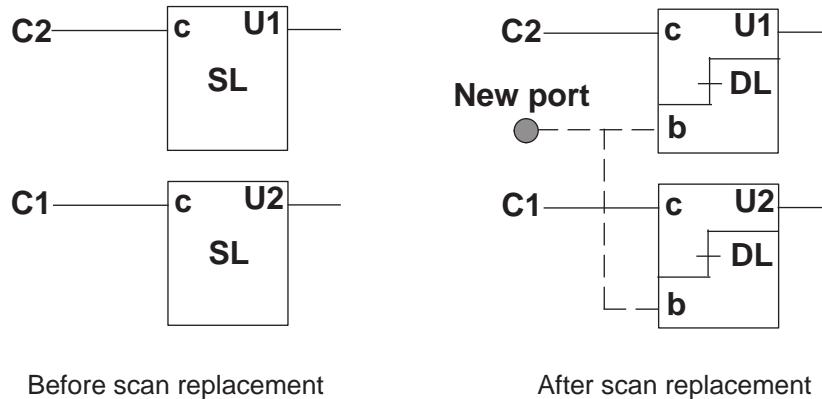
- Connect to a new slave clock port, creating one if necessary, if double-latch cells are driven by different clocks. See [Figure 97](#).

Figure 97 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 98](#).

Figure 98 Connecting Slave Clock Pin: No Double-Latch Cells



Configuring Clock-Gating Cells

The following topics discuss how to incorporate clock-gating logic into your DFT design:

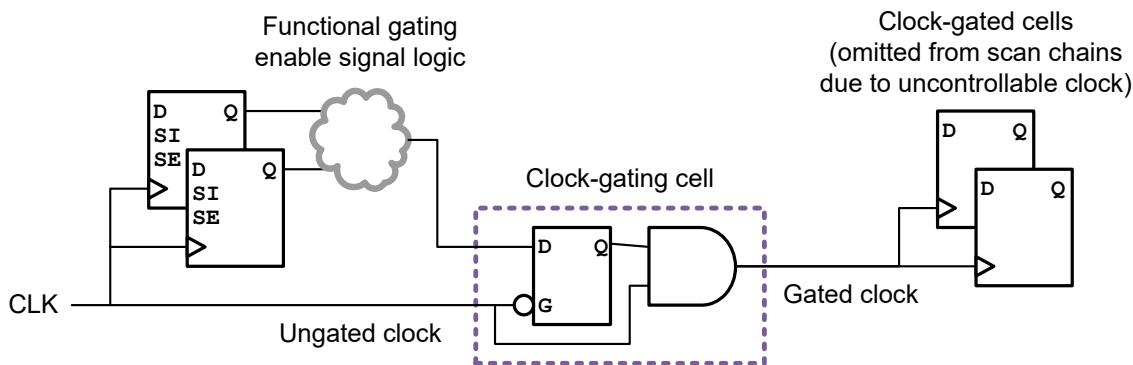
- [Introduction to Clock Gating in DFT Flows](#)
- [Clock-Gating Control Points](#)
- [Discrete-Logic Clock-Gating Cells and Integrated Clock-Gating Cells](#)
- [Inferred and Instantiated Clock-Gating Cells](#)
- [Choosing a Clock-Gating Control Point Configuration](#)
- [Reporting Unconnected Clock-Gating Cell Test Pins During Pre-DFT DRC](#)
- [Automatically Connecting Test Pins During DFT Insertion](#)
- [Specifying Signals for Clock-Gating Cell Test Pin Connections](#)
- [Identifying Clock-Gating Cells in an ASCII Netlist Flow](#)
- [Limitations](#)

Introduction to Clock Gating in DFT Flows

Clock gating provides a way to disable, or *gate*, the clock signal to a set of flip-flops to reduce their switching power consumption. However, clock gating requires special consideration in DFT-inserted designs.

To be included in scan chains, sequential cells must be reliably clocked during scan shift. In the example in [Figure 99](#), the clock-gating control signal is driven by scan cells in the scan chain, which causes pre-DFT DRC to identify the gated clock signal as uncontrollable. As a result, the clock-gated cells are omitted from the scan chain, and test observability is reduced at their register inputs and test controllability is reduced at their register outputs.

Figure 99 Example of Clock-Gating Cell Without Testability Control Signal



To resolve this, clock-gating cells require an override control signal to keep the clock signal always-active in scan shift mode. The following sections describe different ways of implementing and controlling this override control signal of clock-gating cells.

Clock-Gating Control Points

The following topics discuss how clock-gating control points can be configured:

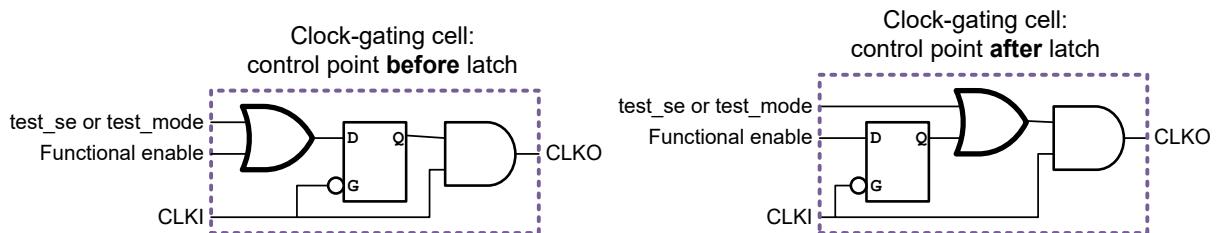
- [Configuring Clock-Gating Control Points](#)
- [Scan-Enable Signal Versus Test-Mode Control Signal](#)
- [Improving Observability When Using Test-Mode Control Signals](#)

Configuring Clock-Gating Control Points

In a clock-gating cell, a *control point* is a testability logic gate that allows the clock to be forced always-active when a test *control signal* is asserted. There are two degrees of freedom in control point implementation, as shown in [Figure 100](#):

- The control point can be placed before or after the latch element.
- The control signal can be driven by the scan-enable or test-mode signal.

Figure 100 Clock-Gating Control Points Before and After Latch



The `set_clock_gating_style` command configures these two aspects using the following options:

- The `-control_point` option specifies where to insert the control point relative to the latch. It can be set to `none`, `before`, or `after`. The default is `none`.
- The `-control_signal` option specifies what type of control signal to use. It can be set to `test_mode` or `scan_enable`. The default is `scan_enable`.

You can use the `-control_signal` option only when the `-control_point` option is set to `before` or `after`. If an existing signal of the specified type has been defined with the `set_dft_signal` command, it is used. Otherwise, a new signal is created. See [Automatically Connecting Test Pins During DFT Insertion on page 277](#) for details.

For most designs, the “before” control point style driven by the scan-enable signal is recommended for the following reasons:

- The “before” control point ensures that no combinational path exists from the control signal input port to the downstream clock pins of the scan cells.
- If the “after” control point is used and both phases of the clock are gated in the design, there is no time at which the control signal can cleanly toggle without truncating an active clock pulse.
- Most integrated clock-gating (ICG) cells are implemented using control points before the latch. For more information on ICG cells, see [Discrete-Logic Clock-Gating Cells and Integrated Clock-Gating Cells on page 271](#).

- A scan-enable control signal ensures that the gated clocks are always-active during scan shift, but that the functional clock-gating paths can still be tested during scan capture.
- A test-mode control signal prevents the functional clock-gating paths from being tested, requiring additional testability logic to be inserted in the design.

The following command implements the recommended control point style:

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

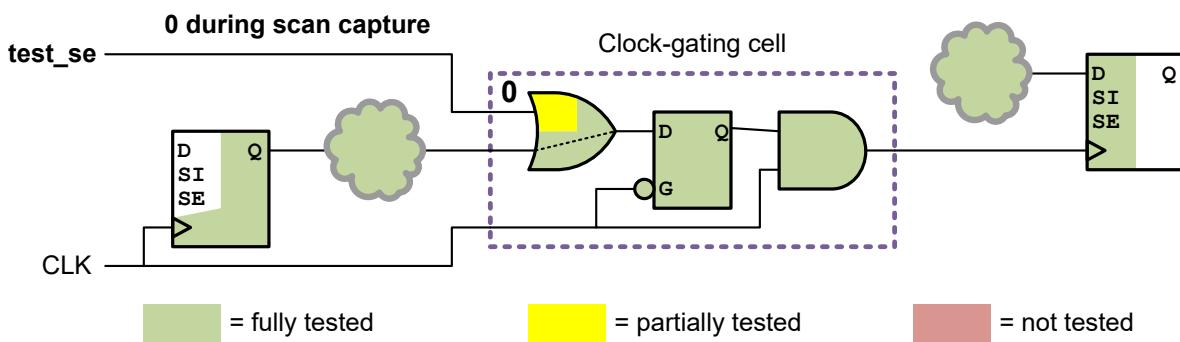
Scan-Enable Signal Versus Test-Mode Control Signal

The scan-enable and test-mode signals differ in the following ways:

- A scan-enable signal is asserted only during scan shift.
- A test-mode signal is asserted during the entire test (scan shift and scan capture).

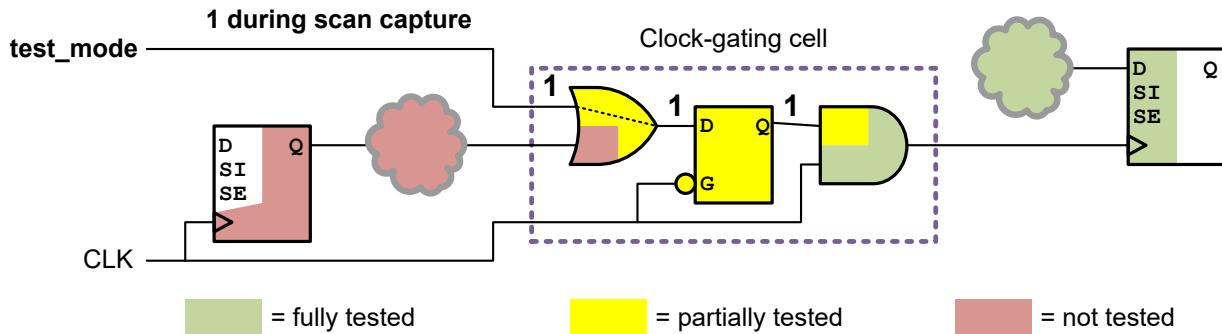
Using the scan-enable signal as the clock-gating control signal typically provides higher fault coverage than the test-mode signal because the functional clock-gating path is exercised in scan capture mode. Fault coverage with the scan-enable signal is comparable to a circuit without clock gating, as shown in [Figure 101](#).

Figure 101 Test Coverage With Scan-Enable Signal



If you use a test-mode signal as the clock-gating control signal, the test-mode signal is asserted during both scan shift and scan capture. This bypasses the functional clock-gating control logic completely and prevents it from being tested in scan capture mode, as shown in [Figure 102](#). In addition, the clock-gating enable signal asserted by the test-mode signal can be tested only for stuck-at-0 faults (assuming an active-high test-mode signal).

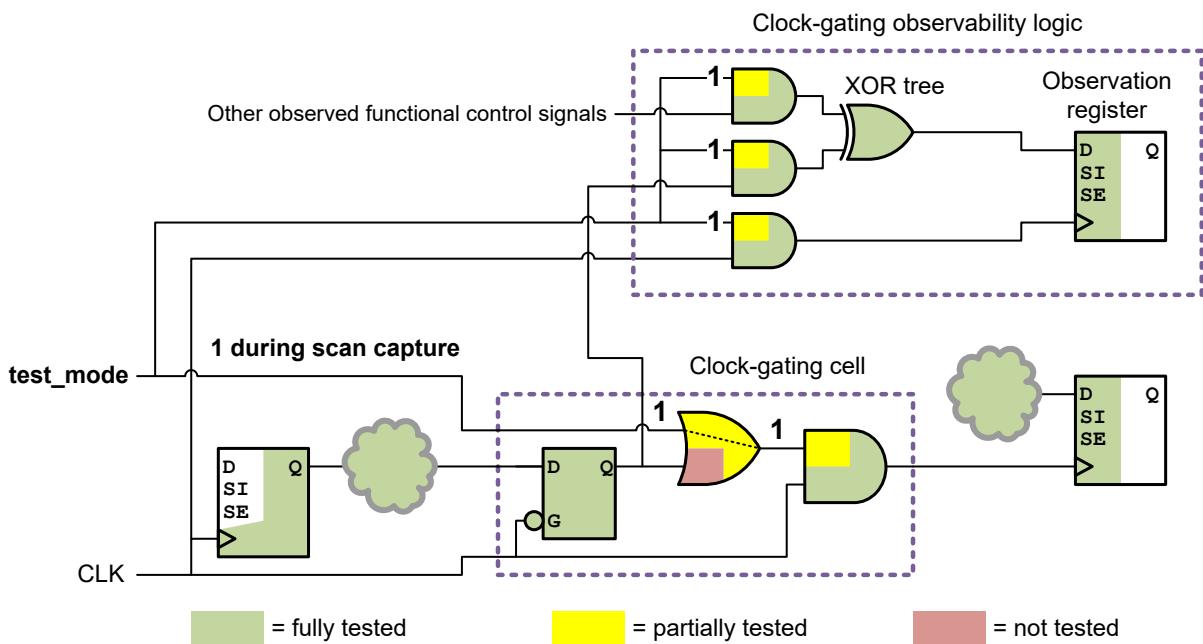
Figure 102 Test Coverage With Test-Mode Signal



Improving Observability When Using Test-Mode Control Signals

If you must use a test-mode signal as the clock-gating control signal, you can enable an observability logic feature that improves coverage of the functional logic generating the gating signal. [Figure 103](#) shows an observability logic example.

Figure 103 Observability Logic in Clock-Gating Circuits



In test mode, an XOR tree observes the functional gating control signals from one or more clock-gating cells; the output of the XOR tree is captured by a dedicated observability register. This observability register is included in the scan chains, but its output is not otherwise connected functionally in the design. AND or NAND gates (depending on synthesis) prevent the observability logic from consuming power during mission mode.

Note that placing the control point after the latch allows the latch to be tested by the observability logic.

To enable clock-gating observability logic when using test-mode control signals, use the following command:

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

The tool inserts an observability logic structure in each design hierarchy level where clock-gating cells exist. By default, the maximum depth of each XOR tree is 5, which means that a maximum of $2^5 = 32$ clock-gating control signals can be observed within a hierarchy level by one observability register. The tool adds more observability registers as needed.

To change the maximum depth of the XOR tree, use the following command:

```
dc_shell> set_clock_gating_style \
           -observation_logic_depth logic_depth
```

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

Discrete-Logic Clock-Gating Cells and Integrated Clock-Gating Cells

Clock-gating cells can be classified by cell structure into two types:

- Discrete-logic clock-gating cells

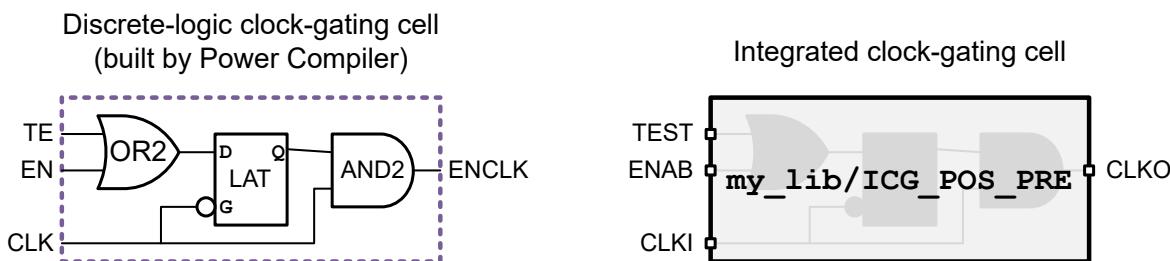
These are clock-gating cells that are built from discrete logic library gates, such as a latch gate and an AND gate.

- Integrated clock-gating cells (ICGs)

These are library cells that contain the clock-gating logic within a single integrated library cell. Their Liberty library cell models describe information such as the type of control point (none, before, or after) and the setup and hold requirements of the gating signal versus the clock signal.

[Figure 104](#) shows an example of both types of clock-gating cells (with active-high clock signals, active-high control signals, and the control points before the latch).

Figure 104 Discrete-Logic and Integrated Clock-Gating Cells



For both types of clock-gating cell, the pin that drives the control signal is called the *test pin* of the clock-gating cell.

You can use Power Compiler to insert either type of clock-gating cell. Use the `set_clock_gating_style` command to configure the desired clock-gating cell characteristics. For discrete-logic clock-gating cells, Power Compiler builds customized gate-level clock-gating structures (contained in a level of hierarchy) according to your specifications. For integrated clock-gating cells, Power Compiler searches the available target libraries and uses integrated clock-gating cell that match your specifications, or you can specify a particular cell by name.

Inferred and Instantiated Clock-Gating Cells

The following topics describe the two ways to incorporate clock-gating logic into your design:

- [Inferring Clock-Gating Cells Using Power Compiler](#)
- [Instantiating Clock-Gating Cells in the RTL](#)

Inferring Clock-Gating Cells Using Power Compiler

You can use Power Compiler to automatically infer clock-gating logic where load-enabled registers are described in the RTL. The resulting gating logic is typically inserted at the leaf-level registers of the clock tree based on each register's functionality, which is referred to as *fine-grained* clock gating.

When Power Compiler inserts clock-gating cells in the design (either discrete-logic or integrated clock-gating cells), it automatically annotates attributes so that DFT Compiler can identify the clock-gating cells and make the necessary test signal connections during DFT insertion.

To automatically insert clock-gating cells using Power Compiler during initial RTL synthesis, use the `-gate_clock` option of the `compile` or `compile_ultra` command. For example,

```
dc_shell> compile_ultra -scan -gate_clock
```

Power Compiler ties the test pins of inserted clock-gating cells to ground. The resulting clock-gating cells behave only in their functional capacity until you run DFT insertion, at which point the test pins are connected to the appropriate test control signals.

For more information on using Power Compiler to configure and insert clock-gating logic in your design, see “Clock Gating” chapter of the Power Compiler User Guide.

Instantiating Clock-Gating Cells in the RTL

You can describe clock-gating logic in your RTL. This is often done near clock tree sources to gate entire clock domains for power savings, which is referred to as *coarse-grained* clock gating.

Unlike inferred clock-gating cells, instantiated clock-gating (ICG) cells are *not* automatically recognized by DFT insertion. You must manually identify them so that their test pins are connected to a clock-gating control signal.

There are three ways to identify instantiated ICG cells:

- `set_app_var power_cg_auto_identify true`

This variable setting causes the tool to look for and identify any not-yet-identified ICG cells. This analysis is performed each time a command that works on clock-gating circuitry is called.

- `identify_clock_gating [-gating_elements cells]`

This command causes the tool to look for and identify any not-yet-identified ICG cells. This analysis is performed when the command is run. You can use the `-gating_elements` option to restrict identification to particular cells.

- `set_dft_clock_gating_pin -pin_name pin object_list`

This command manually identifies ICG cells by specifying their test pins. For details, see [Connecting User-Instantiated Clock-Gating Cells on page 421](#).

DFT insertion connects only test pins that are undriven or driven by a logic constant; these valid test pins are reported by TEST-130 messages during pre-DFT DRC. Test pins with other existing connections are left unchanged, and no message is reported for them.

Clock-gating cell identification method	Reporting command	Warning for existing RTL connections?
<code>set_app_var power_cg_auto_identify true</code>	<code>report_clock_gating (after automatic identification runs)</code>	No
<code>identify_clock_gating</code>	<code>report_clock_gating</code>	No
<code>set_dft_clock_gating_pin</code>	<code>report_dft_clock_gating_pin</code>	Yes (TEST-2059)

For non-ICG clock-gating logic (built with discrete logic gates), you must incorporate your own testability logic; DFT insertion does not make connections to manually instantiated discrete-logic clock-gating logic.

Instantiated ICG Cells With Existing RTL Test-Pin Connections

For instantiated ICG cells, an “existing test-pin connection” refers to an ICG test pin driven by a non-constant signal in the RTL.

Control signals used by existing test-pin connections must be defined in both the `spec` and `existing_dft` signal views:

```
# define ScanEnable signal for clock-gating cells
set_dft_signal -view spec \
    -type ScanEnable -port SE_ICG
```

```
set_dft_signal -view existing_dft \
    -type ScanEnable -port SE_ICG ;# needed for existing RTL connections
```

The `existing_dft` signal definition ensures that pre-DFT DRC understands the existence and function of that signal. Otherwise, scan cells driven by that ICG cell incur D1 or D9 violations and are omitted from the scan chains.

Normally, ICG cells with existing test-pin connections do not need to be identified to the tool. However, they must be identified if you are also inserting LogicBIST self-test; see [Ensuring Testability for Integrated Clock-Gating Cells on page 1029](#).

Choosing a Clock-Gating Control Point Configuration

Table 28 shows the possible combinations of latch-based clock gating, clock waveforms, control signals, and control point location you can use.

Table 28 Latched-Based Clock-Gating Configurations

Clock-gating style	Clock waveform	Control signal	Control point location	Gated register can be scanned?
Latch-based gating for positive-edge flip-flops		test_mode	Before latch	Yes
			After latch	Yes
		scan_enable	Before latch	Yes
			After latch	Yes
		test_mode	Before latch	Yes ⁵
			After latch	Yes
		scan_enable	Before latch	Yes ⁶
			After latch	No
Latch-based gating for negative-edge flip-flops		test_mode	Before latch	Yes
			After latch	Yes
		scan_enable	Before latch	Yes
			After latch	Yes
		test_mode	Before latch	Yes ⁵
			After latch	Yes

5. This configuration requires additional initialization cycles to be manually specified for the test protocol.

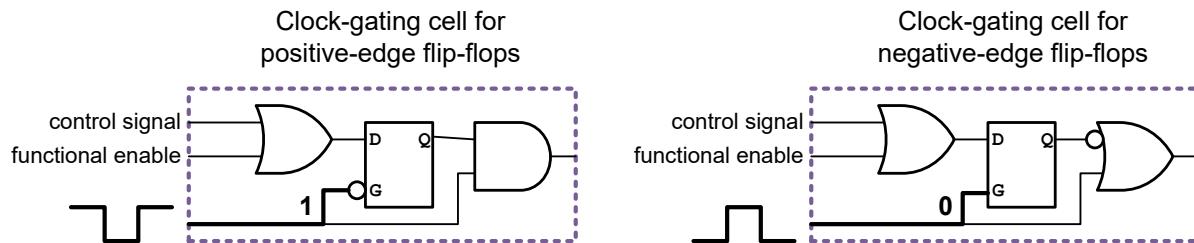
Clock-gating style	Clock waveform	Control signal	Control point location	Gated register can be scanned?
		scan_enable	Before latch	Yes ⁶
			After latch	No

For each combination, the last column indicates whether the gated register can be included in scan chains. Four special cases, marked with footnotes, require additional consideration as described in the following section.

Initialization for Special Cases of Before-Latch Control Points

In the four special cases marked by footnotes in [Table 28](#), the gating latch is inactive at the beginning of the test program (time = 0). This unknown latch state causes an X value to reach the gated flip-flops, which would normally prevent them from being included in scan chains. [Figure 105](#) shows the clock waveform and clock-gating logic for these cases.

Figure 105 Special Clock-Gating Cases With Inactive Latch at Beginning of Test Clock Cycle



If you are using a scan-enable control signal, which asserts and de-asserts during every test pattern, the DRC engine performs an analysis of the clock-gating logic to verify a known state in the latch. This analysis supports only one such level of special-case clock gating, although you can have additional levels of non-special-case clock gating.

If you are using a test-mode control signal, to achieve a known state in the latch, you must add a clock pulse to the `test_setup` section of the test protocol. Use the following `set_dft_drc_configuration` command to update the `test_setup` section with the clock pulse:

```
dc_shell> set_dft_drc_configuration -clock_gating_init_cycles 1
```

If you have multiple cascaded latch-based clock-gating cells and the first latch is loaded with the test-mode signal, use the following `set_dft_drc_configuration` command to update the `test_setup` section with the specified number of clock pulses:

```
dc_shell> set_dft_drc_configuration -clock_gating_init_cycles n
```

6. This configuration supports only one level of clock gating.

Here, n equals 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 clock-gating 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.

Reporting Unconnected Clock-Gating Cell Test Pins During Pre-DFT DRC

When you run pre-DFT DRC, the tool issues a TEST-130 message for every clock-gating cell (discrete-logic or integrated) known to DFT Compiler whose test pin is not yet connected. For example,

```
Warning: Clock gating cell sub1/clk_gate_ZI_reg has unconnected test pin.  

(TEST-130)  

Information: Cells with this violation : sub1/clk_gate_ZE_reg,  

sub1/clk_gate_ZI_reg, sub2/clk_gate_ZE_reg, sub2/clk_gate_ZI_reg.  

(TEST-283)
```

If clock-gating cell test pin connections are enabled (which is the default), this message is informational and no action is needed. DFT insertion hooks up the test pin to the appropriate test signal. Although the test pin is not yet connected, pre-DFT DRC models the clock-gating cell as if connected.

If clock-gating cell test pin connections are disabled (using the `-connect_clock_gating disable` option of the `set_dft_configuration` command), this message is a warning that the indicated test pin is unconnected and will not be connected by DFT insertion. For more information on this case, see the TEST-130 man page.

This message is reported only for clock-gating cell test pins known to DFT Compiler. For clock-gating cells inserted by Power Compiler, no action is needed. For manually instantiated integrated clock-gating cells, use the `set_dft_clock_gating_pin` command to identify the test pins to be connected; if you do not identify these test pins, pre-DFT DRC does not issue this message or model the clock-gating cells as controllable, and DFT insertion will not connect the test pins.

If the test pin of a clock-gating cell is already connected, no TEST-130 message is reported by pre-DFT DRC, and the flip-flops controlled by the clock-gating cell are put onto the scan chain during DFT insertion provided no other violations exist for the cell.

Automatically Connecting Test Pins During DFT Insertion

When you run the `insert_dft` command, the tool automatically hooks up any unconnected test pins of clock-gating cells known to DFT Compiler, corresponding to the TEST-130 messages issued during pre-DFT DRC.

Note:

The `insert_dft` command preserves any clock-gating test pins with preexisting connections; the command creates only missing connections.

DFT insertion makes the connections of the top-level scan-enable or test-mode signals 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.

[Table 29](#) describes the connections made by the `insert_dft` command.

Table 29 Connections Made to the Clock-Gating Cells by insert_dft

Clock-gating control signal	DFT signal defined?	Top-level port used
<code>scan_enable</code>	No <code>set_dft_signal</code> defined with <code>-type ScanEnable</code>	<code>test_se</code> created and connected to test pins of clock-gating cells.
<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. User-defined port <code>test_se</code> used to connect to test pins of clock-gating cells.
<code>test_mode</code>	No <code>set_dft_signal</code> defined with <code>-type TestMode</code>	New port. <code>test_cgtm</code> created to connect to test pins of clock-gating cells.
<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. User-defined port <code>test_mode</code> used to connect to test pins of clock-gating cells.

Note:

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

Clock-gating cell test pins are hooked up by default. To disable this feature, use the following command:

```
dc_shell> set_dft_configuration -connect_clock_gating disable
```

Specifying Signals for Clock-Gating Cell Test Pin Connections

By default, DFT Compiler chooses an available ScanEnable or TestMode signal to connect to clock-gating cell test pins, depending on the type of control signal specified with the `set_clock_gating_style` command. However, you can also define a dedicated ScanEnable or TestMode signal to use for these test pin connections.

Specifying a Global Clock-Gating Control Signal

You can define a global clock-gating ScanEnable or TestMode control signal by using the `-usage clock_gating` option when defining the signal with the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable | TestMode
    -view spec
    -usage clock_gating
    -port port_list
```

The `-type` option must be set to `ScanEnable` or `TestMode` to match the design's clock-gating control-signal style. You can report the control-signal style using the `report_clock_gating -style` command. If these settings do not match, the signal specification is ignored.

The `-view` option must be set to `spec` because DFT insertion makes new connections to the signal.

When you define a clock-gating control signal with the `clock_gating` usage, the `insert_dft` command is limited to using only that signal to connect to the test pins of clock-gating cells. If there are insufficient ScanEnable or TestMode signals for other purposes, DFT Compiler creates additional ScanEnable or TestMode signals as needed.

You can use the `report_dft_signal` and `remove_dft_signal` commands for reporting and removing the specification, respectively.

Specifying Object-Specific Clock-Gating Control Signals

You can also define dedicated ScanEnable or TestMode clock-gating control signals for specific parts of the design by using the `-connect_to` option of the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable | TestMode
    -view spec
    -usage clock_gating
    -port port_list
    [-connect_to object_list]
    [-exclude object_list]
```

The `-connect_to` option specifies a list of design objects that are to use the specified clock-gating control signal. The supported object types are

- Clock-gating cells

For Power Compiler clock-gating cells, specify the hierarchical clock-gating cell. For existing clock-gating cells identified with the `set_dft_clock_gating_pin` command, specify the leaf clock-gating cell.

- Hierarchical cells
- Designs
- Test clock ports

This allows you to make clock-domain-based signal connections. It includes clock-gating cells that gate the specified test clocks. The functional clock behavior is not considered.

Note:

This specification requires that a functional clock also be defined on the test clock port.

- Scan-enable or test-mode pins of CTL-modeled cores

You can also use the `-exclude` option to specify a list of clock-gating cells, hierarchical cells, or design names to exclude from the object-specific control signal.

The following example defines a ScanEnable signal named `SE(CG)` to connect to the test pins of existing clock-gating cells `ICG_CLK100` and `ICG_CLK200`:

```
dc_shell> set_dft_signal \
           -type ScanEnable -view spec -port SE(CG) \
           -usage clock_gating -connect_to {ICG_CLK100 ICG_CLK200}
```

Identifying Clock-Gating Cells in an ASCII Netlist Flow

If you are using an ASCII netlist flow and clock-gating cell attributes are not present, 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.

Limitations

Note the following limitations:

- The `insert_dft` command cannot be used on an unmapped design to connect to clock-gating cells, that is, when a design is still in the RTL stage.
- Only the clock-gating cells recognized by Power Compiler are supported for automatic test pin connection. You must manually specify the test pins of user-instantiated integrated clock-gating cells. For more information, see [Connecting User-Instantiated Clock-Gating Cells on page 421](#).
- If you use the `set_dft_signal -connect_to` command to make clock-domain-based connections to clock-gating cells, only Power Compiler clock-gating cells are considered; user-instantiated clock-gating cells are not considered.
- Clock-gating cell connections are not mode-specific.
- The `preview_dft` and `insert_dft` commands do not report connections made to clock-gating cells.

Specifying a Location for DFT Logic Insertion

By default, DFT Compiler places global test logic, such as test-mode decode logic and compressed scan codecs, at the top level of the current design. Other test logic types, such as lock-up latches and reconfiguration MUXs, are placed at the local point of use.

You can specify alternative insertion locations for different types of test logic with the `set_dft_location` command:

```
set_dft_location dft_hier_name
  [-include test_logic_types]
  [-exclude test_logic_types]
```

The specified instance name must be a hierarchical cell. It cannot be a library cell, black box, or black-box CTL model.

If the specified hierarchical cell does not exist, the `insert_dft` command creates it during test insertion. For more information, see [Creating New DFT Logic Blocks on page 284](#).

By default, all test logic is synthesized inside the specified hierarchical cell. To synthesize only some types of test logic at that location, use the `-include` or `-exclude` option and list the test logic types to be included in or excluded from the specified cell.

For example, to place DFTMAX codec logic inside `my_cell`, and place the remaining test logic at the top level:

```
dc_shell> set_dft_location my_cell -include CODEC
```

To place all test logic inside `my_cell` except for test control module logic, which remains at its default top-level location:

```
dc_shell> set_dft_location my_cell -exclude TCM
```

The valid test logic types are:

- CODEC

This type includes the compressor and decompressor (codec) logic inserted by the tool for compressed scan, serialized compressed scan, and streaming compressed scan modes.

- PIPELINE_SI_LOGIC

This type includes all head and tail pipelined scan data registers.

- PIPELINE_SE_LOGIC

This type includes all pipelined scan-enable logic.

- PLL

This type includes on-chip clock controller (OCC) and clock chain logic. For more information, see [Chapter 12, On-Chip Clocking Support](#).”

- WRAPPER

This type includes core wrapping cells and wrapper mode logic configured by the `set_wrapper_configuration` command. For more information, see [Chapter 11, Wrapping Cores](#).”

- BSR

This type includes the IEEE Std 1149.1 boundary scan register logic inserted when the `set_dft_configuration -bsd enable` command is used.

- TAP

This type includes the IEEE Std 1149.1 TAP controller logic inserted when the `set_dft_configuration -bsd enable` command is used.

- SERIAL_CNTRL

This type includes the serializer clock controller used in serializer flows.

- SERIAL_REG

This type includes the standalone deserializer and serializer registers used in the serializer IP insertion flow. This type does not affect the normal serializer insertion flow.

- XOR_SELECT

This type includes the sharing compressor and block-select logic inserted at the compressor outputs in the shared codec I/O flow. For more information, see [Sharing Codec Scan I/O Pins on page 773](#).

- TCM

This type includes the test control module logic that decodes test-mode signals in a multiple test-mode flow. For more information about multiple test modes, see [Multiple Test Modes on page 357](#).

- LOCKUP_LATCH

This type includes all lock-up latches.

- RETIMING_FLOP

This type includes all retiming flip-flops.

- REC_MUX

This type includes all scan chain reconfiguration MUXs used to reconfigure scan chains for different test modes in a multiple test-mode flow. It also includes scan-out MUXs, tristate and bidirectional disable logic, and any other glue logic added during DFT insertion.

- IEEE_1500

This type includes all DFT-inserted IEEE 1500 logic used for test-mode control. At the chip level, it also includes the server logic that interfaces to the IEEE Std 1149.1 TAP controller.

When test logic is placed in an alternative location, new test signal pins are created on hierarchical blocks as needed to route the test signals to the specified location. [Table 30](#) lists the possible port types and their naming conventions. If you are moving test logic with many individual signals, such as lock-up latch cells, this can result in a large number of hierarchy pins being created.

Table 30 New Hierarchy Pin Naming Conventions for DFT-Modified Instances

Port	Purpose	Direction
test_si%d	External scan input pin	Input
test_so%d	External scan output pin	Output
test_se	Scan enable (single)	Input
test_se%\$	Scan enables (multiple)	Input

Port	Purpose	Direction
test_mode%d	Test-mode	Input
comp_shift_clk%d	External compression shift clock	Input
comp_lfsr_clk%d	External compression linear feedback shift register (lfsr) clock pins	Input
comp_load_en%d	External compression load pins	Input
comp_unload_en%d	External compression unload pins	Input
scan_in%d	Internal scan-in pins	Output
scan_out%d	Internal scan-out pins	Input
Lockup_latch clock	Name of clock driving the lock-up latch	Input
Functional input net name	Net name where the tool inserts the scan-out MUX	Input
Testmode name_out	Output test-mode pin for a test_mode	Output

If you issue multiple `set_dft_location` commands, the `insert_dft` command uses the last specified location for each test logic type during DFT insertion. In [Example 21](#), reconfiguration MUXs and lock-up latches are kept at their local point of use, test-mode decode logic is placed in a top-level UTCM block, and all other test logic types are placed in a UTEST_LOGIC block.

Example 21 Issuing Multiple set_dft_location Commands

```
set_dft_location -exclude {REC_MUX LOCKUP_LATCH BSR} UTEST_LOGIC
set_dft_location -include {TCM} UTCN      ;# TCM location is overwritten
```

You can use the `report_dft_location` command to see the currently defined locations for all DFT logic types. The default location for global test logic types is reported as `<top>`, which represents the top level of the current design. The default location for local test logic types is reported as `<local>`, which represents the local point of use. [Example 22](#) shows the report resulting from the commands in [Example 21](#).

Example 22 Example of a report_dft_location Report

Design Name	DFT PARTITION NAME	DFT TYPE	DFT Hierarchy Location
top	default_partition	BSR	<top>
top	default_partition	CODEC	UTEST_LOGIC
top	default_partition	LOCKUP_LATCH	<local>
top	default_partition	XOR_SELECT	<top>

top	default_partition	RETIMING_FLOP	<local>
top	default_partition	TAP	UTEST_LOGIC
top	default_partition	TCM	UTCM
top	default_partition	PIPELINE_SE_LOGIC	UTEST_LOGIC
top	default_partition	PIPELINE_SI_LOGIC	UTEST_LOGIC
top	default_partition	PLL	UTEST_LOGIC
top	default_partition	REC_MUX	<local>
top	default_partition	SERIAL_CNTRL	UTEST_LOGIC
top	default_partition	SERIAL_REG	UTEST_LOGIC
top	default_partition	WRAPPER	UTEST_LOGIC
top	default_partition	IEEE_1500	UTEST_LOGIC

You can use the `remove_dft_location` command to remove the location specification for one or more test logic types. When a test logic type has no location specification, it is inserted at its default location.

In a multiple partition flow, only the following test logic types can be specified on a per-partition basis: CODEC, SERIAL_REG, PIPELINE_SI_LOGIC, LOCKUP_LATCH, REC_MUX. For other test logic types, you can only specify the location for the default partition.

The following test logic types are not affected by the `set_dft_location` command:

- AutoFix logic
- Automatically inserted test points
- User-defined test points
- AND gates inserted by the `insert_dft` command to suppress toggling

Creating New DFT Logic Blocks

When using the `set_dft_location` command, if a specified hierarchical cell does not exist, the `set_dft_location` command issues a warning:

```
dc_shell> set_dft_location UCODEC -include {CODEC TCM}
Warning: Specified hierarchy name 'UCODEC' doesn't exist in the current
        design 'test'. (UIT-1112)
Accepted DFT location specification.
1
dc_shell> set_dft_location UOCC -include {PLL}
Warning: Specified hierarchy name 'UOCC' doesn't exist in the current
        design 'test'. (UIT-1112)
Overwriting previous DFT Hierarchy Location specification for type 'PLL'.
Accepted DFT location specification.
1
```

During DFT insertion, the `insert_dft` command creates the hierarchical cells before inserting the DFT logic and issues information messages with the cell and design names of the new DFT blocks:

```
dc_shell> insert_dft
...
Created instance 'UCODEC' of design 'top_dft_design'
Created instance 'UOCC' of design 'top_dft_design_1'
    Architecting Scan Chains
```

You can specify cell instance names with the `set_dft_location` command, but the resulting design names are automatically generated. To use specific design names for the new DFT blocks, use the `rename_design` command after DFT insertion completes. For example,

```
set_dft_location UCODEC -include {CODEC TCM}
set_dft_location UOCC -include {PLL}
insert_dft
rename_design [get_attribute [get_cells UCODEC] ref_name] CODEC_design
rename_design [get_attribute [get_cells UOCC] ref_name] OCC_design
```

Partitioning a Scan Design With DFT Partitions

In some cases, you might want to apply different DFT configuration to different parts of the design. For example, you might want to use different scan-enable signals for different blocks, or you might want to enable clock-mixing for some blocks but not others.

You can use *DFT partitions* to do this. They allow you to divide up your design logic into multiple partitions, then you apply DFT configuration commands to each partition.

The following topics describe how to use DFT partitions:

- [Defining DFT Partitions](#)
- [Configuring DFT Partitions](#)
- [Per-Partition Scan Configuration Commands](#)
- [Known Issues of the DFT Partition Flow](#)

Defining DFT Partitions

To use DFT partitions, you must first define them. You can use the following commands to define and manage partition definitions in your top-level run:

- `define_dft_partition`
- `report_dft_partition`
- `remove_dft_partition`

You can use the `define_dft_partition` command to define a partition. The most commonly used options are:

```
define_dft_partition
    partition_name
    [-include list_of_cells_or_references]
    [-clocks list_of_clocks]
```

You must provide a unique name for each partition definition. This name is used to reference the partition when providing codec information, as well as to identify the partition in subsequent DFT reports.

A partition definition can include design references, hierarchical cells, scan cells, core scan segments, or clock domains. Although partitions are usually defined along physical or logical hierarchy boundaries, it is not a requirement.

To specify a set of cells, design references, or core scan segments, use the `-include` option. Leaf cells can be specified, although only sequential cells are relevant to the partition definition. Design references are converted to the set of all hierarchical instances of those designs. A particular cell or design reference can exist in only one partition definition. In [Example 23](#), two partitions are defined using hierarchical cells, and a third partition is defined using a design reference.

Example 23 Defining Three Partitions Using Cells and References

```
define_dft_partition P1 -include [get_cells {U_SMALL_BLK1 U_SMALL_BLK2}]
define_dft_partition P2 -include [get_cells {U_BIG_BLK}]
define_dft_partition P3 -include [get_references {my_ip_design}]
```

A partition can also be defined to include one or more clock domains with the `-clocks` option. The partition includes all flip-flops clocked by the specified clocks. In [Example 24](#), two partitions are created by clock domain.

Example 24 Defining Partitions by Clock Domain

```
define_dft_partition P1 -clocks {CLK1 CLK2}
define_dft_partition P2 -clocks {CLK3}
```

The tool creates a default partition named `default_partition` that includes the flip-flops not included in any user-defined partitions. You cannot use the name `default_partition` when defining a partition.

You can use the `report_dft_partition` command to see what partitions have been defined. [Example 25](#) shows the output from the `report_dft_partition` command for the three partitions previously defined in [Example 23](#). Note that the design reference has been converted to the corresponding set of hierarchical instances of that design.

Example 25 Example of Output From the report_dft_partition Command

```
Cells or Designs defined in Partition 'P1':
  U_SMALL_BLK1
  U_SMALL_BLK2
Cells or Designs defined in Partition 'P2':
  U_BIG_BLK
Cells or Designs defined in Partition 'P3':
  U_MY_IP_BLK
```

You can use the `remove_dft_partition` command to remove one or more user-defined partition definitions. The default partition cannot be removed.

```
remove_dft_partition {P1 P2 P3}
```

Configuring DFT Partitions

After the DFT partitions are defined, you can configure the scan configuration for each partition. Use the `current_dft_partition` command to set the current partition, then apply one or more supported test configuration commands to configure scan for that partition.

All DFT partitions share a common global configuration. Partition-specific configuration commands are applied incrementally on top of the global configuration.

In a DFT partition flow, the sequence of configuration commands is:

- Apply global DFT configuration settings
- Define DFT partitions with `define_dft_partition`
- Apply partition-specific DFT configuration settings to each partition with `current_dft_partition`

[Example 26](#) shows an example of global and partition-specific configuration commands.

Example 26 Configuring Two DFT Partitions

```
# apply global DFT configuration settings
set_scan_configuration -clock_mixing mix_clocks
set_dft_signal -view existing_dft \
```

```

-type ScanClock -timing [list 45 55] -port CLK
set_dft_signal -view spec -type TestMode -port TM

# define DFT partitions
define_dft_partition P1 -include {BLK1}
define_dft_partition P2 -include {BLK2}

# configure DFT partition P1
current_dft_partition P1
set_dft_signal -view spec -type ScanEnable -port SE1
set_dft_signal -view spec -type ScanDataIn -port {SI1 SI2}
set_dft_signal -view spec -type ScanDataOut -port {SO1 SO2}
set_scan_configuration -chain_count 2

# configure DFT partition P2
current_dft_partition P2
set_dft_signal -view spec -type ScanEnable -port SE2
set_dft_signal -view spec -type ScanDataIn -port {SI3 SI4 SI5}
set_dft_signal -view spec -type ScanDataOut -port {SO3 SO4 SO5}
set_scan_configuration -chain_count 3

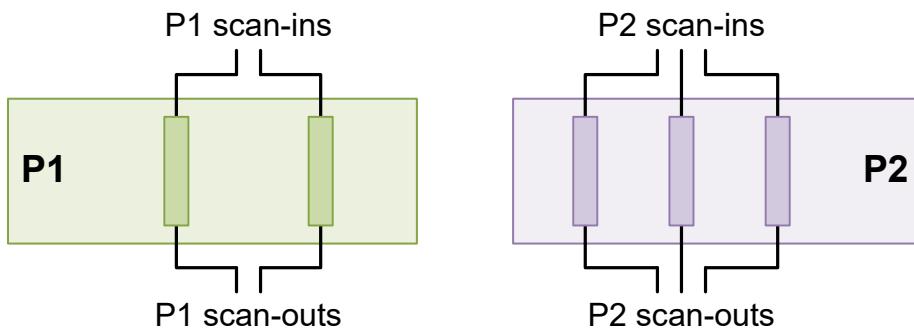
```

If you are defining scan-in or scan-out signals using the `set_dft_signal` command, you must define them as a part of each partition's scan configuration. Scan-enable signals can be defined globally or on a per-partition basis. However, if a scan-enable signal is defined for only one partition, it is automatically applied to the remaining partitions. Other signal types, such as reset, clock, and test-mode signals, must be defined globally before any partitions are defined, or as part of the default partition configuration.

For the entire design, the total scan chain count is the sum of the scan chain counts across all partitions. Each scan chain requires its own scan-in and scan-out pin pair, just as a scan chain does in an unpartitioned flow. Scan chains are not combined or rebalanced across the partitions.

[Figure 106](#) shows the scan mode chain connections for [Example 26](#). A total of five scan-in and scan-out pins are used, two for partition P1 and three for partition P2.

Figure 106 DFT Partition Scan Chains



After configuring the partitions, you can use the `preview_dft` command to display the scan chain distribution across partitions. [Example 27](#) shows a report example.

Example 27 Output From the `preview_dft` Command for Multiple DFT Partitions

```
*****
Current mode: Internal_scan
*****  
  

Number of chains: 5
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix  
  

Scan chain '1' (SI1 --> SO1) contains 32 cells (Partition 'P1')
  Active in modes: Internal_scan  
  

Scan chain '2' (SI2 --> SO2) contains 32 cells (Partition 'P1')
  Active in modes: Internal_scan  
  

Scan chain '3' (SI3 --> SO3) contains 22 cells (Partition 'P2')
  Active in modes: Internal_scan  
  

Scan chain '4' (SI4 --> SO4) contains 21 cells (Partition 'P2')
  Active in modes: Internal_scan  
  

Scan chain '5' (SI5 --> SO5) contains 21 cells (Partition 'P2')
  Active in modes: Internal_scan
```

Per-Partition Scan Configuration Commands

This topic lists the commands you can use to configure DFT insertion on a per-partition basis. Commands not listed in this section should be applied as part of the global DFT configuration.

set_scan_configuration

The following `set_scan_configuration` options can be specified on a per-partition basis:

- `-chain_count`
- `-max_length`
- `-exact_length`
- `-clock_mixing`
- `-insert_terminal_lockup`
- `-test_mode`

- `-exclude_elements`
- `-voltage_mixing`
- `-power_domain_mixing`

set_dft_signal

The following `set_dft_signal` options can be specified on a per-partition basis:

- `-view`
- `-type ScanDataIn | ScanDataOut | ScanEnable | LOSPipelineEnable`
- `-port`
- `-hookup_pin`
- `-hookup_sense`
- `-active_state`

set_dft_location

The following `set_dft_location` test logic types can be specified on a per-partition basis with the `-include` and `-exclude` options:

- `CODEC`
- `SERIAL_REG`
- `PIPELINE_SI_LOGIC`
- `LOCKUP_LATCH`
- `REC_MUX`

For other test logic types, you can only specify the location for the default partition.

set_scan_path

The following `set_scan_path` options can be specified on a per-partition basis:

- `-include_elements`
- `-head_elements`
- `-tail_elements`
- `-ordered_elements`
- `-complete`
- `-exact_length`

- `-scan_master_clock`
- `-scan_slave_clock`
- `-scan_enable`
- `-scan_data_in`
- `-scan_data_out`

set_testability_configuration

The following `set_testability_configuration` options can be specified on a per-partition basis:

- `-clock_signal`
- `-allowed_clock_signal`
- `-disallowed_clock_signal`
- `-control_signal`
- `-test_points_per_scan_cell`
- `-max_test_points`

set_wrapper_configuration

The following `set_wrapper_configuration` options can be specified on a per-partition basis:

- `-chain_count`
- `-max_length`
- `-mix_cells`

Known Issues of the DFT Partition Flow

The following known issues apply to the partition flow:

- The `-include` option of the `set_scan_path` command requires lists of cells and design names. The option does not accept collections.
- If you repeat the same partition name, no error or warning message is issued. The first partition definition is honored and the rest are ignored.
- If you define a scan-enable signal for a partition, that scan-enable signal is reused for any subsequently configured partitions for which the signal has not been defined. This reuse avoids having to create new scan-enable signals.

- The `define_dft_partition` command does not perform duplicate object checking between cell instances and design modules. The overlapping objects will be included only in one of the partitions.
- If you apply commands or options that do not support per-partition specification to a DFT partition, they are ignored with no warning.

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 the `reset_design` command 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
dc_shell> # manually modify scan_arch.tcl to reflect desired architecture
dc_shell> source scan_arch.tcl
dc_shell> preview_dft
dc_shell> insert_dft
```

See Also

- [Previewing the DFT Logic on page 603](#) for more information about previewing scan chain structures

10

Advanced DFT Architecture Methodologies

This chapter describes advanced features that can be used while inserting scan circuitry into your design. These features can be used to improve design testability using manual and automatic techniques, improve the frequency of the scan testing logic, reduce power consumption during test, and provide improved integration of tool-inserted and user-defined test logic.

This chapter describes advanced DFT architecture-related methodologies and processes in the following topics:

- [Inserting Test Points](#)
- [Using AutoFix](#)
- [Using Pipelined Scan Enables for Launch-On-Extra-Shift \(LOES\)](#)
- [Multiple Test Modes](#)
- [Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces](#)
- [Multivoltage Support](#)
- [Controlling Power Modes During Test](#)
- [Reducing Shift Power Using Functional Output Pin Gating](#)
- [Controlling Clock-Gating Cell Test Pin Connections](#)
- [Internal Pins Flow](#)
- [Creating Scan Groups](#)
- [Shift Register Identification](#)
- [Performing Scan Extraction](#)

Inserting Test Points

Test points are points in the design where the TestMAX DFT tool inserts logic to improve the testability of the design. The tool can automatically determine where to insert test

points to improve test coverage and reduce pattern count. You can also manually define where test points are to be inserted.

The test point capabilities are described in the following topics:

- [Test Point Types](#)
- [Test Point Structures](#)
- [Automatically Inserted Test Points](#)
- [User-Defined Test Points](#)
- [Previewing the Test Point Logic](#)
- [Inserting the Test Point Logic](#)

Caution:

To use any functionality in this section, you must enable the `testability` client by using the following command:

```
dc_shell> set_dft_configuration -testability enable
```

Otherwise, the tool provides the legacy test point functionality described in [Appendix B, Legacy Test Point Insertion](#).

Test Point Types

The available test point types are:

- [Force Test Points](#)
- [Control Test Points](#)
- [Observe Test Points](#)
- [Multicycle Test Points](#)

Note:

The test point schematics in these topics show the functional operation of the test points. During synthesis, constant logic is simplified, and the test point logic might be optimized into the surrounding logic.

Force Test Points

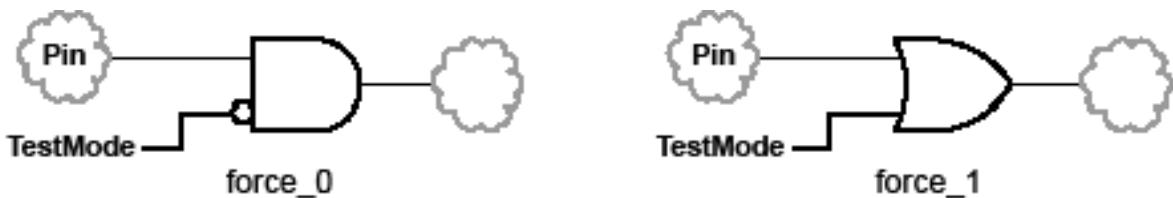
Force test points allow a signal to be overridden (always force) throughout the entire test program. They are typically used to block some other value (such as an X value) from propagating.

The following force test point types are available:

- force_0
- force_1
- force_01

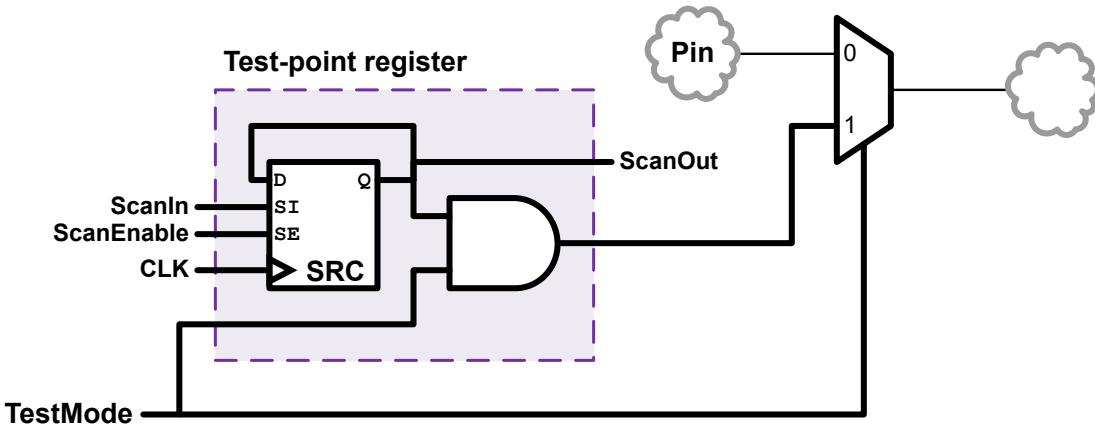
The `force_0` and `force_1` test point types allow a signal to be replaced with a constant 0 or constant 1 value throughout the entire test session. These test point types are useful when a particular signal must be forced to a known value for testability purposes. A logic gate is used to replace the original signal with a fixed constant 0 or 1 value when the `TestMode` signal is asserted. See [Figure 107](#).

Figure 107 Example of a force_0 or force_1 Test Point



The `force_01` test point type allows a signal to be replaced with a scan-selected value throughout the entire test session. A multiplexer is used to replace the original signal with the output of this scan register when the `TestMode` signal is asserted. See [Figure 108](#).

Figure 108 Example of a force_01 Test Point



The forced value can vary per-pattern (as the scan register reloads with each pattern), but it remains constant for all capture cycles within a given pattern.

Control Test Points

Control test points allow a hard-to-control signal to be controllable (selectively forced) for some test vectors but not others. They are typically inserted to increase the fault coverage of the design. They provide some control while still allowing some observation of upstream logic.

The following control test point types are available:

- control_0
- control_1
- control_01

A `control_0` or `control_1` test point is built with a controlling logic gate, an enabling AND gate, and a source scan register. When `TestMode` is not asserted, the signal always retains its original value. When `TestMode` is asserted, the signal is forced with a fixed constant 0 or 1 value only when the output of the scan register selects the constant value. This allows the test program to select either the original signal behavior or the constant-forced behavior on a per-pattern basis. See [Figure 109](#) and [Figure 110](#).

Figure 109 Example of a control_1 Test Point

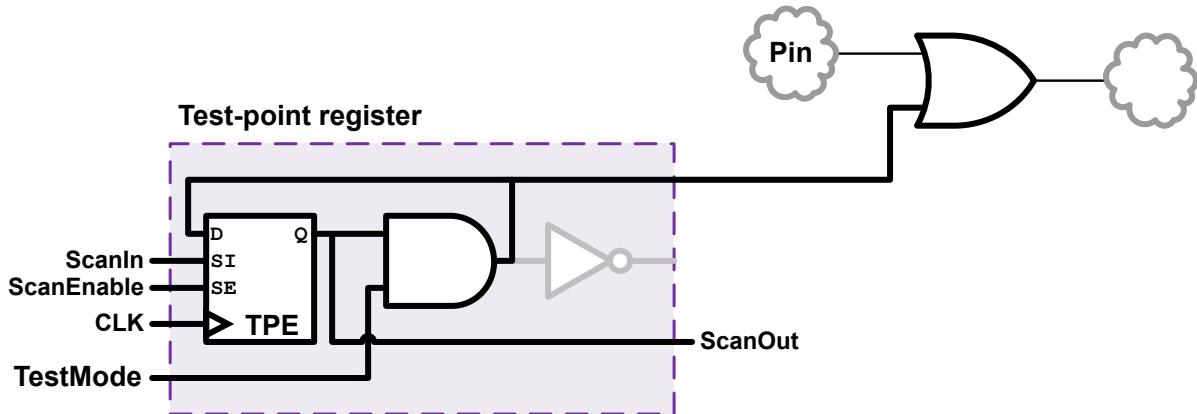
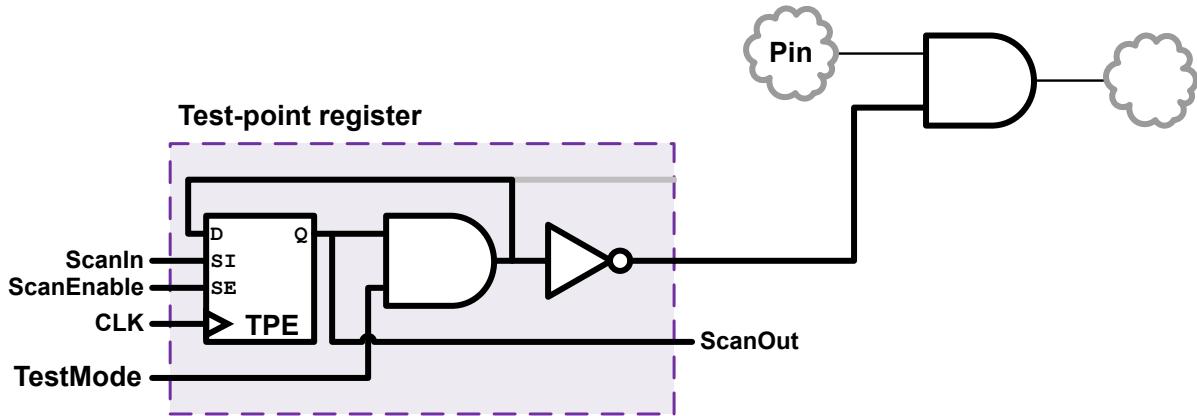
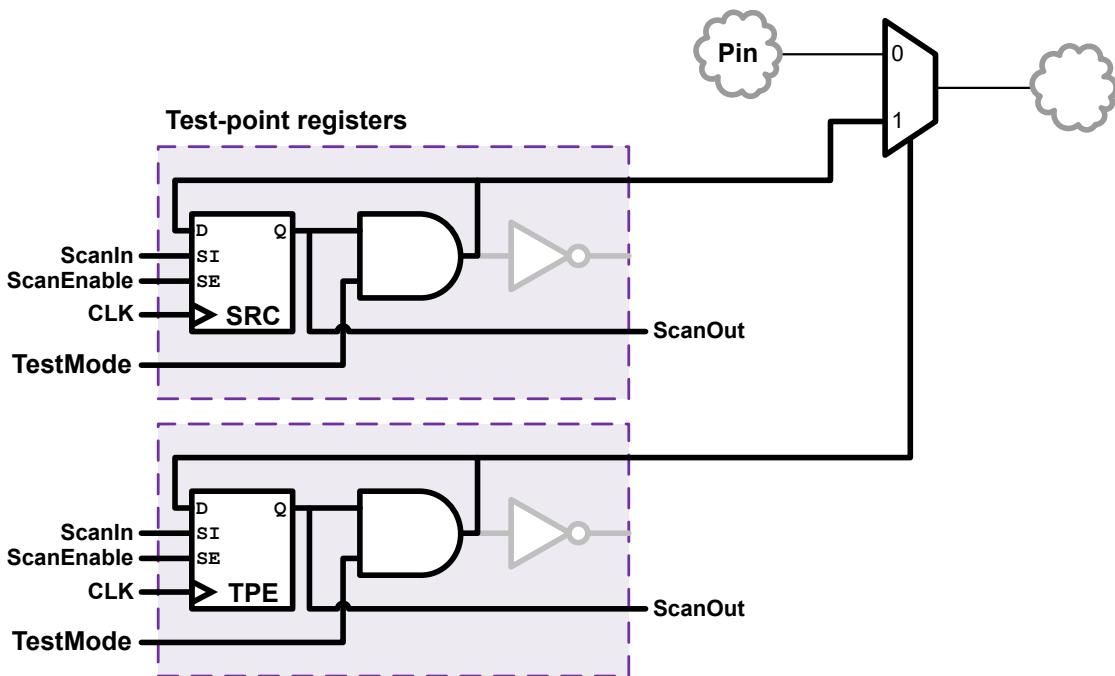


Figure 110 Example of a control_0 Test Point



A control_01 test point is similar to the control_0 and control_1 test point types, except that a scan-selected source signal value from a scan register is selectively driven onto the net on a vector-by-vector basis. As a result, the control_01 test point requires two scan cells per control point, one for the source signal value and one for the enable register that specifies that the source signal should be driven. See Figure 111.

Figure 111 Example of a control_01 Test Point



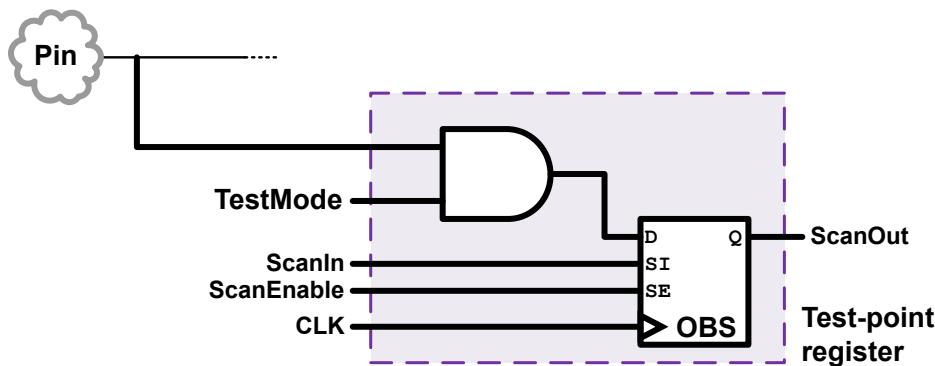
A control point can be enabled or disabled per-pattern, but its assertion behavior remains constant across capture cycles within a given pattern.

Observe Test Points

The `observe` test point type is typically inserted at hard-to-observe signals in a design to reduce test data volume or to increase coverage.

An `observe` test point is a scan register with its data input connected to the signal to be observed. An AND gate prevents the register from capturing and propagating toggle activity when not in use. It also allows transitions to be blocked in particular test modes, such as during at-speed testing. See [Figure 112](#).

Figure 112 Example of an observe Test Point



Multicycle Test Points

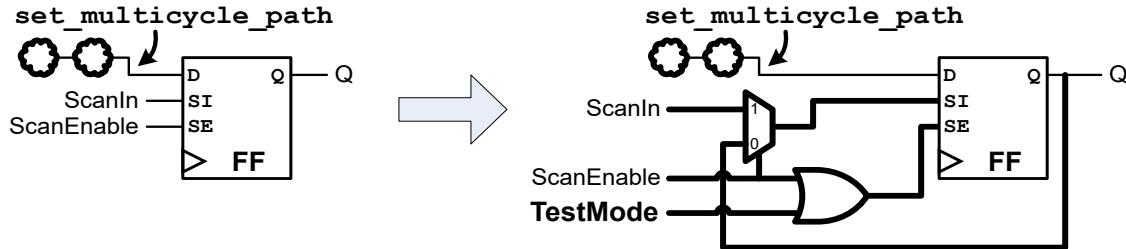
During ATPG, when a scan cell captures a value from a logic path constrained by a multicycle path exception, it captures a dynamic X value because the multicycle logic might not be stable by the capturing clock edge. These captured X values can affect other captured values in compressed designs, and they can propagate through the scan cell into other areas of the design when fast sequential ATPG is used.

The multicycle test point prevents these X values from being captured. It implements the following scan capture behavior:

- When the TestMode signal is asserted, the register holds state instead of capturing.
- When the TestMode signal is deasserted, the register captures normally.

Scan shift operation and functional operation are unaffected. This test point uses reconfigured scan path logic to avoid inserting logic along the functional path. See [Figure 113](#).

Figure 113 Example of a Multicycle Test Point



The multicycle test point does not provide coverage for the blocked capture path. However, it does prevent multicycle X values from propagating into scan compression logic or self-test logic.

Multicycle test points can only be inserted using automatic test point insertion.

Test Point Structures

The following topics describe how test point logic is structured:

- [Test Point Components](#)
- [Test Point Register Clocks](#)
- [Test Point Enable Logic](#)
- [Sharing Test Point Registers](#)

Test Point Components

Test points are constructed from (up to) three primary components:

- **Pin**

The functional pin where the test point is inserted to assert its behavior. This is the pin where the test point is “located at.” Every test point has a corresponding insertion pin.

- **Register**

A scan-controllable register that provides source (driving) or sink (capturing) capability to the test point logic. A single register can be shared by multiple test points. Some test points do not use a register.

- **Control signal**

The TestMode or IbistEnable signal that activates the test point logic.

[Table 31](#) summarizes which components are used by each type of test point.

Table 31 Components Used by Each Type of Test Point

Test point type	Source register?	Sink register?	Control signal?
force_0, force_1			X
force_01	X		X
control_0, control_1	X		X
control_01	X (two registers)		X
observe		X	X
multicycle			X

Note:

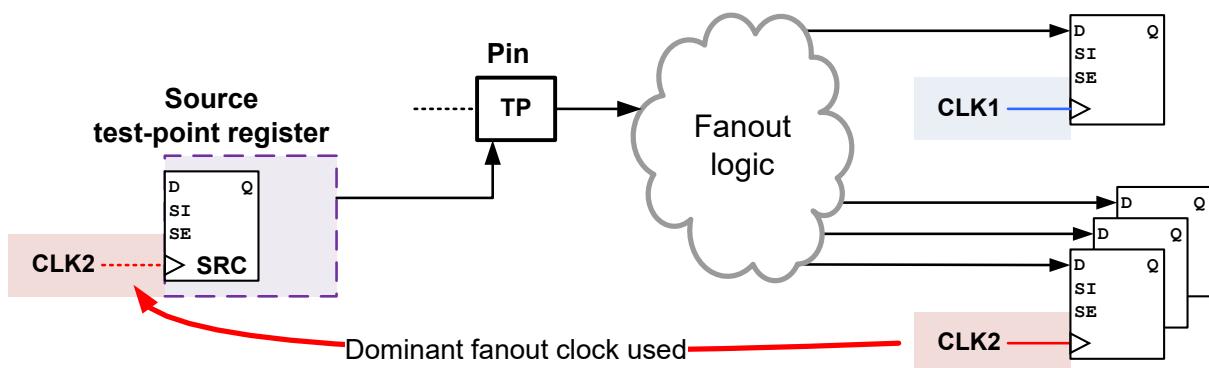
Test point registers do not use a traditional reset *signal*; their value is set by scan shift when used and blocked by the control signal when not used.

Test Point Register Clocks

As described in [Test Point Components on page 299](#), some test point types use a register to drive (source) or capture (sink) data. By default, the tool clocks this register with the same clock as the surrounding logic.

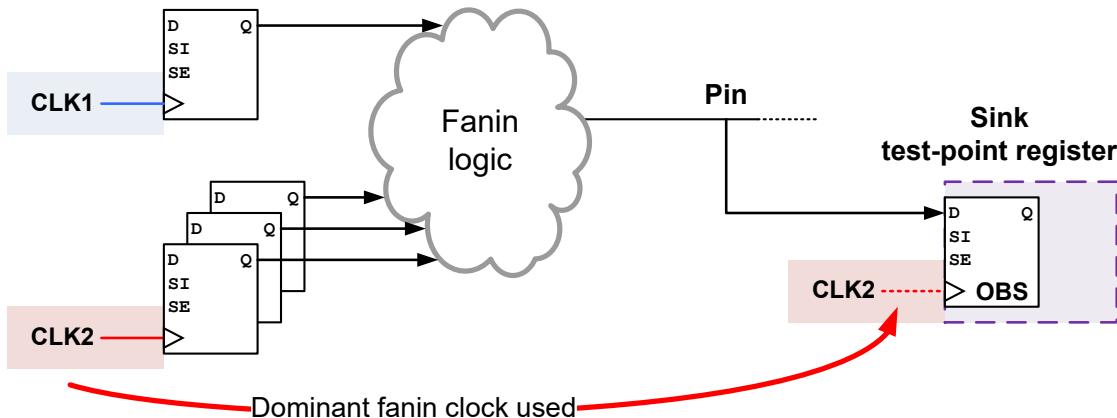
For source registers, the tool uses the dominant clock of the fanout registers, which capture data propagating from the test point:

Figure 114 Using Dominant Fanout Clock



For sink registers, the tool uses the dominant clock of the fanin registers, which drive data that propagates to the test point:

Figure 115 Using Dominant Fanin Clock



Or, you can specify a dedicated test point clock signal to be used for all test point registers:

- You can specify the name of a scan clock signal, defined as a `ScanClock` signal type with the `set_dft_signal` command.
- In a DFT-inserted OCC controller flow, you can specify the name of a PLL output pin. In this case, the tool maps the test point clock to the output pin of the corresponding OCC controller during DFT insertion.
- In a user-defined OCC controller flow, you can directly specify the name of an output pin of an existing OCC controller.

For more information about OCC controller flows, see [Chapter 12, On-Chip Clocking Support](#).

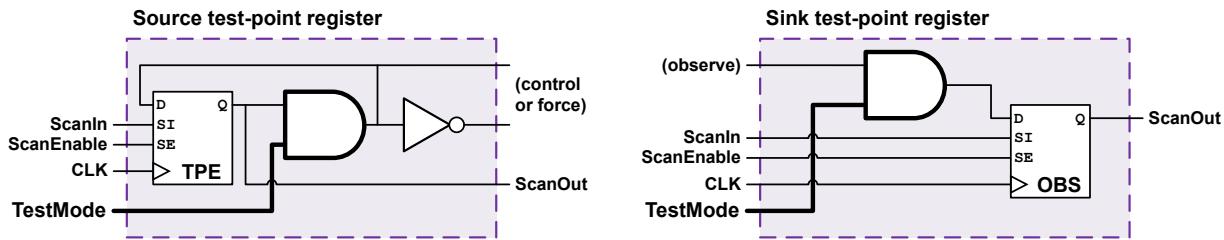
For multivoltage designs, the register is associated with the power domain of the block in which the test point is inserted.

Test Point Enable Logic

When a test point is not enabled, its register output must be held constant.

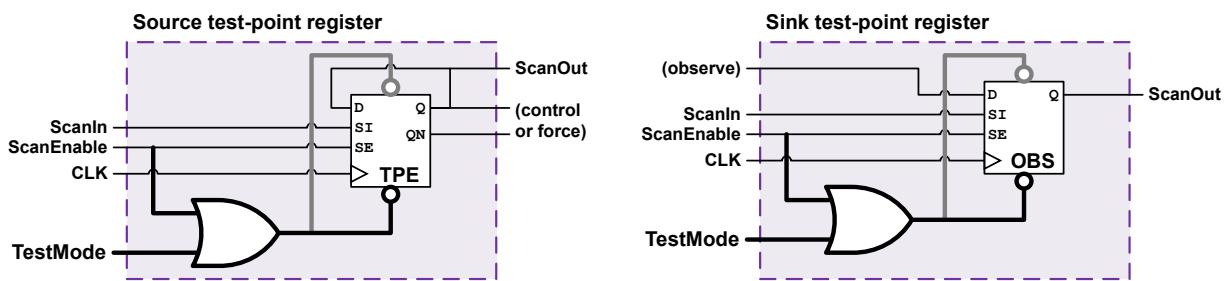
When the `test_tp_enable_logic_type` application variable is set to its default of `gate`, the tool uses a gate-based (AND or OR) enable logic structure to meet this requirement, as shown in [Figure 116](#).

Figure 116 Default Gate-Based Enable Logic Structure



When the `test_tp_enable_logic_type` application variable is set to `reset` or `set`, the tool uses an asynchronous reset- or set-based enable logic structure, as shown in Figure 117.

Figure 117 Reset-Based and Set-Based Enable Logic Structure



The asynchronous enable logic behaves as shown in Table 32.

Table 32 Asynchronous Set/Reset Enable Logic Behavior

TestMode (control signal)	ScanEnable	Register operation
1 (test point enabled)	1	Register scan shifts
1 (test point enabled)	0	Source register holds state, sink register captures
0 (test point disabled)	1	Register scan shifts (in a test mode where the test point is not enabled)
0 (test point disabled)	0	Register “captures” set or reset value in test mode, holds set or reset value in functional mode

Note:

The figures and table in this topic assume active-high DFT signals and active-low asynchronous pins for simplicity. The actual logic implementation considers the signal and pin polarities in your design.

The selected enable logic type is used for all DFT-inserted source and sink test point registers, including the “control+observe” test point used by the `self_gating` testability target.

See Also

- [SolvNet article 2835133, “Why Doesn't a Reset-Based Test Point Register Need a Defined Reset Signal?”](#) for details on how the reset signal is used

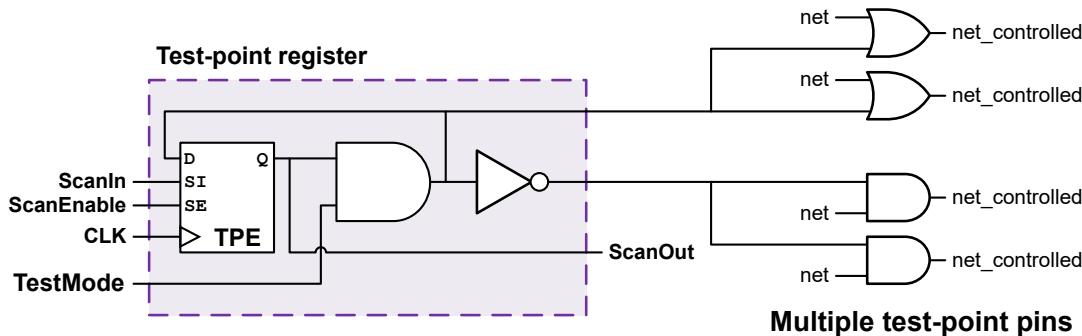
Sharing Test Point Registers

By default, to reduce the area overhead of test point logic, the TestMAX DFT tool shares each test point register with multiple test points.

Sharing Source and Enable Registers

A source or enable test point register can be shared with multiple force or control test point pins. No additional logic gates are required; the register outputs are tied to multiple test point logic gates. [Figure 118](#) shows the logic for multiple `control_0` and `control_1` test points that share the same enable register.

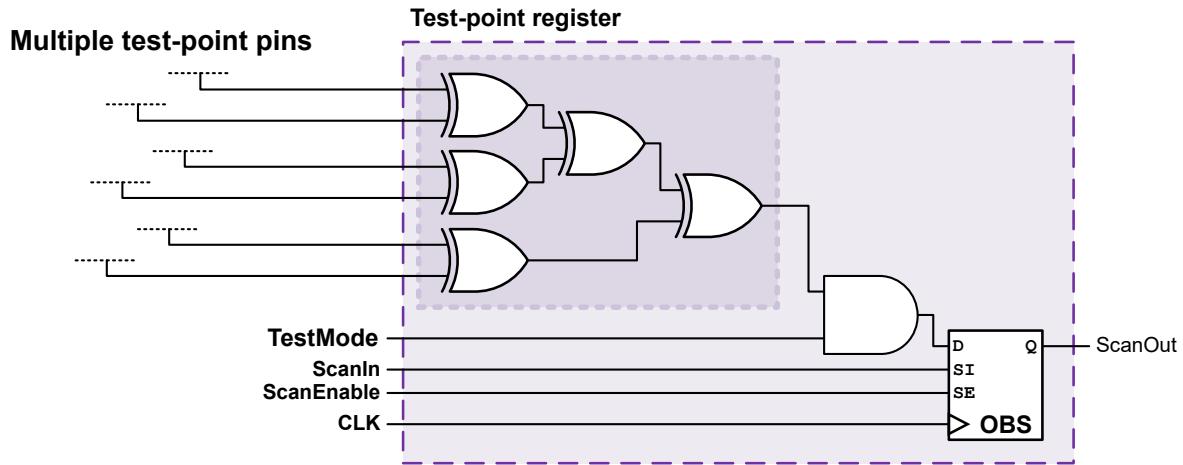
Figure 118 Shared Source Register for Multiple control_0/1 Test Points



Sharing Sink Registers

A sink test point register can be shared with multiple observe test point pins. The tool builds an XOR reduction tree which collapses multiple observed signals down to a single sink signal connected to the AND gate at the data input of the sink register. See [Figure 119](#).

Figure 119 Shared Sink Register For Multiple observe Test Points



Sharing Rules

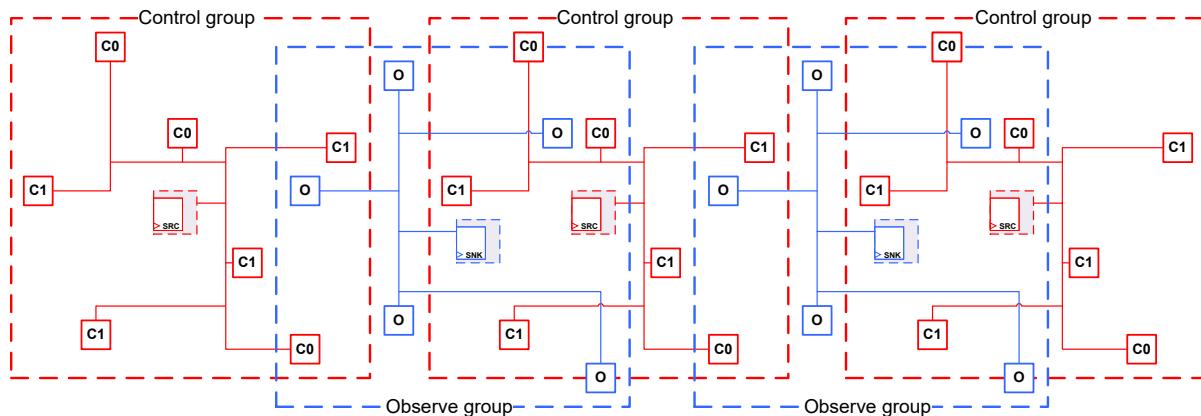
The test point pins within each sharing group share the same test point type, clock domain, and power domain. The pins are chosen to be in close physical or logical proximity. The maximum number of pins in a group is set by the `-test_points_per_scan_cell` option of the relevant test point configuration command. The register is inserted in the lowest hierarchy level common to all pins.

A register cannot be shared as both a source (data) and enable (control) register.

Physical Test Point Grouping

In Design Compiler in topographical mode and in Design Compiler Graphical, the tool groups pins that are in close physical proximity, then creates the test point register within the group. Test point groups of differing type, clock domain, or power domain are created independently and thus might overlap.

Figure 120 Shared Source Registers (Red) and Sink Registers (Blue)



See [SolvNet article 2670826, “Visualizing Test-Point Register Sharing in the Layout View”](#) to display the sharing groups in your own design.

Wire Load Mode Grouping

In wire load mode, compatible pins are sorted in alphanumerical order, then grouped along the sorted list. This method tends to keep pins together with common logical hierarchy, which is correlated (to some degree) to physical proximity.

Automatically Inserted Test Points

You can automatically insert test points in your design to improve its testability. With this feature, the TestMAX DFT tool calls the TestMAX Advisor tool to analyze the design and determine an optimal set of test points, then the TestMAX DFT tool implements them during DFT insertion.

You can use one or more test point *targets*, each focusing on a different aspect of testability:

- `random_resistant`

This target inserts test points that improve random-pattern coverage. This improves the coverage for a given pattern count. It can also improve the maximum coverage obtainable for the design.

- `untestable_logic`

This target inserts test points that make untestable logic testable. This improves the maximum coverage obtainable for a design. It also improves the coverage for a given pattern count.

- `x_blocking`

This target inserts test points to block X values at their sources so they cannot propagate into downstream logic and be captured.

- `multicycle_paths`

This target inserts test points to prevent multicycle-path-constrained logic from being captured by scan cells (which ATPG treats as an X value).

- `shadow_wrapper`

This target inserts *shadow wrappers* around untestable blocks or macrocells so that surrounding logic can be tested. The outputs are forced to known values and the inputs are observed to ensure coverage.

- `core_wrapper`

This target inserts a test-point-based wrapper chain at the data I/O ports of the current design. Only inward-facing (INTEST) functionality is provided, but test point register sharing ensures very low overhead.

- `self_gating`

This target inserts test points that improve the testability of XOR self-gating logic. (For details on this clock-gating feature, see the “XOR Self-Gating” section of the Power Compiler User Guide.)

- `user`

This target inserts test points whose types and locations are provided by the user.

The following topics describe how to configure automatic test point insertion:

- [Enabling Automatic Test Point Insertion](#)
- [Configuring Global Test Point Insertion Settings](#)
- [Configuring the Random-Resistant Test Point Target](#)
- [Configuring the Untestable Logic Test Point Target](#)
- [Configuring the X-Blocking Test Point Target](#)
- [Configuring the Multicycle Path Test Point Target](#)
- [Configuring the Shadow Wrapper Test Point Target](#)
- [Configuring the Core Wrapper Test Point Target](#)
- [Configuring the XOR Self-Gating Test Point Target](#)
- [Configuring the User-Defined Test Point Target](#)

- [Enabling Multiple Targets in a Single Command](#)
- [Implementing Test Points From an External File](#)
- [Customizing the Test Point Analysis](#)
- [Running Test Point Analysis](#)
- [Automatic Test Point Insertion Example Script](#)
- [Limitations](#)

Enabling Automatic Test Point Insertion

To enable automatic test point insertion, issue the following command before pre-DFT DRC:

```
dc_shell> set_dft_configuration -testability enable
```

Note:

A TestMAX_DFT and TestMAX Advisor license, or a DFTMAX license and a Spyglass® DFT ADV license, are required to use the automatic test point insertion feature.

Then, use the `set_testability_configuration` command to configure one or more automatic test point targets:

- To configure global settings, which are shared by all targets, omit the `-target` option.
- To enable a particular target (and to optionally configure any target-specific options it supports), specify that target with the `-target` option.

To enable multiple test point targets, issue a separate configuration command for each target, as shown in [Automatic Test Point Insertion Example Script on page 326](#).

Before previewing the test points with the `preview_dft` command, run test point analysis as described in [Running Test Point Analysis on page 325](#).

Configuring Global Test Point Insertion Settings

To configure global aspects and defaults of automatic test point insertion, use the `set_testability_configuration` command without the `-target` option:

```
set_testability_configuration
  [-clock_signal clock_name]
  [-allowed_clock_signal clock_list]
  [-disallowed_clock_signal clock_list]
  [-control_signal control_name]
  [-only_from_file false | true]
  [-test_points_per_scan_cell n]
  [-sg_command_file file_name]
```

```

[-max_test_points n]
[-test_point_file file_name]
[-effort low | medium | high]
[-target_test_coverage coverage_value]
[-random_pattern_count n]
[-include_elements cell_list]
[-include_fanin_cone pin_port_list]
[-include_fanout_cone pin_port_list]
[-exclude_elements cell_list]
[-exclude_fanin_cone pin_port_list]
[-exclude_fanout_cone pin_port_list]

```

Table 33 shows the global configuration options.

Table 33 Global set_testability_configuration Options

To do this	Use this option
Use a single dedicated clock for all test point registers in the design ⁷	-clock_signal clock_name (default is the dominant clock)
Use a particular TestMode signal to enable the test points ⁷	-control_signal control_name (default is the first available TestMode signal)
Specify the list of a previously defined scan clock signals to use for DFT-inserted test point registers.	-allowed_clock_signal clock_list
Specify the list of a previously defined scan clock signals that you must not use for DFT-inserted test point registers.	-disallowed_clock_signal clock_list
Specify whether file-based test points should augment or replace analysis-based test points. ⁷	-only_from_file false true
Specify the number of force, control, or observe points that can share a test point register ⁷	-test_points_per_scan_cell n (default is 8)
Customize the test point analysis with user-provided TestMAX Advisor Tcl commands	-sg_command_file file_name
Set the maximum number of random_resistant and untestable_logic test points (combined) that can be inserted	-max_test_points n (default is 5% of the total register count)
Implement test points defined in an external file (see Implementing Test Points From an External File on page 321)	-test_point_file file_name

7. This option can also be specified per-target to override the global value.

To do this	Use this option
Control the settings used for per-partition random_resistant analysis (see Configuring the Random-Resistant Test Point Target on page 309)	-effort low medium high -target_test_coverage coverage_value -random_pattern_count n
Restrict test point insertion to only particular hierarchical cells or logic cones	-include_elements cell_list -include_fanin_cone pin_port_list -include_fanout_cone pin_port_list (default is to consider the entire design)
Exclude particular hierarchical cells or logic cones from test point insertion	-exclude_elements cell_list -exclude_fanin_cone pin_port_list -exclude_fanout_cone pin_port_list (default is not to exclude anything)

The `-max_test_points` option applies to the `random_resistant` and `untestable_logic` targets. When both targets are enabled, the analysis automatically allocates the global limit across them. Any per-target `-max_test_points` limits apply in addition to, not in place of, the global limit. For details, see [SolvNet article 3021459, “How Does The -max_test_points Option Work?”](#)

Configuring the Random-Resistant Test Point Target

You can use the `random_resistant` test point target to improve the testability of hard-to-test logic in your design. This improves the coverage for a given pattern count. It can also improve the maximum coverage obtainable and ATPG effectiveness for the design.

The random-resistant target invokes a TestMAX Advisor algorithm that determines an optimal set of test points that improves random-pattern test coverage for a given pattern count. It provides the following benefits:

- High capacity—No random pattern simulation is used; instead, the pattern count is a parameter in a mathematical probabilistic fault-detection analysis.
- Easy to use—The algorithm finds the optimum set of control_0, control_1, and observe test points; there is no need to guess at per-type limits.

To enable and configure the random-resistant target, use the following command:

```
set_testability_configuration
  -target random_resistant
  [-clock_signal clock_name]
  [-control_signal control_name]
  [-allowed_clock_signal clock_list]
  [-disallowed_clock_signal clock_list]
```

```

[-only_from_file false | true]
[-test_points_per_scan_cell n]
[-effort low | medium | high]
[-max_test_points n]
[-target_test_coverage coverage_value]
[-random_pattern_count n]
[-include_elements cell_list]
[-include_fanin_cone pin_port_list]
[-include_fanout_cone pin_port_list]
[-exclude_elements cell_list]
[-exclude_fanin_cone pin_port_list]
[-exclude_fanout_cone pin_port_list]

```

The `-target_random_resistant` option enables the random-resistant target and is required to use it. The remaining options, described in [Table 34](#), can be specified to override their defaults.

Table 34 Random-Resistant set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal clock_name</code> <code>-control_signal control_name</code> <code>-test_points_per_scan_cell n</code>
Control the runtime-versus-accuracy tradeoff of the analysis	<code>-effort low medium high</code> (default is <code>medium</code>)
Specify the list of a previously defined scan clock signals to use for DFT-inserted test point registers.	<code>-allowed_clock_signal clock_list</code>
Specify the list of a previously defined scan clock signals that you must not use for DFT-inserted test point registers.	<code>-disallowed_clock_signal clock_list</code>
Specify whether file-based test points should augment or replace analysis-based test points. ⁸	<code>-only_from_file false true</code>
Set the number of random-resistant test points at which the analysis completes ⁸	<code>-max_test_points n</code> (default is no target-specific limit)
Set the random-pattern test coverage value at which the analysis completes ⁸	<code>-target_test_coverage coverage_value</code> (default is 100)
Specifies the random pattern count used for the analysis	<code>-random_pattern_count n</code> (default is 64000)

8. Random-resistant analysis completes when either of these criteria is met.

To do this	Use this option
Restrict test point insertion to only particular hierarchical cells or logic cones	<pre>-include_elements cell_list -exclude_fanin_cone pin_port_list -exclude_fanout_cone pin_port_list (default is to consider the entire design)</pre>
Exclude particular hierarchical cells or logic cones from test point insertion	<pre>-exclude_elements cell_list -exclude_fanin_cone pin_port_list -exclude_fanout_cone pin_port_list (default is not to exclude anything)</pre>

If you are inserting LogicBIST self-test, for best results, set the `-random_pattern_count` option to the number of self-test patterns.

See Also

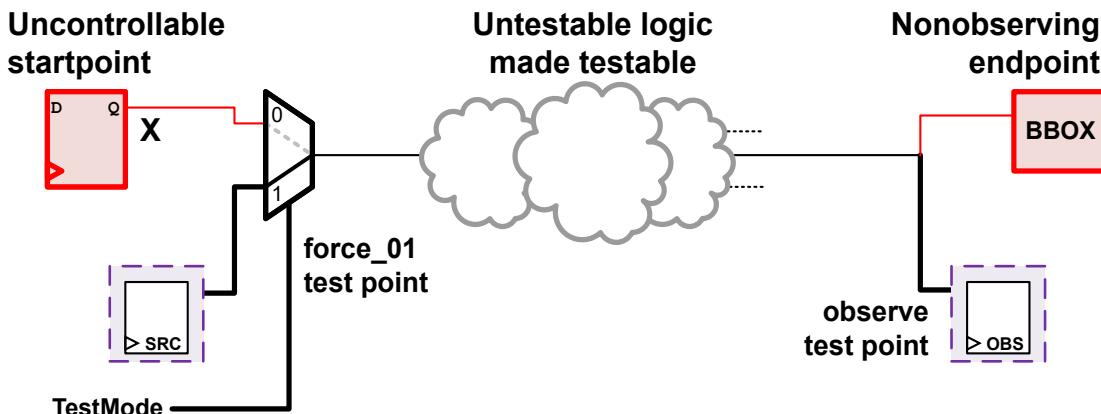
- [Info_random_resistance rule](#) documentation in the TestMAX Advisor documentation for details on the random-resistant analysis algorithm and its parameters

Configuring the Untestable Logic Test Point Target

You can use the `untestable_logic` test point target to make untestable logic testable. This improves the maximum coverage obtainable for a design. It also improves the coverage for a given pattern count.

The untestable logic target invokes a TestMAX Advisor algorithm that determines an optimal set of `force_01` and `observe` test points to control uncontrollable nets and observe unobservable logic, respectively.

Figure 121 Untestable Logic Made Testable



The untestable logic target differs from the X-blocking target as follows:

- It inserts observe test points as well as force_01 test points.
- It performs gain-based analysis, which prefers test points with the highest testability improvement first.
- It considers the `-max_test_points` option.

To enable and configure the untestable logic target, use the following command:

```
set_testability_configuration
  -target untestable_logic
    [-clock_signal clock_name]
    [-control_signal control_name]
    [-allowed_clock_signal clock_list]
    [-disallowed_clock_signal clock_list]
    [-only_from_file false | true]
    [-test_points_per_scan_cell n]
    [-max_test_points n]
    [-include_elements cell_list]
    [-include_fanin_cone pin_port_list]
    [-include_fanout_cone pin_port_list]
    [-exclude_elements cell_list]
    [-exclude_fanin_cone pin_port_list]
    [-exclude_fanout_cone pin_port_list]
```

The `-target untestable_logic` option enables the untestable logic target and is required to use it. The remaining options, described in [Table 35](#), can be specified to override their defaults.

Table 35 Untestable Logic set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal clock_name</code> <code>-control_signal control_name</code> <code>-test_points_per_scan_cell n</code>
Specify the list of a previously defined scan clock signals to use for DFT-inserted test point registers.	<code>-allowed_clock_signal clock_list</code>
Specify the list of a previously defined scan clock signals that you must not use for DFT-inserted test point registers.	<code>-disallowed_clock_signal clock_list</code>
Specify whether file-based test points should augment or replace analysis-based test points. ¹	<code>-only_from_file false true</code>

To do this	Use this option
Set the number of untestable logic test points at which the analysis completes	<code>-max_test_points n</code> (default is no target-specific limit)
Restrict test point insertion to only particular hierarchical cells or logic cones	<code>-include_elements cell_list</code> <code>-include_fanin_cone pin_port_list</code> <code>-include_fanout_cone pin_port_list</code> (default is to consider the entire design)
Exclude particular hierarchical cells or logic cones from test point insertion	<code>-exclude_elements cell_list</code> <code>-exclude_fanin_cone pin_port_list</code> <code>-exclude_fanout_cone pin_port_list</code> (default is not to exclude anything)

This target is highly recommended for designs with LogicBIST self-test.

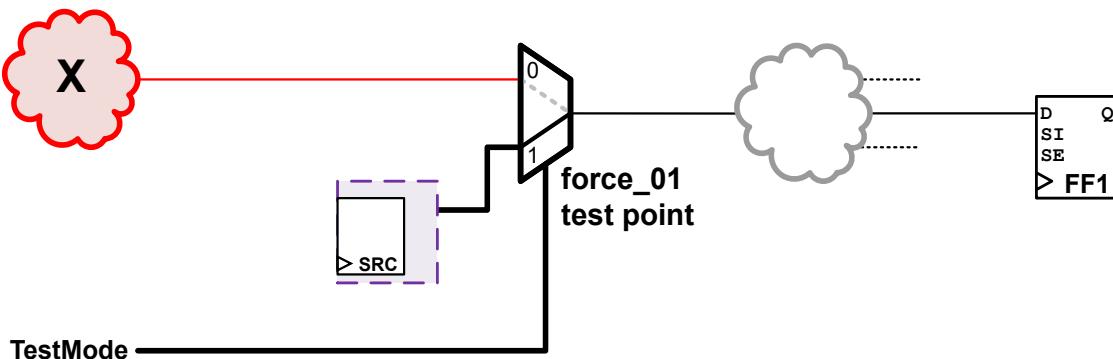
See Also

- [TA_10 rule documentation](#) in the TestMAX Advisor documentation for details on the untestable logic analysis algorithm

Configuring the X-Blocking Test Point Target

You can use the `x_blocking` test point target to identify and block X-value sources from black-box cells. This target inserts `force_01` test points at the sources to force scan-controllable values in place of the X values.

Figure 122 X-Value Source Blocked by `force_01` Test Point



The X-blocking target differs from the untestable logic target as follows:

- It blocks all X sources in the design regardless of gain improvement, which is important for designs with LogicBIST self-test.
- It does not consider the `-max_test_points` option.

To enable the X-blocking target, use the following command:

```
set_testability_configuration
  -target x_blocking
    [-clock_signal clock_name]
    [-control_signal control_name]
    [-test_points_per_scan_cell n]
```

The `-target x_blocking` option enables the X-blocking target and is required to use it. The remaining options, described in [Table 36](#), can be specified to override their defaults.

Table 36 X-Blocking set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal clock_name</code> <code>-control_signal control_name</code> <code>-test_points_per_scan_cell n</code>

This target blocks X values from black-box cells. It does not block values from registers with uncontrolled clock or asynchronous set/reset signals.

This target is highly recommended for designs with LogicBIST self-test.

Configuring the Multicycle Path Test Point Target

You can use the `multicycle_paths` test point target to prevent multicycle-path-constrained logic from being captured by scan cells (which ATPG treats as an X value). This target inserts multicycle path test points at multicycle-constrained data input pins of scan cells.

To enable the multicycle paths target, use the following command:

```
set_testability_configuration
  -target multicycle_paths
    [-control_signal control_name]
```

The `-target multicycle_paths` option enables the multicycle paths target and is required to use it. The remaining options, described in [Table 37](#), can be specified to override their defaults.

Table 37 Multicycle Paths set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of the parameter	<code>-control_signal control_name</code>

You must apply all multicycle constraints before test point analysis is run. See [Previewing the Test Point Logic on page 330](#).

See Also

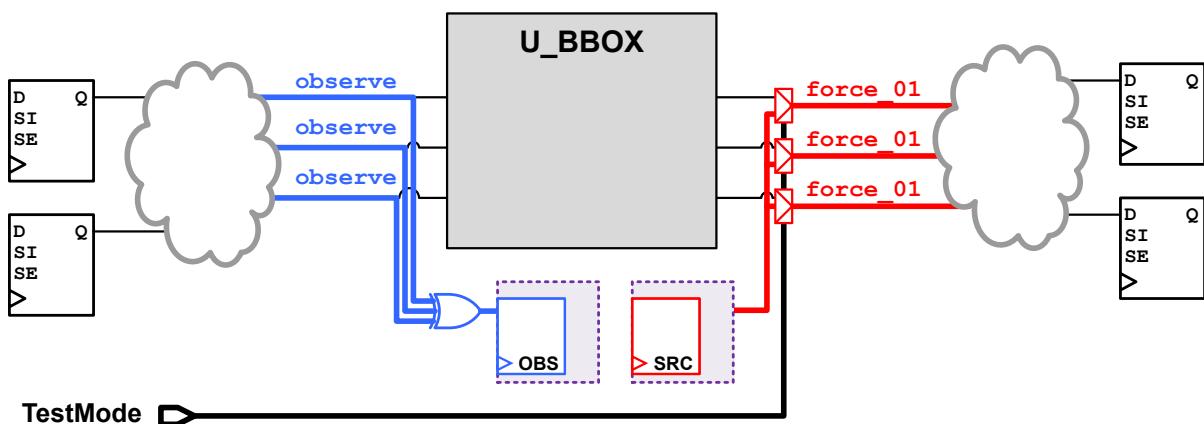
- [Atspeed_05 rule](#) documentation in the TestMAX Advisor documentation for details on the multicycle paths analysis rule
- [Multicycle Test Points on page 298](#) for information on the multicycle path test point

Configuring the Shadow Wrapper Test Point Target

You can use the `shadow_wrapper` test point target to allow logic around untestable blocks or macrocells to be tested.

This target inserts force_01 test points at data output pins to drive known values, and it inserts observe test points at data input pins to ensure coverage. If the functional logic around a pin already allows for testability, the analysis detects this and does not insert a test point.

Figure 123 Shadow Wrapper Around a Black-Box Cell



To enable the shadow wrapper target, use the following command:

```
set_testability_configuration
  -target shadow_wrapper
  -isolate_elements cell_list
```

```
[-clock_signal clock_name]
[-control_signal control_name]
[-test_points_per_scan_cell n]
```

The `-target shadow_wrapper` option enables the shadow wrapper target and the `-isolate_elements` option specifies the list of cells to isolate; both are required. The remaining options, described in [Table 38](#), can be specified to override their defaults.

Table 38 Shadow Wrapper set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal <i>clock_name</i></code> <code>-control_signal <i>control_name</i></code> <code>-test_points_per_scan_cell <i>n</i></code>

Only hierarchical, black-box, and CTL-modeled cells can be isolated.

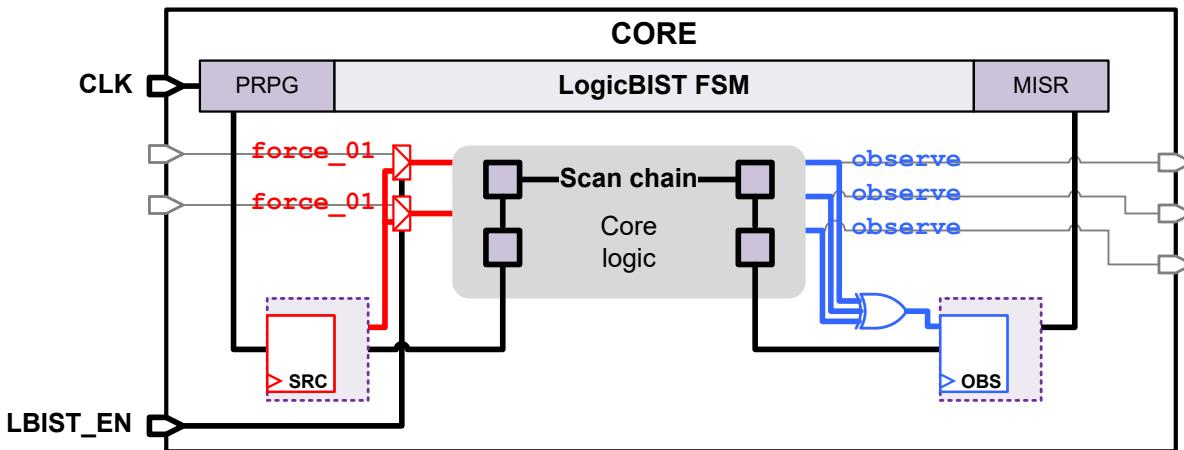
Configuring the Core Wrapper Test Point Target

You can use the `core_wrapper` test point target to insert an inward-facing-only wrapper chain, constructed using test points, at the data I/O ports of the current design.

This target inserts force_01 test points at data input ports to drive known values, and it inserts observe test points at data output ports to ensure coverage. If the functional logic around a port already allows for testability, the analysis detects this and reuses the functional registers instead of inserting a test point.

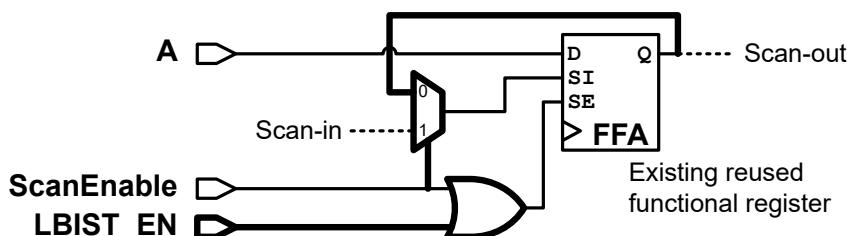
This is not a replacement for full core wrapping in a hierarchical test flow. Instead, it provides a lightweight core isolation capability suitable for unwrapped designs with built-in self-test (BIST). This isolation method is area-efficient, especially when reused registers are used.

Figure 124 Test-Point-Based Wrapper Chain and LogicBIST Self-Test Logic



If an input port is directly registered, its functional register is reused to implement the force_01 test point by modifying the scan-enable connection to hold state during capture, as shown in Figure 125. Similarly, functional output registers can take the place of observe test points. Small amounts of logic can be permitted between ports and their registers by adjusting fanout and depth threshold values.

Figure 125 Functional Input Register Reused as State-Holding force_01 Test Point



Just as with the full core-wrapping DFT client, clock, test, and asynchronous set/reset ports are not wrapped. See [Wrapper Cells and Wrapper Chains on page 442](#) for details.

To enable this lightweight core wrapper target, use the following command:

```
set_testability_configuration
  -target core_wrapper
  [-clock_signal clock_name]
  [-control_signal control_name]
  [-test_points_per_scan_cell n]
  [-reuse_threshold threshold_value]
  [-depth_threshold threshold_value]
```

The `-target core_wrapper` option enables the core wrapper target and is required to use it. The remaining options, described in [Table 39](#), can be specified to override their

defaults. They work the same as described in [The Maximized Reuse Core Wrapping Flow on page 452](#).

Table 39 Core Wrapper set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal clock_name</code> <code>-control_signal control_name</code> <code>-test_points_per_scan_cell n</code>
Specify the maximum number of functional I/O registers allowed for reuse before adding a test point	<code>-reuse_threshold threshold_value</code> (default is 0)
Set the maximum number of combinational logic levels (including buffers and inverters) allowed for reuse before adding a test point	<code>-depth_threshold threshold_value</code> (default is 1)
Exclude particular ports from being wrapped	<code>-exclude_elements port_list</code> (default is not to exclude ports)

Configuring the XOR Self-Gating Test Point Target

You can use the `self_gating` test point target to improve the testability of XOR self-gating logic. (For details on this clock-gating feature, see the “XOR Self-Gating” section of the Power Compiler User Guide.)

XOR self-gating logic is inherently difficult to test. False-positive comparator faults are difficult to test, while false-negative comparator faults are redundant and impossible to test. The next-state values are often driven by complex datapath logic that is difficult to control.

This target improves XOR self-gating testability by inserting the following:

- Per-comparator `control_0` test points

These allow comparators to be masked so that individual comparators can be tested. Each test point reuses the functional register from the neighboring bit.

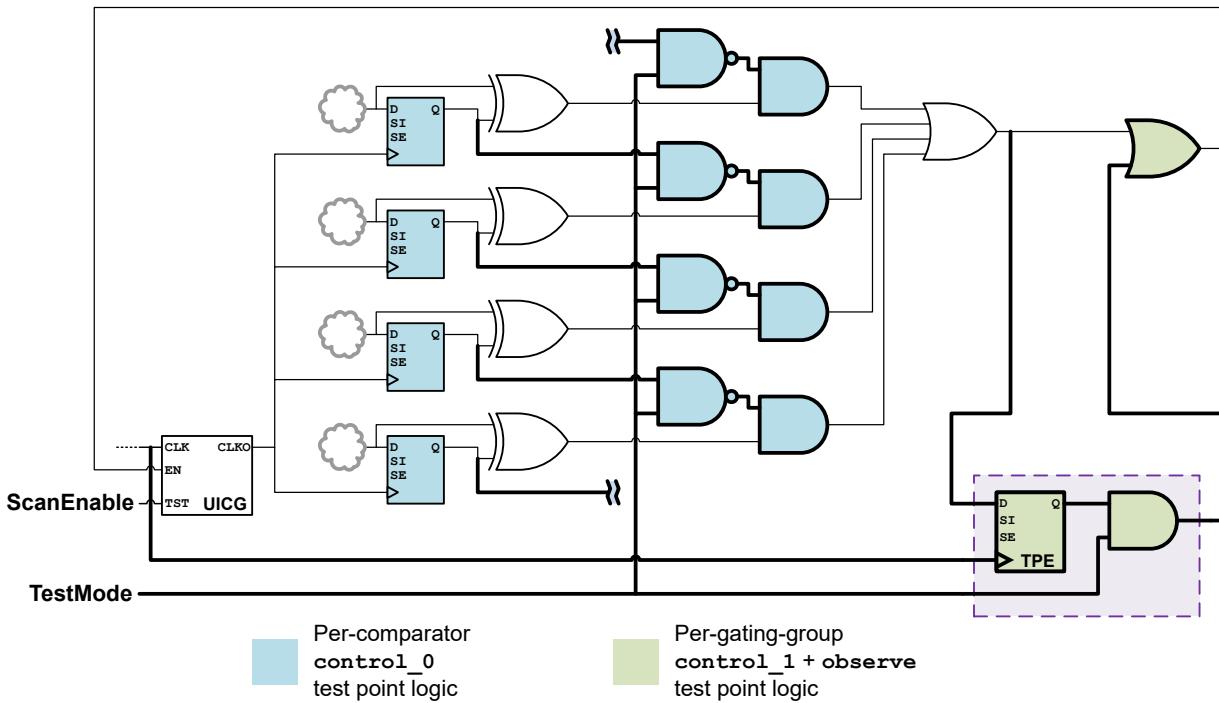
- Per-gating-group `observe` test point

This allows the clock-gating cell enable signal to be directly tested.

- Per-gating-group `control_1` test point

This allows the next-state values to be captured and tested independently of the self-gating logic. It shares its register with the `observe` test point.

Figure 126 XOR Self-Gating Test Points



For self-gating groups with three or fewer registers, the per-comparator `control_0` test points are not inserted.

To enable the XOR self-gating target, use the following command:

```
set_testability_configuration
  -target self_gating
  [-clock_signal clock_name]
  [-control_signal control_name]
```

The `-target self_gating` option enables the self-gating target and is required to use it. The remaining options, described in [Table 40](#), can be specified to override their defaults.

Table 40 XOR Self-Gating set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal clock_name</code> <code>-control_signal control_name</code>

Configuring the User-Defined Test Point Target

The `user` test point target controls the implementation of user-defined test points.

This target applies to the following:

- User-defined test points specified by the `-test_point_file` option

Test points not from TestMAX Advisor in an external file are assigned to the `user` test point target. User-defined test points from an external file are not implemented unless the `user` target is enabled.

- The `set_test_point_element` command

User-defined test points defined with the `set_test_point_element` command are always implemented, whether the `user` target is enabled or not. However, the `user` target allows you to specify implementation defaults for them.

To enable the user-defined target, use the following command:

```
set_testability_configuration
  -target user
  [-clock_signal clock_name]
  [-control_signal control_name]
  [-test_points_per_scan_cell n]
```

The `-target user` option enables the user-defined target and is required to use test points from an external file. The remaining options, described in [Table 41](#), can be specified to override their defaults.

Table 41 User-Defined set_testability_configuration Options

To do this	Use this option
Override the corresponding global value of a parameter	<code>-clock_signal clock_name</code> <code>-control_signal control_name</code> <code>-test_points_per_scan_cell n</code>

See Also

- [Implementing Test Points From an External File on page 321](#) for details on TestMAX Advisor and user-defined test point files

Enabling Multiple Targets in a Single Command

You can enable multiple targets in a single `set_testability_command` by specifying a list with the `-target` option:

```
dc_shell> set_testability_configuration \
  -target {random_resistant x_blocking}
Information: Creating testability configuration for target
  'random_resistant'.
Information: Creating testability configuration for target 'x_blocking'.
```

```
Accepted testability configuration specification for design 'top'.
1
```

If you specify an option that pertains to some targets but not others, the tool warns of unsupported target option combinations (but accepts the valid combinations):

```
dc_shell> set_testability_configuration \
    -target {random_resistant x_blocking} -max_test_points 100
Warning: The '-max_test_points' option does not apply to the
'x_blocking' target. (UIT-1810)
Information: Creating testability configuration for target
'random_resistant'.
Information: Creating testability configuration for target 'x_blocking'.
Accepted testability configuration specification for design 'top'.
1
```

For suggestions on resolving the UIT-1810 warning message, see its man page.

Implementing Test Points From an External File

You can implement test points defined in an external file by using the `-test_point_file` option of the `set_testability_configuration` command:

```
set_testability_configuration
  -test_point_file file_name
```

The `-test_point_file` option is a global option that specifies the name of the test point file to implement.

The test point file must be formatted as follows:

- A test point is described by a line containing three fields, separated by whitespace:
 - A user-defined label (not used by the tool; can be any string)
 - The test point type keyword
 - The net where the test point should be inserted
- For net names, the hierarchy separator character can be “/” (used by the synthesis tools) or “.” (used by the TestMAX Advisor tool).
- Text after “#” is a comment and is ignored.
- Blank lines are ignored.

Note that file-based test points are not implemented unless their targets are enabled (just as with analysis-based test points). Test points are assigned to targets as described in the following sections.

Using User-Defined Test Point Files

Test points from user-defined test point files are assigned to the `user` test point target.

[Example 28](#) shows an example user-defined test point file.

Example 28 Example User-Defined Test Point File

```
# Put these memories into test mode
* force_1 MEM0_0/TSTMODE
* force_1 MEM0_1/TSTMODE
* force_1 MEM1_0/TSTMODE
* force_1 MEM1_1/TSTMODE

# these test points come from our in-house Perl script
TP1 control_1 core0/U37641/z      # my_algorithm.pl result: Q=478 V=479
TP2 control_0 core4/U73087/z      # my_algorithm.pl result: Q=861 V=22
TP3 control_0 core0/U64599/z      # my_algorithm.pl result: Q=227 V=964
TP4 control_1 core4/U99749/z      # my_algorithm.pl result: Q=841 V=9
```

Using TestMAX Advisor Test Point Files

Test points from a TestMAX Advisor test point file (with in-line comments) are routed to their corresponding targets. [Example 29](#) shows an example test point file generated by the TestMAX Advisor tool that contains `random_resistant` and `x_blocking` test points.

Example 29 Example TestMAX Advisor Test Point File

```
# Moment : "TP analysis start" # Time : 2017- 1-30 9:52:19
# Design : "top"
# Initial Random Pattern Test Coverage : 80.68602
# Stuck At Test Coverage : 99.44180
# Random Pattern Count : 10
# Effort Level : medium
# Requested Test Points : 1
# Cut-Off Gain : 0.00000, (cumulative gain of last '1' test points)
# Thread Count : 8

# Index Test_Point_Type Net Comment
1 observe sub3.N50 #Gain : 0.04860 Cov : 80.73462 S@TC : 99.4
# Search completed : 1 (dft_rrf_tp_count(1 - 1)) test points identified.
# BENCHMARK : "COMPLETE TP ANALYSIS" # Time : 5.7086
# Moment : "TP analysis End" # Time : 2017- 1-30 9:52:24

# X Source Section
# Index force_01 net
* force_01 sub2.nZ[0] # -xsource -cell_name sub2.BBOX
* force_01 sub2.nZ[1] # -xsource -cell_name sub2.BBOX
* force_01 sub2.nZ[2] # -xsource -cell_name sub2.BBOX
* force_01 sub2.nZ[3] # -xsource -cell_name sub2.BBOX
```

Using TestMAX Advisor Test Point Files Without Rerunning Analysis

By default, analysis is performed for all enabled targets. However, when implementing test points from a TestMAX Advisor file, you can prevent analysis from rerunning for its targets by specifying the `-only_from_file true` option for those targets. Analysis is still performed for any targets enabled without this option.

The following command implements `random_resistant` and `x_blocking` test points from a TestMAX Advisor file, then also implements shadow-wrapper test points that are not described in the file:

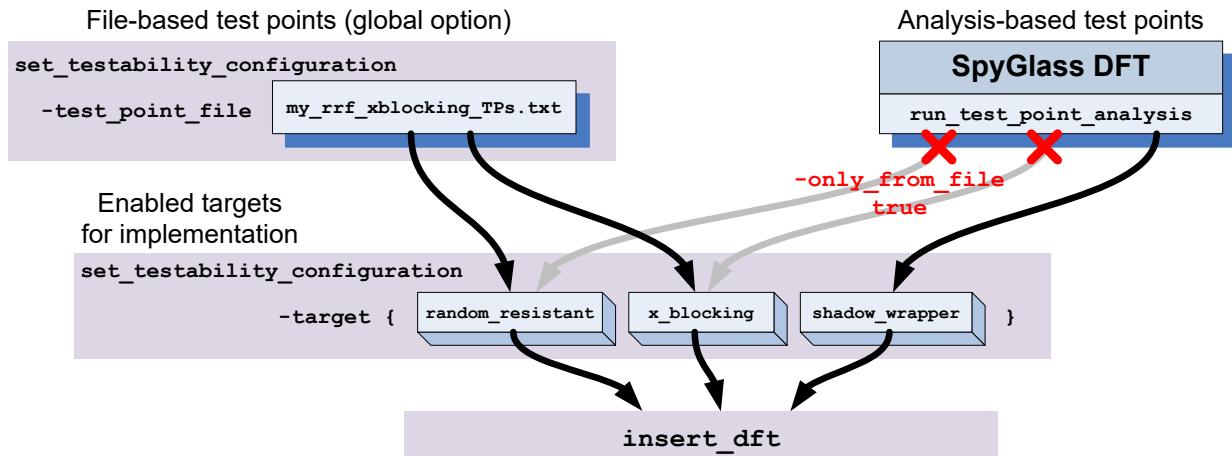
```
# global options - specify test point file
set_testability_configuration -test_point_file my_rrf_xblocking_TPs.txt

# enable targets for implementation
## implement these from the existing TestMAX Advisor file:
set_testability_configuration -target {random_resistant} \
    -only_from_file true
set_testability_configuration -target {x_blocking} \
    -only_from_file true

## derive these in the current run (during run_test_point_analysis)
set_testability_configuration -target {shadow_wrapper} \
    -isolate_elements {IP_BLOCK1 IP_BLOCK2}
```

[Figure 127](#) shows how the test points are populated in each target for this example.

[Figure 127](#) Combining File-Based and Analysis-Based Test Points



If you implement TestMAX Advisor test points from a file by enabling its target without specifying the `-only_from_file true` option, both the file-based and analysis-based test points are implemented for that target.

Customizing the Test Point Analysis

To customize the test point analysis, you can provide a user file containing one or more TestMAX Advisor Tcl commands to be included:

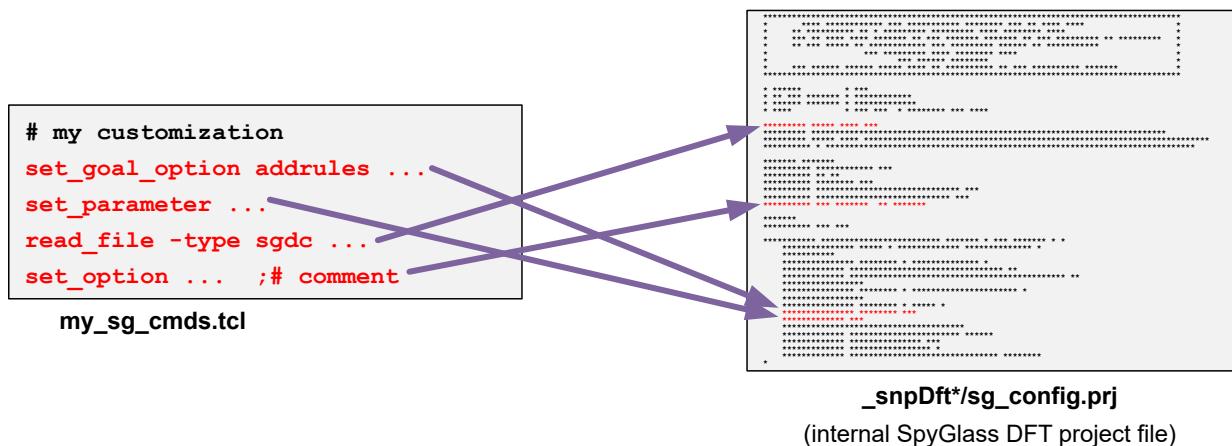
```
dc_shell> set_testability_configuration -sg_command_file my_sg_cmds.tcl
```

This user file is an unordered list of one or more of the following commands:

```
read_file
set_parameter
set_option
set_goal_option
```

The `run_test_point_analysis` command automatically places the commands at the appropriate points in the internal analysis script (also known as the TestMAX Advisor project file):

Figure 128 Including User Commands in the TestMAX Advisor Test Point Analysis



Note:

TestMAX Advisor design constraint (*.sgdc) commands cannot be placed directly in the user file, but they can be placed in a separate file and applied by a `read_file -type sgdc` command in the user file.

Comments after commands are preserved; full-line comments are not.

After the `run_test_point_analysis` command completes, you can examine the TestMAX Advisor project file by looking at the `sg_config.prj` file in the newly created `_snpDft*` directory.

Running Test Point Analysis

If you have configured testability analysis features that require TestMAX Advisor analysis, you must run the analysis after pre-DFT DRC and prior to previewing the test point implementation. To do this, use the `run_test_point_analysis` command:

```
create_test_protocol
dft_drc
run_test_point_analysis
preview_dft -test_points all
```

The `run_test_point_analysis` command requires that the `SPYGLASS_HOME` environmental variable (not Tcl application variable) be set so it can find the `spyglass` executable. You can check this as follows:

```
dc_shell> echo $env(SPYGLASS_HOME)
/global/apps/spyglass_2017.09-SP2
```

Caution:

You must use Spyglass version N-2017.12-SP2 or later.

When you run the `run_test_point_analysis` command, it automatically reports the status of the TestMAX Advisor analysis:

```
dc_shell> run_test_point_analysis
Information: Starting test point analysis.
Information: Test point analysis directory is '/proj/chip/_snpDft_user3.25145.0'.
Information: SpyGlass run started at 09:06:21 AM on May 01 2018
Information: SpyGlass Predictive Analyzer(R) - Version SpyGlass_v0-2019.06

Estimated stuck-at test coverage: 0.73%
Global test point limit of 20 specified.

Analysis for 'untestable_logic' target started.
No per-target limit specified.
4 test points found with estimated test coverage: 64.52%
4 test points found.
Analysis for 'untestable_logic' target completed.

Analysis for 'random_resistant' target started.
No per-target limit specified.
Estimated random pattern test coverage: 78.50%
1 test points found with estimated random pattern coverage: 84.57%
2 test points found with estimated random pattern coverage: 87.82%
3 test points found with estimated random pattern coverage: 88.05%
...
15 test points found with estimated random pattern coverage: 91.31%
16 test points found with estimated random pattern coverage: 91.33%
16 test points found.
Analysis for 'random_resistant' target completed.

SpyGlass Message Summary:
Reported Messages: 0 Fatal, 0 Errors, 0 Warnings, 23 Infos

SpyGlass-DFT Technology Summary:
```

```
Random pattern fault coverage = 91.2%
Random pattern test coverage = 91.3%
Stuck-at fault coverage = 64.5%
Stuck-at test coverage = 64.5%
Percentage of scannable flops = 99.7%
```

Information: SpyGlass critical reports for the current run are present in directory './sg_config/consolidated_reports/top_sg_dft_testpoint_analysis/'.

Information: SpyGlass run completed at 09:09:00 AM on May 01 2018
 Information: Test point analysis completed.

1

The following targets do not require analysis (although doing so is harmless):

- The core_wrapper target
- The user target
- Any target that has the -only_from_file true option set

Automatic Test Point Insertion Example Script

The following script inserts several kinds of testability test points, using a test-mode control signal named TM_TESTPOINTS.

```
# define DFT signals
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing [list 45 55]
set_dft_signal -view spec -type TestMode \
    -port TM_TESTPOINTS

# enable automatic test point insertion
set_dft_configuration -testability enable

# configure global automatic test point insertion settings
# (shared by all targets)
set_testability_configuration \
    -control_signal TM_TESTPOINTS \
    -test_points_per_scan_cell 16

# enable and configure test point targets
set_testability_configuration \
    -target random_resistant \
    -random_pattern_count 1024

set_testability_configuration \
    -target x_blocking

set_testability_configuration \
    -target shadow_wrappers \
    -isolate_elements {IP_CORE1 IP_CORE2}

# preview test points
```

```
create_test_protocol
dft_drc
run_test_point_analysis ;# runs TestMAX Advisor to compute test points
preview_dft -test_points all

# insert DFT logic
insert_dft
```

Limitations

Automatic test point insertion has the following limitations:

- For targets that require TestMAX Advisor analysis, you must use Spyglass version N-2017.12-SP2 or later.
- The `-control_signal` option *must* be specified, otherwise the control signal for the test point logic is tied to logic 0.
- The `shadow_wrapper` target supports only leaf cells (such as macro cells), not hierarchical cells.
- The `multiplex_paths` target does not process bused endpoints unless the following command is run prior to pre-DFT DRC and test point analysis:

```
dc_shell> change_names -rules verilog -hierarchy
```

- Test point registers are always positive-edge, even if the dominant clocking around a test point is negative-edge.
- Testability analysis does not consider the presence of user-defined test points.
- Testability analysis does not consider `set_scan_element false` specifications.
- The `report_testability_configuration` command does not show the inherited `global -test_points_per_scan_cell` value for targets.
- The `reset_testability_configuration` command is not supported.
- File names provided to the `-sg_command_file` option must be absolute. (You can use the Tcl `file normalize` command for this.) You cannot use paths relative to the current directory.
- AutoFix is not supported with the new test point infrastructure.
- Spurious TEST-394 warnings are issued for test point blocks.

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.

Note:

A DFTMAX or TestMAX DFT license is required to use the user-defined test point insertion feature.

The following topics describe how to implement user-defined test points:

- [Enabling User-Defined Test Point Insertion](#)
- [Configuring User-Defined Test Points](#)
- [Limitations](#)

Enabling User-Defined Test Point Insertion

To enable user-defined test point insertion, you must first issue the following command before pre-DFT DRC:

```
dc_shell> set_dft_configuration -testability enable
```

After you have enabled user-defined test point insertion, you can enable and configure one or more user-defined test point specifications with the `set_test_point_element` command, as described in the following topics.

Caution:

If you do not enable the `testability` client, then the `set_test_point_element` command provides the legacy behavior described in [Appendix B, Legacy Test Point Insertion](#).

Configuring User-Defined Test Points

You can use the `set_test_point_element` command to specify the location and type of user-defined test points to insert in the design during DFT insertion, as well as other aspects of test point construction. User-defined test points can be defined at leaf pins, hierarchy pins, and ports. These test points are then inserted by the `insert_dft` command.

To define a user-defined test point, use the following command:

```
set_test_point_element
  -type force_0 | force_1 | force_01
    | control_0 | control_1 | control_01 | observe
  [-control_signal control_name]
  [-clock_signal clock_name]
  [-test_points_per_source_or_sink n]
  pin_port_list
```

Specify the test point type and the list of signal pins or ports to be forced, controlled, or observed. For more information on test point types, see [Test Point Types on page 294](#).

The remaining options, described in [Table 42](#), can be specified to override their defaults.

Table 42 set_test_point_element Options

To do this	Use this option
Use a particular test clock for any test point registers needed by these test points	<code>-clock_signal clock_name</code> (default is the dominant clock)
Use a particular TestMode, ScanEnable, or IbistEnable signal to enable these test points	<code>-control_signal control_name</code> (default is the first available TestMode signal)
Specify the number of force, control, or observe points (in this specification only) that can share a test point register	<code>-test_points_per_source_or_sink n</code> (default is 8)

Caution:

Options not listed above are unsupported. They are used by the legacy user-defined test point feature that is used when the `testability` DFT client is not enabled. See [Appendix B, Legacy Test Point Insertion](#).

The test points are implemented using clock, control signal, and test points per scan cell settings as follows, highest precedence first:

- From the `set_test_point_element` specification
- From the `set_testability_configuration` global specification
- Using the global testability defaults

Note:

The `user` testability target does not need to be enabled to implement test points using the `set_test_point_element` command, and the `user` target settings do not affect test points implemented by the `set_test_point_element` command.

By default, any needed source or sink registers are created by the `insert_dft` command.

Registers are not shared across multiple `set_test_point_element` specifications. If test points within a limited physical region should share registers, they should all be provided in a single `set_test_point_element` command.

If the specified pin list spans multiple clock or power domain configurations, the tool creates separate test point registers for each configuration.

After specifying test point definitions with the `set_test_point_element` command, you can report them with the `report_test_point_element` command, or remove them before DFT insertion with the `remove_test_point_element` command. For more information about these commands, see the man pages.

See Also

- [Sharing Test Point Registers on page 303](#) for details on test point registers sharing
- [Configuring the User-Defined Test Point Target on page 319](#) for details on the user testability target
- [Configuring Global Test Point Insertion Settings on page 307](#) for details on the global testability configuration

Limitations

User-defined test point insertion has the following limitations:

- The `-control_signal` option *must* be specified, otherwise the control signal for the test point logic is tied to logic 0.
- All specifications must use the same explicitly specified control signal; you cannot use multiple control signals across `set_test_point_element` specifications.
- Test point registers are always positive-edge, even if the dominant clocking around a test point is negative-edge.
- User-defined test points do not inherit the `-test_points_per_scan_cell` value from the global or user target `set_testability_configuration` value.

Previewing the Test Point Logic

To preview the test point logic that the tool will implement according to your specifications, use the following command after running test point analysis:

```
dc_shell> preview_dft -test_points all
```

Note:

If you are using TestMAX Advisor testability features, you must first run analysis as described in [Running Test Point Analysis on page 325](#).

The test point preview report is organized as follows:

- The legend at the beginning defines attributes used in the report.
 - Test point registers and test point pins use separate attributes.
- In the table, each line with values in the “Test Point Type” and “Pin” columns indicates a test point of that type at that pin.
- For each test point, the “Test Point Register” column indicates the source or sink register used for that test point.
 - An empty value means this test point shares the same register as the previous test point.
 - A value of “-” means no test point register is needed for that test point type.
- The summary at the end reports indicates how many test points and test point registers will be implemented.

[Example 30](#) shows an example test point preview report.

Example 30 Example Test Point Preview Report

```
***** Test Point Plan Report *****
```

Test point register attributes:
 d - dedicated (DFT-inserted) test point register
 f - reused (functional) test point register
 tpe - test point enable signal
 src - test point source signal
 snk - test point sink signal

Test point pin attributes:
 r - random-resistant test point pin
 x - X-blocking test point pin
 m - multicycle path test point pin
 w - core wrapper test point pin
 s - shadow wrapper test point pin
 g - self-gating test point pin
 a - AutoFix test point pin
 u - user-defined test point pin

Index	Test Point Register	Test Point Type	Pins
1	U_dft_tp_sdtc_ip_0/dtc_reg	(d, src) (CLK1)	
		force_01	IP_BLOCK/RX0 (u)
		force_01	IP_BLOCK/RX1 (u)
2	U_dft_tp_sdtc_ip_1/dtc_reg	(d, snk) (CLK1)	
		observe	IP_BLOCK/TX0 (u)
		observe	IP_BLOCK/TX1 (u)

```

3      sub3/mult_60/U_dft_tp_sdtc_ip_2/dtc_reg ( d, tpe ) ( CLK3 )
                  control_0          sub3/mult_60/U1057/Z (r)
                  control_1          sub3/mult_60/U1156/Z (r)
                  control_1          sub3/mult_60/U1262/Z (r)
                  control_0          sub3/mult_60/U1343/Z (r)
4      sub3/mult_60/U_dft_tp_sdtc_ip_3/dtc_reg ( d, snk ) ( CLK3 )
                  observe           sub3/mult_60/U4479/Z (r)
                  observe           sub3/mult_60/U4483/Z (r)
                  observe           sub3/mult_60/U4487/Z (r)
                  observe           sub3/mult_60/U4596/Z (r)
5      sub2/U_dft_tp_sdtc_ip_4/dtc_reg ( d, src ) ( CLK2 )
                  force_01          sub2/BBOX/Z[0] (x)
                  force_01          sub2/BBOX/Z[1] (x)
6      sub2/U_dft_tp_sdtc_ip_5/dtc_reg ( d, snk ) ( CLK2 )
                  observe           sub2/BBOX/A[0] (x)
                  observe           sub2/BBOX/A[1] (x)
7      -                multicycle       MULT_reg[3]/D (m)
8      -                multicycle       MULT_reg[2]/D (m)
9      -                multicycle       MULT_reg[1]/D (m)
10     -                multicycle       MULT_reg[0]/D (m)

***** Test Point Summary *****
Number of testability force_01 test points: 2
Number of testability control_0 test points: 2
Number of testability control_1 test points: 2
Number of testability observe test points: 6
Number of testability multicycle path test points: 4
Number of user-defined force_01 test points 2
Number of user-defined observe test points: 2

Total number of test points: 20
Total number of DFT-inserted test point registers: 6
*****
```

Inserting the Test Point Logic

After you define the test point insertion configuration, the `insert_dft` command inserts the test point logic. Test point scan registers are placed in the lowest level of hierarchy common to all test points for that register.

The tool creates a new test-mode signal if one is needed but not defined.

Using AutoFix

The AutoFix feature automatically fixes scan rule violations resulting from the following types of uncontrollable signals:

- Clock signals
- Asynchronous set signals
- Asynchronous reset signals
- Three-state bus enable signals
- Bidirectional enable signals

By default, only the three-state bus and bidirectional fixing capabilities are enabled. You can enable fixing for one or more additional signal types to fix testability problems in your design. You can specify AutoFix configurations globally or on particular design objects. AutoFix is supported in both the multiplexed flip-flop and LSSD scan styles.

When enabled, AutoFix automatically fixes all violations of the specified type(s) found by the `dft_drc` command. If there are no violations, AutoFix makes no changes to the design.

This topic covers the following:

- [When to Use AutoFix](#)
- [The AutoFix Flow](#)
- [Configuring AutoFix](#)
- [AutoFix Script Example](#)

When to Use AutoFix

Use AutoFix to resolve testability problems caused by uncontrollable signals, as described in the following topics:

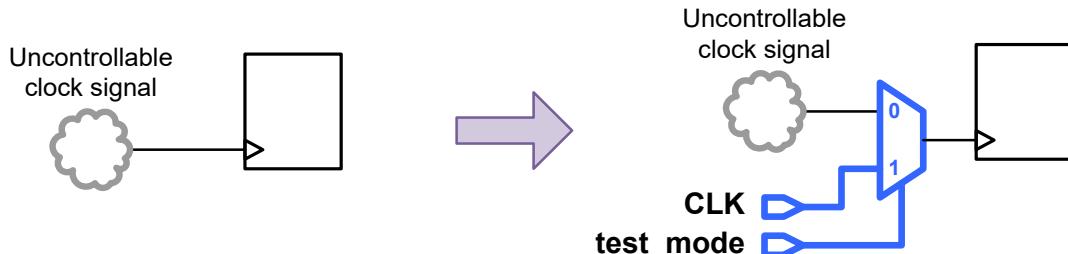
- [Uncontrollable Clock Signals](#)
- [Uncontrollable Asynchronous Set and Reset Signals](#)
- [Uncontrollable Three-State Bus Enable Signals](#)
- [Uncontrollable Bidirectional Enable Signals](#)

Uncontrollable Clock Signals

Each scan flip-flop in a design must be clocked by a signal that can be controlled by a primary input port. Otherwise, the clocking of data into the flip-flop cannot be controlled during test. Uncontrollable clock signals are flagged by the `dft_drc` command as design rule violations. If you do not fix these violations, the associated flip-flops are not included in scan chains and faults downstream from the flip-flop outputs might not be detectable.

When AutoFix is enabled for uncontrollable clock signals, it inserts a multiplexer test point to select a controllable clock signal during test, as shown in [Figure 129](#). The multiplexer is controlled by a test-mode signal. For mission-mode operation, the test-mode signal is inactive and the circuit operation is unchanged. During test, the signal is asserted and the flip-flop is clocked by the controllable primary input signal.

Figure 129 AutoFix Controllability Logic for an Uncontrollable Clock Signal

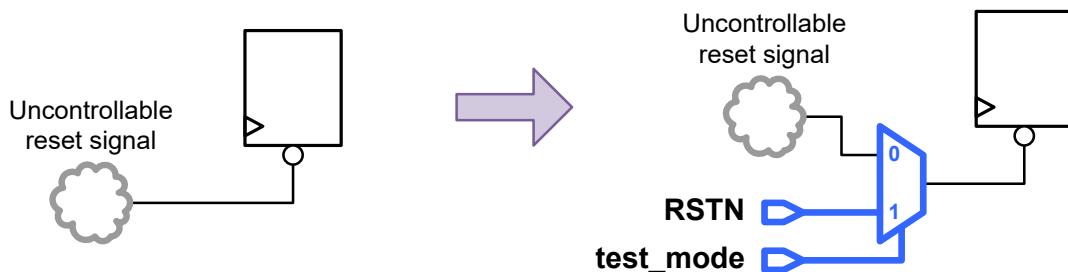


Uncontrollable Asynchronous Set and Reset Signals

The asynchronous set and reset 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.

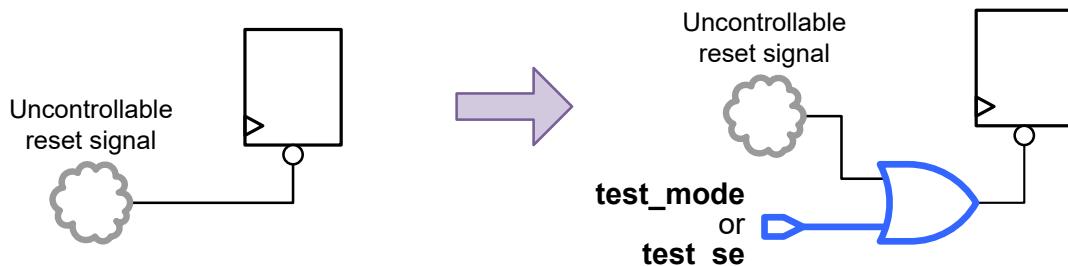
When AutoFix is enabled for uncontrollable asynchronous set or reset signals, by default it inserts a multiplexer test point to select a controllable set or reset signal during test, as shown in [Figure 130](#). The multiplexer is controlled by a test-mode signal. For mission-mode operation, the test-mode signal is inactive and the circuit operation is unchanged. During test, the asynchronous signal is driven by the controllable primary input signal.

Figure 130 AutoFix MUX-Based Controllability Logic for an Uncontrollable Reset Signal



AutoFix can also insert gating logic to de-assert the uncontrollable asynchronous set or reset signal, as shown in Figure 131. The gating logic is controlled by a test-mode or scan-enable control signal. For mission-mode operation, the control signal is inactive and the circuit operation is unchanged. When controlled by a test-mode signal, the asynchronous signal is held inactive throughout the entire test program. When controlled by a scan-enable signal, the asynchronous signal is held inactive during scan shift but remains controlled by the functional logic during scan capture.

Figure 131 AutoFix Gating-Based Controllability Logic for an Uncontrollable Reset Signal



Typically, one of the following configurations is used:

- The `mux` method with a test-mode signal

This method provides direct control of asynchronous resets during scan shift and scan capture. However, it blocks the functional reset path; this can cause faults in the reset logic to become untestable.

- The gate method with a scan-enable signal

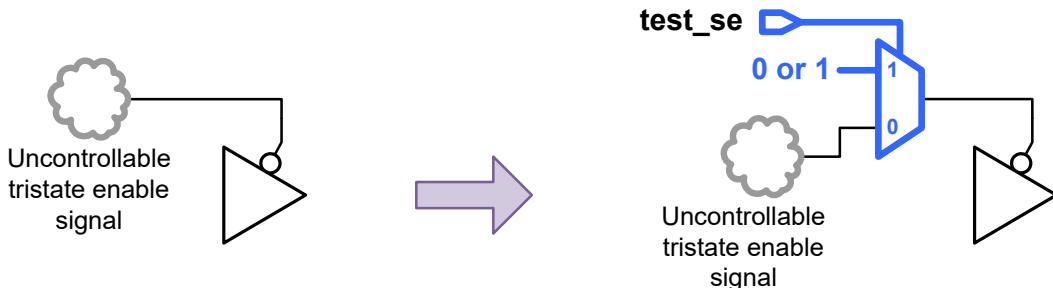
This method disables asynchronous resets during scan shift only. During scan capture, the functional reset logic controls the reset line; this does not impose any restrictions on reset logic testability.

Uncontrollable Three-State Bus Enable Signals

Three-state bus enable signals must be controllable during scan shift. Otherwise, the three-state buses might float or be driven to contention as controlling scan cells shift values through the scan chains.

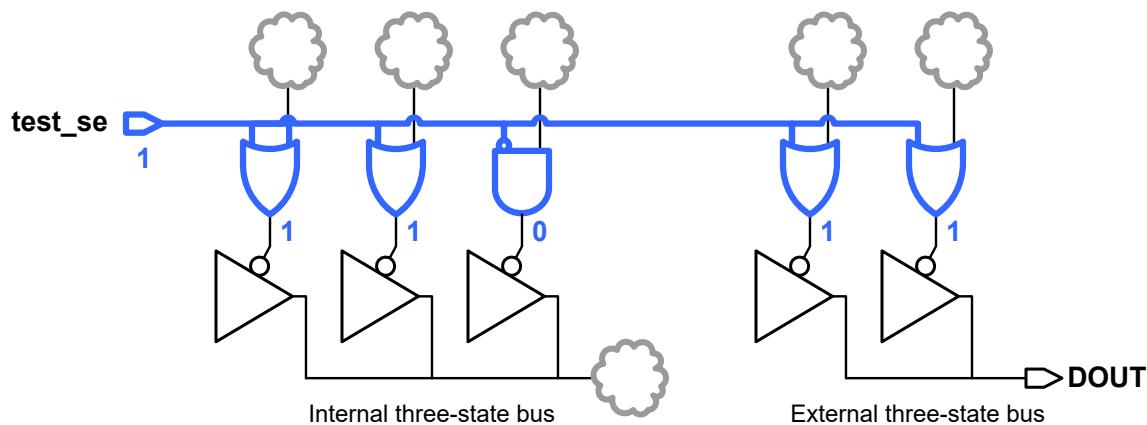
When AutoFix is enabled for an uncontrollable three-state enable signal, it inserts a multiplexer test point to select a constant enable signal during test, as shown in Figure 132. The multiplexer is controlled by a scan-enable signal. For mission-mode and scan capture operation, the test-mode signal is inactive and the circuit operation is unchanged. During scan shift, the signal is asserted during scan shift and the three-state driver is controlled by the constant value. Scan capture operation is unchanged.

Figure 132 AutoFix Controllability Logic for an Uncontrollable Tristate Signal



The constant value applied to the three-state driver depends on the type of three-state bus, shown in [Figure 133](#). For an internal bus that exists entirely within the current design, only a single tristate driver is active on each tristate net during scan shift; the rest are held inactive. For an external bus that has a driver outside the current design, none of the drivers are active on each tristate net during scan shift.

Figure 133 Three-State Bus Types



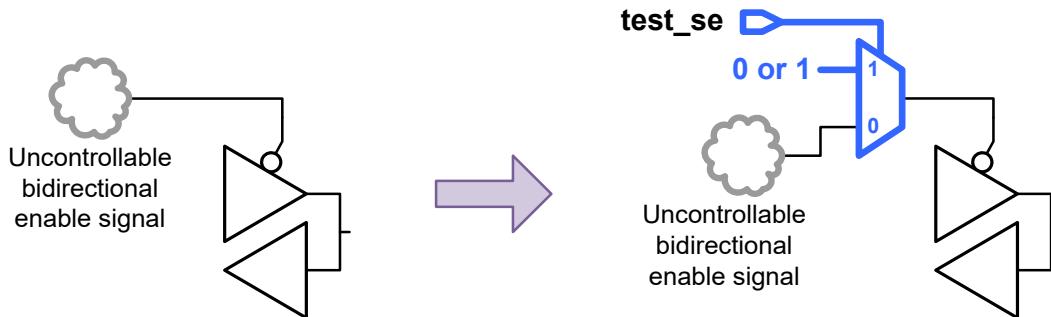
If you have a three-state bus that spans multiple blocks that are AutoFixed separately, at least one block should be fixed as an internal bus so that the bus is driven during scan shift.

Uncontrollable Bidirectional Enable Signals

Bidirectional enable signals must be controllable during scan shift. Otherwise, the bidirectional ports might float or be driven to contention as controlling scan cells shift values through the scan chains.

When AutoFix is enabled for an uncontrollable bidirectional enable signal, it inserts a multiplexer test point to select a constant enable signal during test, as shown in [Figure 134](#). By default, the constant value is chosen so that the bidirectional driver is in input mode during scan shift; scan capture operation is unchanged.

Figure 134 AutoFix Controllability Logic for an Uncontrollable Bidirectional Signal



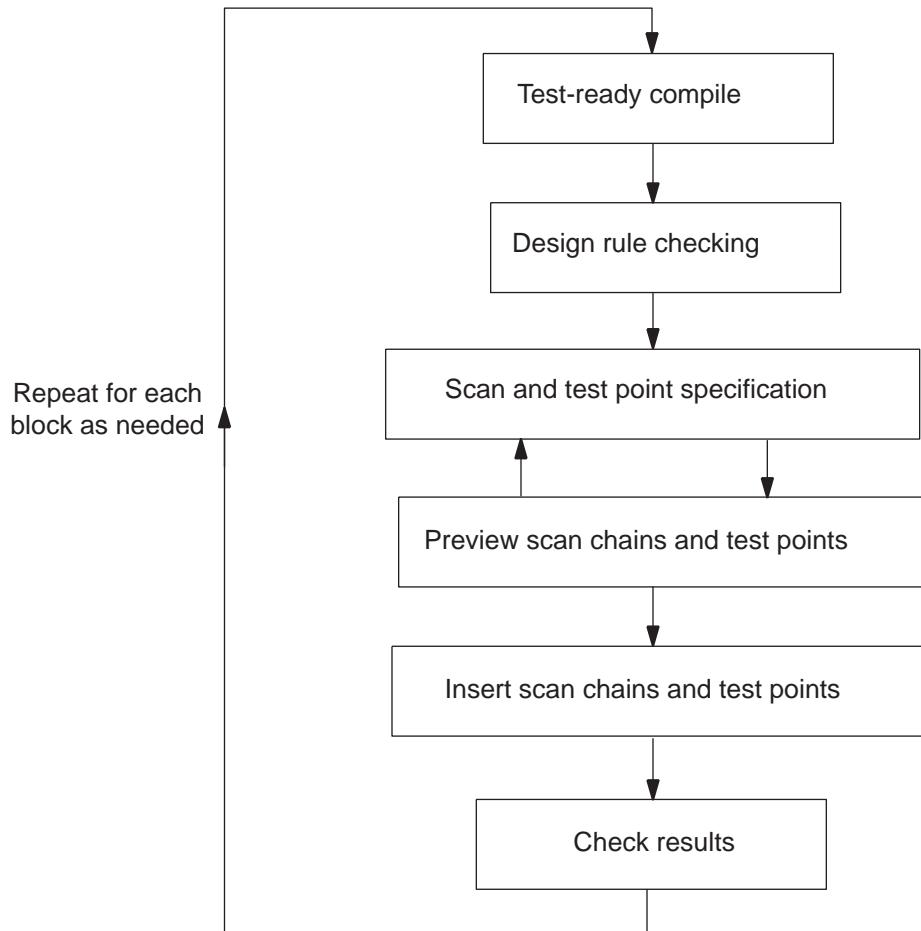
Scan-out ports driven by bidirectional drivers are always forced to the output direction during scan shift, regardless of whether AutoFix bidirectional fixing is enabled or how it is configured. For more information, see [Sharing a Scan Output With a Functional Port on page 228](#).

If a bidirectional driver cell drives an output port instead of an inout port, AutoFix classifies the driver as a three-state bus driver instead of a bidirectional driver because data values cannot propagate in the input direction.

The AutoFix Flow

The AutoFix design flow is very similar to the ordinary scan synthesis design flow. The general steps in the design flow are illustrated in [Figure 135](#).

Figure 135 Scan Synthesis Design Flow With AutoFix



You start with the `compile -scan` and `dft_drc` commands. Then you specify the parameters for scan insertion and AutoFix. After you set these parameters, you run the `preview_dft` command to get a preview of the scan chains and AutoFix 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` and `report_scan_path` commands.

Configuring AutoFix

The following topics explain how to use AutoFix:

- [Enabling AutoFix Capabilities](#)
- [Configuring Clock AutoFixing](#)
- [Configuring Set and Reset AutoFixing](#)
- [Configuring Three-State Bus AutoFixing](#)
- [Configuring Bidirectional AutoFixing](#)
- [Applying Hierarchical AutoFix Specifications](#)
- [Previewing the AutoFix Implementation](#)

Enabling AutoFix Capabilities

You can enable or disable individual AutoFix capabilities using the options of the `set_dft_configuration` command shown in [Table 43](#).

Table 43 Options of the set_dft_configuration Command to Enable AutoFix

Signal type to AutoFix	Enabling option of the set_dft_configuration command	Default
Clock signals	<code>-fix_clock disable enable</code>	disable
Asynchronous set signals	<code>-fix_set disable enable</code>	disable
Asynchronous reset signals	<code>-fix_reset disable enable</code>	disable
Three-state bus enable signals	<code>-fix_bus disable enable</code>	enable
Bidirectional enable signals	<code>-fix_bidirectional disable enable</code>	enable

By default, only the three-state bus and bidirectional fixing capabilities are enabled. 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.

After enabling an AutoFix capability, configure it using the `set_ autofix_configuration` command.

Configuring Clock AutoFixing

Use the following options of the `set_ autofix_ configuration` command to configure clock AutoFixing:

```
set_ autofix_ configuration
  -type clock
  [-test_ data clock_ signal]
  [-control_ signal test_ mode_ signal]
```

To specify an existing scan clock signal to use, define it as a `TestData` signal as well as a `ScanClock` signal, then specify it with the `-test_ data` option of the `set_ autofix_ configuration` command:

```
set_ dft_ signal -view existing_ dft -type ScanClock -port CLK \
  -timing [list 45 55]
set_ dft_ signal -view spec -type TestData -port CLK

set_ autofix_ configuration -type clock -test_ data CLK
```

If no clock signal is specified, AutoFix chooses an available scan clock signal that is also defined as a `TestData` signal. If no such signal exists, AutoFix creates a dedicated scan clock signal to use for fixing uncontrollable asynchronous clock signals.

To specify an existing test-mode signal to use for AutoFixed clock test points, use the `-control_ signal` option of the `set_ autofix_ configuration` command:

```
set_ dft_ signal -view spec -type TestMode -port AUTOFIX_ TM

set_ autofix_ configuration -type clock -control_ signal AUTOFIX_ TM
```

If no control signal is specified, AutoFix chooses an available test-mode signal that is not already used for another purpose. If no such signal exists, AutoFix creates a dedicated test mode signal.

Configuring Set and Reset AutoFixing

Use the following options of the `set_ autofix_ configuration` command to configure set and reset AutoFixing:

```
set_ autofix_ configuration
  -type set | reset
  [-method mux | gate]
  [-test_ data set_ reset_ signal]
  [-control_ signal control_ signal]
  [-fix_ latch disable | enable]
```

By default, AutoFix uses MUXs to fix uncontrollable asynchronous set or reset signals. To specify an existing controllable set or reset signal to be selected by the MUXs, define it as

a `TestData` signal as well as a Set or Reset signal, then specify it with the `-test_data` option of the `set_autofix_configuration` command:

```
set_dft_signal -view existing_dft -type Reset -port RSTN -active_state 0
set_dft_signal -view spec -type TestData -port RSTN -active_state 0

set_autofix_configuration -type set -test_data RSTN
set_autofix_configuration -type reset -test_data RSTN
```

If no such signal is specified, AutoFix chooses a set or reset signal that is also defined as a `TestData` signal. If no such signal exists, AutoFix creates a dedicated asynchronous reset signal to use for fixing uncontrollable asynchronous set and reset signals.

Use the `-type set` or `-type reset` option of the `set_autofix_configuration` command to configure set or reset AutoFixing, respectively. You can use a single existing asynchronous set or reset signal to AutoFix both uncontrollable set and reset signals. However, the same asynchronous signal cannot be used to AutoFix both the set and reset pins of the same cell. If this occurs in your design, you must specify separate set and reset signals.

The `mux` fixing method can use only a test-mode control signal. To specify an existing test-mode signal to control the MUXs, use the `-control_signal` option of the `set_autofix_configuration` command:

```
set_dft_signal -view spec -type TestMode -port AUTOFIX_TM

set_autofix_configuration -type set -control_signal AUTOFIX_TM
set_autofix_configuration -type reset -control_signal AUTOFIX_TM
```

If no control signal is specified, AutoFix chooses an available test-mode signal that is not already used for another purpose. If no such signal exists, AutoFix creates a dedicated test mode signal.

To fix uncontrollable sets and resets using gating logic instead of a MUX, use the `-method` option to specify the `gate` fixing method:

```
set_autofix_configuration -type set -method gate
set_autofix_configuration -type reset -method gate
```

The `gate` fixing method can use either a scan-enable or test-mode control signal. To specify a scan-enable or test-mode signal to control the gating logic, use the `-control_signal` option of the `set_autofix_configuration` command:

```
set_autofix_configuration -type set -method gate -control_signal SE_FIX
set_autofix_configuration -type reset -method gate -control_signal SE_FIX
```

If no control signal is specified, AutoFix chooses an available test-mode signal that is not already used for another purpose. If no such signal exists, AutoFix creates a dedicated test mode signal.

By default, AutoFix does not consider set or reset pins of latch cells. To consider latch set and reset pins, use the `-fix_latch enable` option of the `set_ autofix_ configuration` command.

Note:

If you use the `gate` fixing method along with a scan-enable control signal, you must also allow unstable reset signals in both DFT DRC and TestMAX ATPG DRC. For more information, see [SolvNet article 021644, "How Can I Improve Testability for Internally Generated Asynchronous Set and Reset Signals?"](#)

Configuring Three-State Bus AutoFixing

Use the following options of the `set_ autofix_ configuration` command to configure three-state bus AutoFixing:

```
set_ autofix_ configuration
  -type internal_bus | external_bus
    [-method no_disabling | enable_one | disable_all]
```

The `internal_bus` type configures three-state buses that do not drive ports of the current design. The `external_bus` type configures three-state buses that drive ports of the current design.

Use the `-method` option to control how the tristate drivers are to be enabled during scan shift. The values are

- `no_disabling` – do not insert controlling logic
- `enable_one` – enable one driver; disable all other drivers
- `disable_all` – disable all drivers

The default methods for the `internal_bus` and `external_bus` types are `enable_one` and `disable_all`, respectively. If you have a three-state bus that spans multiple blocks that are AutoFixed separately, configure one block to use the `enable_one` method to avoid bus float during scan shift.

AutoFix combines all scan-enable signals defined without the `-usage` option into a single merged signal to enable all three-state bus AutoFix test points. This ensures that no three-state contention occurs when data is scanned through the shift chains. You cannot use the `-control_signal` option to specify a control signal for three-state bus AutoFixing.

Configuring Bidirectional AutoFixing

Use the following options of the `set_ autofix_ configuration` command to configure bidirectional AutoFixing:

```
set_ autofix_ configuration
  -type bidirectional
    [-method input | output | no_disabling]
```

Use the `-method` option to control how the bidirectional drivers are to be enabled during scan shift. The values are

- `input` – force bidirectional drivers to input direction
- `output` – force bidirectional drivers to output direction
- `no_disabling` – do not insert controlling logic

The default method for the `bidirectional` type is `input`.

AutoFix combines all scan-enable signals defined without the `-usage` option into a single merged signal to enable all bidirectional AutoFix test points. This ensures that no bidirectional contention occurs when data is scanned through the shift chains. You cannot use the `-control_signal` option to specify a control signal for bidirectional AutoFixing.

Applying Hierarchical AutoFix Specifications

By default, AutoFix specifications applied with the `set_ autofix_ configuration` command apply to the entire design. To fix only particular design objects, specify them with the `-include_ elements` option of the `set_ autofix_ configuration` command. For example,

```
set_ autofix_ configuration -type set -include_ elements {MY_CORE}
set_ autofix_ configuration -type reset -include_ elements {MY_CORE}
```

To fix the design globally except for some particular design objects, specify them with the `-exclude_ elements` option of the `set_ autofix_ configuration` command. For example,

```
set_ autofix_ configuration -type set -exclude_ elements {U_IP_CORE}
set_ autofix_ configuration -type reset -exclude_ elements {U_IP_CORE}
```

You can specify both the `-include_ elements` and `-exclude_ elements` options to exclude cells within an included cell, but not vice versa.

The `set_ autofix_ configuration` command applies a global configuration; subsequent specifications for a capability take precedence over previous specifications. If you need to specify different fixing configurations for different areas of the design, use the `set_ autofix_ element` command, which differs from the `set_ autofix_ configuration` command as follows:

- The `set_ autofix_ element` command applies a local fixing configuration to the specified list of design objects.
- Multiple `set_ autofix_ element` command specifications can be applied to the design.

You can mix global and local specifications. The global fixing configuration is used where no local configuration applies. For example,

```
# specify global fixing configuration
set_ autofix_ configuration -type clock -test_ data CLK
```

```
# specify local fixing configuration
set_ autofix_element -type clock -test_data RXCLK {UBLK_RX}
set_ autofix_element -type clock -test_data TXCLK {UBLK_TX}
```

To apply local fixing configurations only to particular parts of the design without also globally fixing the design, specify a global fixing configuration that includes all of the local design objects (to limit global fixing), then apply the local fixing specifications. For example,

```
# specify global configuration that prevents fixing outside UBLK_RX
# and UBLK_TX
set_ autofix_configuration -type clock -include_elements {UBLK_RX UBLK_TX}

# specify local fixing configuration
set_ autofix_element -type clock -test_data RXCLK {UBLK_RX}
set_ autofix_element -type clock -test_data TXCLK {UBLK_TX}
```

Note the following precedence rules:

- Local specifications (applied with the `set_ autofix_element` command) take precedence over global specifications (applied with the `set_ autofix_configuration` command).
- Local specifications at lower hierarchy levels take precedence over local specifications at higher levels.
- If multiple specifications apply to the same object, later specifications take precedence over earlier specifications.
- Specifications for different fixing types are independent and do not affect each other.

[Table 44](#) shows the valid design object types you can use in AutoFix specifications for each fixing type.

Table 44 Valid Design Object Types for AutoFix Specifications

Fixing type	Valid design object types
clock	cell (hierarchical and leaf)
set	cell (hierarchical and leaf)
reset	cell (hierarchical and leaf)
internal_bus	net
external_bus	net
bidirectional	port

Previewing the AutoFix Implementation

After you enable and configure the AutoFix capabilities, you can preview the scan architecture with the fixes included. To do this, use the `preview_dft -test_points all -show {cells}` command, which provides the following information:

- The number of test points implemented by AutoFix
- The scan chain configuration with AutoFix considered

The test point section of the preview report shows the test points to be implemented. It does not include three-state bus or bidirectional test points.

The scan cells section of the preview report shows only the sequential cells included in scan chains; it does not show the cells omitted due to DRC violations. Check to see if the DRC-violating cells to be AutoFixed exist in the report. If needed, you can revise the AutoFix configuration and rerun the `preview_dft` command.

Note:

Before DFT insertion, the `dft_drc` command always reports the DRC violations without AutoFix considered; use the `preview_dft` command to assess the effects of AutoFix.

When you are satisfied with the scan chains and test points reported by the `preview_dft` command, insert DFT using the `insert_dft` command. You should always run the `dft_drc` command after DFT insertion to check for any remaining design rule violations.

AutoFix Script Example

The script in [Example 31](#) fixes uncontrollable clock and reset signals using AutoFix.

Example 31 Scan Synthesis With Test Point Insertion

```
current_design MY_DESIGN
compile -scan
create_test_protocol
dft_drc

# configure scan
set_scan_configuration -clock_mixing mix_edges ...

# configure DFT signals
set_dft_signal -view spec -type TestMode -port TEST_MODE

set_dft_signal -view existing_dft -type ScanClock \
    -port {CLK RXCLK TXCLK} -timing {45 55}
set_dft_signal -view existing_dft -type Reset -port RSTN -active_state 0

# configure signals for AutoFix
```

```

set_dft_signal -view spec -type TestData -port {CLK RXCLK TXCLK RSTN}

# configure clock AutoFix
set_dft_configuration -fix_clock enable
set_automix_configuration -type clock -test_data CLK
set_automix_element -type clock -test_data RXCLK {U_RX_BLK}
set_automix_element -type clock -test_data TXCLK {U_TX_BLK}

# configure reset AutoFix
set_dft_configuration -fix_reset enable
set_automix_configuration -type reset -test_data RSTN \
-exclude_elements {U_IP_CORE}

preview_dft -test_points all
insert_dft
dft_drc
report_scan_path -chain all

```

Using Pipelined Scan Enables for Launch-On-Extra-Shift (LOES)

You can use pipelined scan-enable signals to provide launch-on-extra-shift (LOES) transition-delay timing in TestMAX ATPG, which improves ATPG efficiency and reduces pattern count. This is described in the following topics:

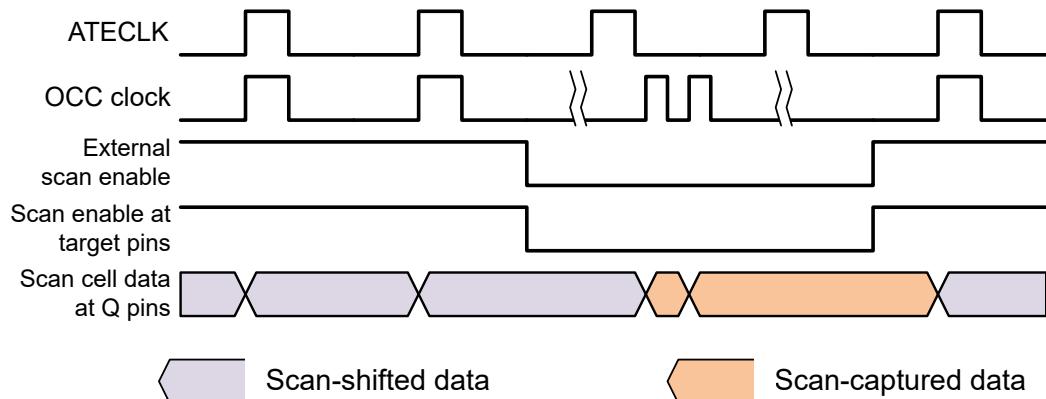
- [The Pipelined Scan-Enable Architecture](#)
- [Pipelined Scan-Enable Requirements](#)
- [Implementing Pipelined Scan-Enable Signals](#)
- [Pipelined Scan-Enable Signals in Hierarchical Flows](#)
- [Implementation Considerations for Pipelined Scan-Enable Signals](#)
- [Pipelined Scan Enable Limitations](#)

The Pipelined Scan-Enable Architecture

Transition-delay fault testing requires two at-speed clock cycles in a row—one to launch and one to capture—so that the launched data must propagate through the logic cones at-speed to be captured properly. These at-speed clock pulses are typically provided by an on-chip clocking (OCC) controller, as described in [Chapter 12, On-Chip Clocking Support.](#)

By default, DFT Compiler implements a simple scan-enable signal, which is externally controlled at test clock frequencies. This requires the use of Launch-on-Capture (LOC) transition-delay timing, shown in [Figure 136](#), where scan enable is de-asserted *before* both the launch and capture clock pulses.

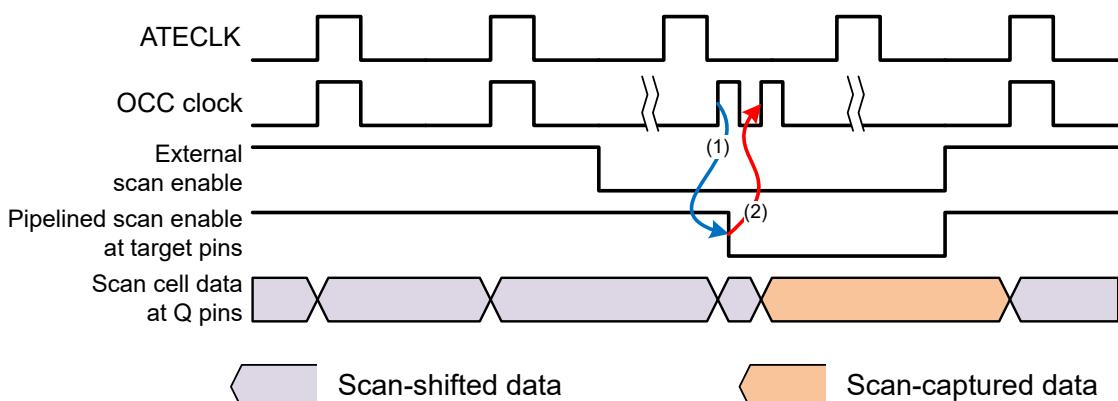
Figure 136 Launch-on-Capture (LOC) Transition-Delay Timing



Because scan enable is de-asserted for the launch clock pulse, the launch data must be controlled indirectly (captured through the logic cones) by the data from the previous scan-shift cycle. This requires fast-sequential ATPG, which imposes additional constraints and overhead on ATPG.

To avoid this limitation, you can implement *pipelined scan-enable signals*, which can generate the scan-enable transition at-speed. This allows launch-on-extra-shift (LOES) transition-delay timing, as shown in [Figure 137](#), where TestMAX ATPG directly controls the launch data.

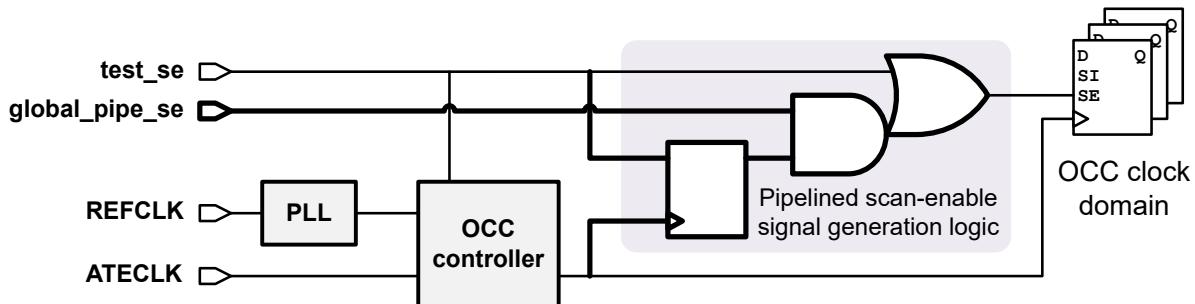
Figure 137 Launch-On-Extra-Shift (LOES) Transition-Delay Timing



The waveforms for the external signals are identical between LOC and LOES, but the scan-enable signal that reaches the scan cells is different.

The pipelined scan-enable feature works by creating a registered scan-enable transition on the chip, just as an OCC controller generates controlled at-speed clock pulses on the chip. [Figure 138](#) shows the pipelined scan-enable logic structure.

Figure 138 Pipelined Scan-Enable Logic



The global_pipe_se signal controls whether the output scan-enable signal operates in nonpipelined or pipelined mode:

- When the global_pipe_se signal is de-asserted, the register is bypassed and the output scan-enable signal is a simple non-pipelined scan-enable signal.
 - When the global_pipe_se signal is asserted,
 - When the input scan-enable signal is asserted, the output scan-enable signal is immediately asserted.
 - When the input scan-enable signal is de-asserted, the register holds the output scan-enable asserted until the next leading clock edge of the *at-speed* clock.

Each clock domain and clock edge has its own pipelined scan-enable logic construct that de-asserts the scan-enable signal synchronized to that clock edge.

Note:

The pipelined scan-enable feature and the pipelined scan data feature are independent and unrelated. Either can be used separately, or they can be used together, but the commands, limitations, and messages are specific to one or the other and must not be confused.

For more information on launch-on-capture (LOC) and launch-on-extra-shift (LOES) timing, see the “Transition-Delay Fault ATPG Timing Modes” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Pipelined Scan-Enable Requirements

Launch-on-extra-shift (LOES) with pipelined scan-enable signals has the following requirements:

- All scan-enable signals must have the same pipelined scan-enable depth after DFT insertion. Valid depths are zero (not pipelined) and one (pipelined).
- Each clock domain and clock edge must have its own pipelined scan-enable signal, which all scan flip-flops clocked by that domain and edge must use.
- Clock-gating cells should also use the pipelined scan enable for its clock domain and edge.

It is possible to use LOES when clock-gating cells use the nonpipelined scan enable. However, you should use the pipelined scan enable because it maximizes the amount of data that can be launched in the extra shift, which should result in a smaller pattern set. It also allows the use of the legacy ATPG method Launch-On-Last-Shift, where the last scan shift doubles as the transition launch cycle (which might be important in the case of design reuse).

- The non-pipelined scan enable must be used for OCC controllers. This allows the OCC controller to generate the extra-shift launch clock followed by the capture clock as an at-speed clock pair.
- The non-pipelined scan enable must be used for clock chains used by OCC controllers. This holds the clock chain data steady during capture. If the pipelined scan enable is used, the clock chain bits are corrupted by the launch clock.
- When clock mixing is used, lock-up latches might be ineffective because the clock timing in the extra shift uses the launch waveform table rather than shift timing. As a result, DRC in the TestMAX ATPG tool marks the first flip-flop following the clock domain crossing as disturbed, which can reduce coverage slightly (although an incremental LOC run can detect these faults).

All on-chip scan-enable signals (nonpipelined, and pipelined for each clock edge) can be derived from a single scan-enable signal source.

Implementing Pipelined Scan-Enable Signals

To implement pipelined scan-enable signals, do the following:

1. Enable the pipelined scan-enable feature:

```
dc_shell> set_scan_configuration -pipeline_scan_enable true
```

2. Define the enable signal:

```
dc_shell> set_dft_signal -view spec -type LOSPipelineEnable \
    -port PSE_EN -active_state 1 -test_mode all
```

This enable signal selects launch-on-extra-shift (LOES) when asserted and Launch-on-Capture (LOC) operation when de-asserted.

If you do not define an enable signal, the tool creates a signal named `global_pipe_se`.

You can use the `-connect_to` option to control the enable signal per clock domain. The enable signal must be defined as a `ScanClock` before the pipelined scan enable definition. For example:

```
dc_shell> set_dft_signal -view spec -type LOSPipelineEnable \
    -port PSE_EN -active_state 1 -test_mode all -connect_to
    CLK1
```

You can drive the enable signal from an on-chip configuration register by using the internal pins flow. For more information, see [Internal Pins Flow on page 429](#).

3. (Optional) In Design Compiler Graphical, implement pipelined scan-enable clusters for large or timing-critical designs.

You can create multiple physically compact *clusters* of scan cells, each with their own local pipelined scan-enable (PSE) construct. This improves the critical path delay between the pipeline registers and their scan cells.

To do this, use the `-pipeline_fanout_limit` option of the `set_scan_configuration` command to specify the number of scan cells per PSE cluster. The tool creates as many clusters as needed, each with exactly the specified number of scan cells. (The last-created cluster in each clock domain might have fewer scan cells.) The pipeline registers have the `size_only` attribute set, which prevents duplicate register merging by the `compile_ultra` command.

See [Implementation Considerations for Pipelined Scan-Enable Signals on page 353](#) for details on clustering and timing. See SolvNet article 2543967, “The Pipelined Scan-Enable Fanout Limit, Duplicate Scan-Enable Signals, and the TEST-1073 Error,” for details on how the fanout limit is applied.

The default is to not implement clusters.

Caution:

Do not use this feature in wire load mode because scan cell clustering cannot use physical information, which might degrade results.

If you are using a DFT-inserted OCC controller, the clock connection might be incorrect. See [Pipelined Scan Enable Limitations on page 356](#) for details.

The test protocol created by the tool does not constrain the pipeline enable signal, so post-DFT DRC checks both the LOC and LOES modes. The protocol written out by the `write_test_protocol` command also does not constrain the enable signal. Use the `add_pi_constraint` command in TestMAX ATPG to assert or de-assert the pipeline enable signal. See “Using Launch-On Extra-Shift Timing” in TestMAX ATPG and TestMAX Diagnosis Online Help for more information.

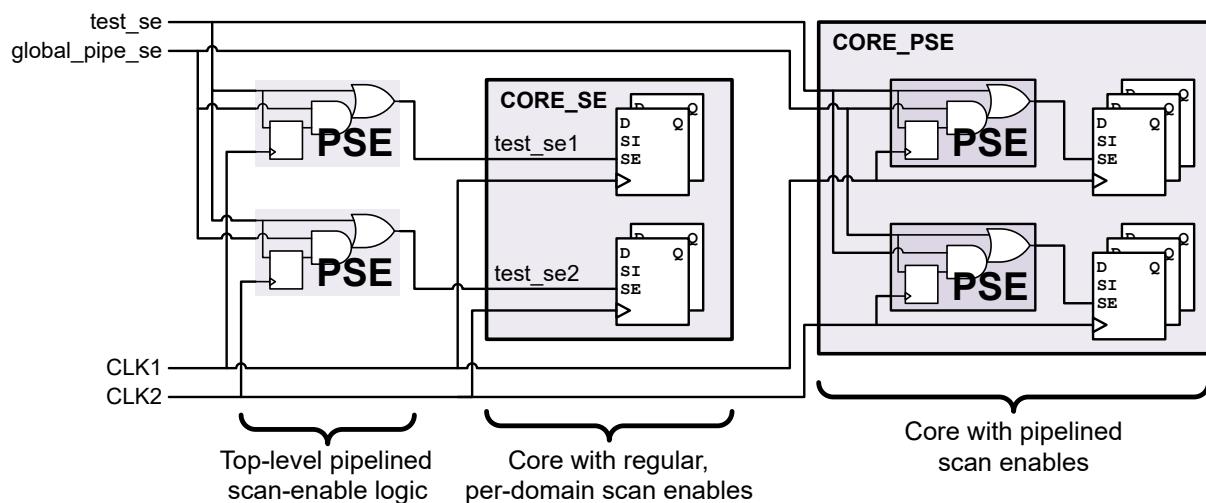
Pipelined Scan-Enable Signals in Hierarchical Flows

In hierarchical flows, the rules for pipelined scan-enable signals are:

- You must enable pipelined scan enables at every hierarchical integration level above where pipelined scan enables have been implemented.
- The tool adds top-level pipeline logic to top-level logic and nonpipelined cores.
- Nonpipelined cores must be created with per-clock-domain scan-enable signals to allow pipelining at the top level.

[Figure 139](#) shows the integration of a pipelined and nonpipelined core.

Figure 139 Pipelined Scan-Enable and Core Integration



Creating Cores With Pipelined Scan Enable

To create a core that contains pipelined scan enables, see [Implementing Pipelined Scan-Enable Signals on page 350](#).

Integrating Cores With Pipelined Scan Enable

At the top level, enable and configure the pipelined scan-enable feature, as described in [Implementing Pipelined Scan-Enable Signals on page 350](#). You must do this at all hierarchical core integration levels above where the feature is used.

The tool recognizes when a core already contains pipelined scan-enable signals. It automatically makes the connections from the top-level signals to the corresponding core pins. It does not insert any additional pipelining logic for these connections, which ensures that all scan cells have exactly one level of scan-enable pipelining.

Implementing Nonpipelined Scan-Enable Cores That Can Be Pipelined

You can create nonpipelined scan-enable DFT-inserted cores whose scan-enable signals can be pipelined when the core is integrated. This might be useful in some design scenarios, such as to share top-level pipeline constructs across multiple integrated cores, or to create blocks for design reuse that might or might not require pipelined scan enables.

To do this, create the core as follows:

1. Enable domain-specific scan-enable signals:

```
dc_shell> set_scan_configuration -domain_based_scan_enable true
```

This causes the tool to create a nonpipelined scan-enable signal for each scan clock and edge.

2. (Optional) If you have existing ports for each domain-specific scan-enable signal, define them using the `-associated_clock` option of the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type ScanEnable \
    -port SE_CLK1 -associated_clock CLK1
```

```
dc_shell> set_dft_signal -view spec -type ScanEnable \
    -port SE_CLK2 -associated_clock CLK2
```

```
dc_shell> set_dft_signal -view spec -type ScanEnable \
    -port SE_CLK3 -associated_clock CLK3
```

Caution:

When using the `-associated_clock` option, you must define scan-enable signals for *all* scan clock domains. Otherwise, the tool reuses existing scan

signals for unspecified clocks (instead of creating new signals), which is incorrect.

Note:

You cannot use the `-associated_clock` option to define a scan-enable signal for an OCC controller or clock chain. The tool always creates a new scan-enable signal for them.

The scan-enable signals are timing-critical because they will eventually be driven by pipeline registers. Be sure the signals are constrained at-speed and meet timing. For timing details, see [Implementation Considerations for Pipelined Scan-Enable Signals on page 353](#).

When you integrate the core at the top level (with pipelined scan-enable signals), the tool automatically drives each core-level scan enable with a top-level pipeline construct of the corresponding clock.

Implementation Considerations for Pipelined Scan-Enable Signals

Note the following considerations when implementing pipelined scan-enable signals.

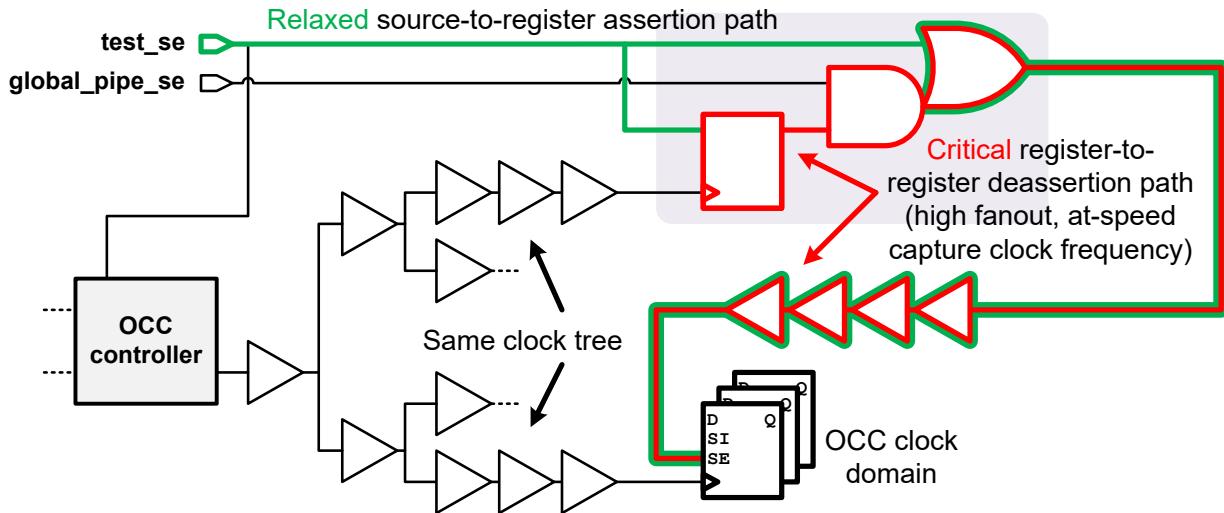
Timing Considerations

For a regular scan-enable signal, the signal path is a single-cycle path at the shift clock frequency, which is likely to be fairly slow—and even that requirement can be relaxed by adding extra cycles in TestMAX ATPG using the `-use_delay_capture_start` and `-use_delay_capture_end` options of the `write_patterns` command.

For a pipelined scan-enable signal, this relaxed timing applies only to the external signal source. The de-assertion event from the pipeline register must reach *all* downstream flip-flop scan-enable pins in a single cycle at the capture clock frequency, which is much faster than shift. This path cannot be relaxed because a multicycle path would result in bad patterns, and slowing down the clock makes the test less-than-at-speed.

[Figure 140](#) shows the relaxed externally driven assertion path in green and the critical register-driven de-assertion path in red.

Figure 140 Pipelined Scan-Enable Pipeline Register Timing Path



The pipeline register and its fanout scan cells are all typically driven by the same skew-balanced clock tree. As a result, the scan-enable buffer tree paths are standard register-to-register timing paths, with allowable delays between nearly zero and almost a full clock cycle. The buffer tree does not need to be skew-balanced.

In addition,

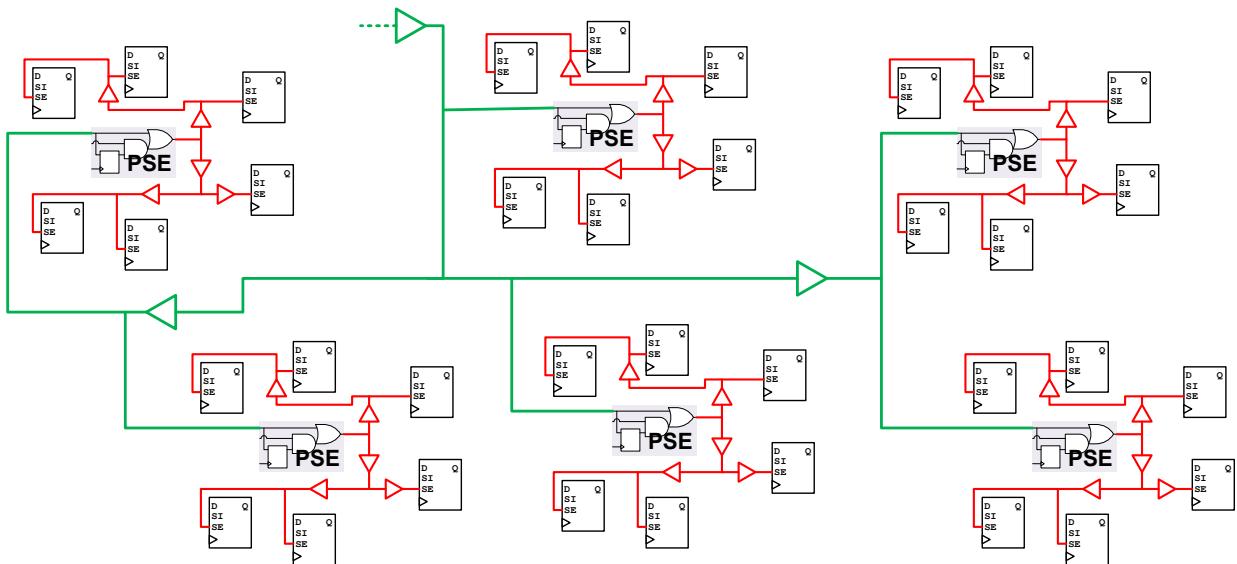
- For high-frequency clocks driving large clock domains, the buffer tree delay might be greater than the clock period. In this case, you can move the pipeline register's clock connection earlier in the clock tree, as long as hold constraints are met in all operating conditions.
- If you use power-aware functional output gating, the critical path for transition delay testing becomes the path from the scan-enable pipeline register through the toggle suppression gate into the functional logic. This feature is described in [Reducing Shift Power Using Functional Output Pin Gating on page 414](#).

Physical Synthesis Implementation Considerations

In Design Compiler in topographical mode and in Design Compiler Graphical, the tool creates physically aware buffer trees on the pipelined scan-enable nets to avoid synthesis design rule violations. For details, see “Performing Automatic High-Fanout Synthesis” in the *Design Compiler User Guide*.

If you set a fanout limit to implement pipelined scan-enable clusters, the tool analyzes scan cell locations to create compact clusters. Then, it builds physically aware buffer trees to all cluster pipeline registers (green) and within each cluster from the pipeline register to the scan cells (red), as shown in [Figure 141](#).

Figure 141 Pipelined Scan-Enable Signal With Fanout-Limited Scan Cell Clusters



See SolvNet article 2552200, “Visualizing Pipelined Scan-Enable Clusters in the Layout View” to display the clustering in your own design.

Wire Load Mode Implementation Considerations

In wire load mode, the tool buffers high-fanout nets by default, but because there is no physical information, it is likely that these buffer trees will be removed and rebuilt during layout.

To defer scan-enable network buffering until layout, use the following commands after DFT insertion and before post-DFT optimization:

```
# disable DRC fixing on scan nets
set_auto_disable_drc_nets -scan true ;# includes PSE register net

# apply ideal network property to SE ports and PSE register outputs
set_ideal_network {test_se global_pipe_se}
set_ideal_network [get_pins -hierarchical *test_pipe_se/* \
    -filter {pin_direction == out}]
```

In wire load mode, it is recommended not to set a fanout limit to create pipelined scan-enable clusters because there is no physical information to optimally choose which scan cells to include in each cluster.

Pipelined Scan Enable Limitations

Note the following requirements and limitations of pipelined scan-enable signals:

- Scan-enable signals defined with the `-usage scan` option of the `set_dft_signal` command are not supported. Scan-enable signals with no usage, and those defined with the `-usage clock_gating` option, are supported.
- When using DFT-inserted OCC controllers, the clock connection to the pipeline scan-enable registers might be wrong.

This usually means that the tool uses the clock coming from the uncontrolled PLL source. In this case, TestMAX ATPG will generate patterns, but those patterns will fail Verilog simulation. The correct clock connection to the pipeline scan enable register is the output of the OCC controller, at the leaf level of the clock tree. If the register is in a module with scan flip-flops that it controls, the same clock signal that drives the scan flip-flops should also drive the pipeline scan-enable register. The register's instance name is `test_pipe_se_<m>_reg_<n>`. There might be more than one for each clock, and they might be buried in the design hierarchy, depending on the settings.

- If you set the `-internal_clocks` option of the `set_scan_configuration` command to `multi` or `single`, pipelined scan-enable insertion and domain-based scan-enable insertion treat each internal clock region as a separate scan clock domain, which might not be the desired result.

Excluding Elements from a Pipelined Scan-Enable Configuration

When you enable the pipelined scan enable feature, all scan-enable connections are subject to pipelining behavior. However, in some design scenarios, you might want to exclude specific scan elements from pipelining. To do this, use the `set_pipeline_scan_enable_configuration` command with the `-exclude_element_list` option.

The element list to be excluded can consist of any of the following:

- Instance names - The tool excludes any scan-enable connection to that cell.
- Hierarchies - The tool excludes any scan-enable connection to any instance contained in that hierarchy or its child hierarchies.
- Scan clock ports – The tool excludes any scan element that belongs to that clock domain.
- Scan clock hookup pins - The tool excludes any scan element that belongs to that clock domain.

This example shows how you can exclude a specified hierarchy:

```
dc_shell> set_pipeline_scan_enable_configuration -exclude { de_a }
```

When a specific scan-enable connection is excluded from a pipelined scan-enable signal, the tool routes the connection as if the pipelined scan-enable were not enabled. The tool tries to connect directly from the port or hookup-pin to the scan-enable pin of the scan cell.

Multiple Test Modes

A design's scan interface must accommodate a variety of structural tests. Scan test, burn-in, and other tests performed at various production steps might require different types of access to scan elements of a design. To accommodate these different test requirements, multiple scan architectures can be provided on the same scan design. This approach is also useful for complex test schemes such as scan compression and core wrapping, which target tests in different parts of a design at different times.

The following topics explain the process for setting up architectures to perform multiple test-mode scan insertion:

- [Introduction to Multiple Test Modes](#)
- [Defining Test Modes](#)
- [Applying Test Specifications to a Test Mode](#)
- [Recommended Ordering of Global and Mode-Specific Commands](#)
- [Using Multiple Test Modes in Hierarchical Flows](#)
- [Supported Test Specification Commands for Test Modes](#)
- [Multiple Test-Mode Scan Insertion Script Examples](#)

See Also

- [DFTMAX Scan Compression and Multiple Test Modes on page 675](#) for more information about defining and configuring multiple DFTMAX compression modes
- [DFTMAX Ultra Compression and Multiple Test Modes on page 940](#) for more information about defining and configuring multiple DFTMAX Ultra compression modes

Introduction to Multiple Test Modes

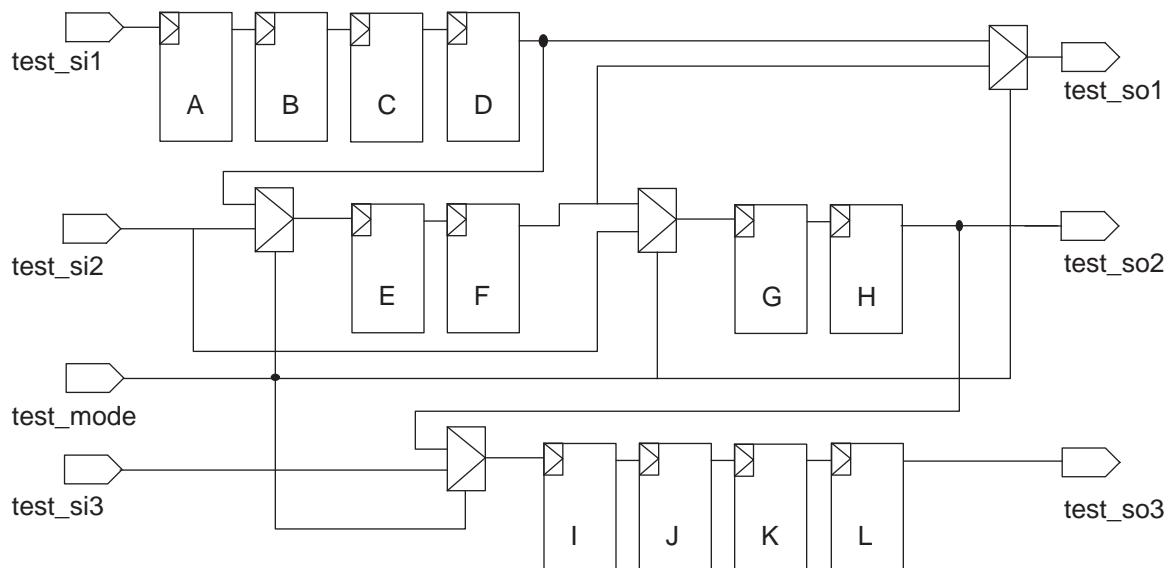
You can reconfigure the scan chains in your design to suit various tester requirements by defining different modes of operation, called *test modes*. For example, suppose you have a simple design with 12 scan cells that must operate in two different scan modes. In one mode, scan data is shifted through two chains (six cells each), and in the other mode, scan data is shifted through three chains (four cells each). [Figure 142](#) shows how DFT

Compiler inserts MUXes to support these two scan chain configurations. These MUXes are known as *reconfiguration MUXes*.

Note:

Reconfiguration MUXes might appear at any level in your design. This is usually dependent on the location of the scan elements where the MUXing takes place.

Figure 142 Configuring Scan Chains With Test-Mode Logic



Each test mode is activated by asserting one or more test-mode signals according to a particular test-mode encoding. Different test modes can have different scan-in and scan-out pin counts, and can even have independent sets of scan-in and scan-out pins altogether.

Defining Test Modes

To define a test mode, use the `define_test_mode` command:

```
define_test_mode test_mode_name
```

Each test mode must have a unique name that is used to refer to the test mode in subsequent DFT commands or reports. The name of the default standard scan mode

is `Internal_scan`. You can configure this default mode and define additional modes, or you can create an entirely new set of named modes without using the default test mode.

[Example 32](#) defines three newly-named test modes.

Example 32 Defining Three New User-Defined Test Modes

```
define_test_mode LONG      ;# long scan chains
define_test_mode MEDIUM   ;# medium scan chains
define_test_mode SHORT    ;# short scan chains
```

Defining the Usage of a Test Mode

By default, a test mode represents a standard scan test mode of operation. To specify how a test mode is to be used, use the `-usage` option of the `define_test_mode` command.

The valid keywords for this option are:

- `scan` – This is the traditional standard scan mode operation, which is the default if the `-usage` option is not specified. 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.
- `scan_compression` – This is the compressed scan mode of operation provided by the DFTMAX compression. In this mode, the internal scan chains are driven by combinational scan data decompressors, and the scan chains drive the combinational scan data compressors. This mode is used for testing all logic internal to the core with reduced test data volume and test application time.
- `streaming_compression` – This is the compressed scan mode of operation provided by DFTMAX Ultra compression. In this mode, the internal scan chains are driven by shift register scan data decompressors, and the scan chains drive the shift register scan data compressors. This mode is used for testing all logic internal to the core with significantly reduced test data volume and test application time.
- `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.

[Example 33](#) defines three standard scan test modes plus a DFTMAX compressed scan mode.

Example 33 Providing Test-Mode Usage Information With the -usage Option

```
define_test_mode LONG      ;# long scan chains
define_test_mode MEDIUM    ;# medium scan chains
define_test_mode SHORT     ;# short scan chains
define_test_mode COMPRESSED -usage scan_compression
```

After test modes have been defined, you can use the `list_test_modes` command to report the currently defined test-mode names. [Example 34](#) shows the report for the three standard scan test modes defined in [Example 32](#).

Example 34 Test Modes Reported by the list_test_modes Command

```
Test Modes
=====
```

```
Name: LONG
Type: InternalTest
Focus:

Name: MEDIUM
Type: InternalTest
Focus:

Name: SHORT
Type: InternalTest
Focus:

Name: Mission_mode
Type: Normal
```

Defining the Encoding of a Test Mode

Each test mode is activated by asserting one or more test-mode signals according to a particular encoding associated with that test mode. To declare these test-mode signals, use the `set_dft_signal -type TestMode` command. If no test-mode signals are available, or not enough test-mode signals are available to satisfy the test-mode encodings, DFT Compiler creates new test-mode ports as needed.

By default, DFT Compiler assigns a binary encoding to the test modes. Binary encoding requires the fewest number of test-mode signals. With binary encoding, n test-mode signals can provide test-mode encodings for up to $2^n - 1$ test modes, allowing for an encoding that deactivates all test modes and activates mission mode.

To report the test-mode signals and encodings associated with each test mode, use the `preview_dft` command. [Example 35](#) shows the preview report for the three standard scan test modes defined in [Example 32](#).

Example 35 Test-Mode Encodings Reported by the preview_dft Command

```
=====
Test Mode Controller Information
=====

Test Mode Controller Ports
-----
test_mode: test_mode2
test_mode: test_mode1

Test Mode Controller Index (MSB --> LSB)
-----
test_mode2, test_mode1

Control signal value - Test Mode
-----
00 LONG - InternalTest

01 MEDIUM - InternalTest

10 SHORT - InternalTest
```

You can also specify one-hot test-mode encoding by using the following command:

```
set_dft_configuration -mode_decoding_style one_hot
```

This command causes one-hot test-mode encodings to be used, and simplified test-mode decoding logic to be built that takes advantage of the properties of one-hot encodings. One-hot encodings provide simplified decoding logic, at the expense of a greater number of required test-mode signals. With one-hot encoding, n test-mode signals can provide test-mode encodings for up to n test modes, with mission mode being activated by an encoding where none of the test-mode signals are asserted.

Note:

One-hot test-mode encodings take the active state of each test-mode signal into account. If you define all test-mode signals using the `-active_state 0` option of the `set_dft_signal` command, each one-hot encoding contains a single asserted 0 value.

You can also specify your own test-mode encodings with the `-encoding` option of the `define_test_mode` command. The syntax of the encoding argument consists of one or more test-mode signal names and binary value pairs. These pairs are separated by a space when multiple ports are specified.

Example 36 shows how to use a binary encoding where mission mode is activated by the 00 encoding and the three test-mode encodings have at least one test-mode signal asserted.

Example 36 Specifying User-Defined Binary Test-Mode Encodings

```
set_dft_signal -view spec -port {TM1 TM0} -type TestMode
define_test_mode LONG -encoding {TM1 0 TM0 1}
define_test_mode MEDIUM -encoding {TM1 1 TM0 0}
define_test_mode SHORT -encoding {TM1 1 TM0 1}
```

If you are providing your own one-hot encodings, configure the test-mode decoding to `one_hot` to build simplified one-hot decoding logic. **Example 37** shows how to configure one-hot encodings for the three example test modes.

Example 37 Specifying User-Defined One-Hot Test-Mode Encodings

```
set_dft_signal -view spec -port {TM2 TM1 TM0} -type TestMode
set_dft_configuration -mode_decoding_style one_hot
define_test_mode LONG -encoding {TM2 0 TM1 0 TM0 1}
define_test_mode MEDIUM -encoding {TM2 0 TM1 1 TM0 0}
define_test_mode SHORT -encoding {TM2 1 TM1 0 TM0 0}
```

If your test-mode control signals come from internal design pins, such as the outputs of test control registers, use the `-hookup_pin` option from the internal pins flow to make the connections. **Example 38** shows how to hook up two test-mode signals to corresponding control register output pins.

Example 38 Using Internal Design Pins for Test-Mode Signals

```
set_dft_signal -view spec -type TestMode \
    -port TM1 -hookup_pin TESTCTL_reg[1]/Q
set_dft_signal -view spec -type TestMode \
    -port TM0 -hookup_pin TESTCTL_reg[0]/Q

define_test_mode LONG -encoding {TM1 0 TM0 1}
define_test_mode MEDIUM -encoding {TM1 1 TM0 0}
define_test_mode SHORT -encoding {TM1 1 TM0 1}
```

If the internal pins do not directly correspond to top-level ports, you must use the internal pins flow. For more information, see [Internal Pins Flow on page 429](#).

Applying Test Specifications to a Test Mode

After you have defined test modes, you can apply mode-specific test specifications to each test mode. Not all commands and options can be used to apply mode-specific test

specifications. For more information about the available commands and options, see [Supported Test Specification Commands for Test Modes on page 367](#).

When you apply test specifications before defining any test modes, they are applied to the default test mode. As new test modes are defined, they inherit the test specification settings from this default mode. You can use this behavior to predefined global test specifications shared by all modes.

After you define a test mode with the `define_test_mode` command, that test mode becomes the *current test mode*. Subsequent scan specification commands apply only to that test mode. You can use this behavior to implicitly apply mode-specific test specifications after defining a test mode, as shown in [Example 39](#).

Example 39 Applying Scan Specifications Using Implicitly Defined Current Test Mode

```
define_test_mode LONG      ;# long scan chains
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks

define_test_mode MEDIUM    ;# medium scan chains
set_scan_configuration -chain_count 3 -clock_mixing mix_clocks

define_test_mode SHORT     ;# short scan chains
set_scan_configuration -chain_count 5
```

You can also use the `current_test_mode` command to change the current test mode at any time, as shown in [Example 40](#).

Example 40 Applying Scan Specifications Using Explicitly Defined Current Test Mode

```
define_test_mode LONG
define_test_mode MEDIUM
define_test_mode SHORT

current_test_mode LONG      ;# long scan chains
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks

current_test_mode MEDIUM    ;# medium scan chains
set_scan_configuration -chain_count 3 -clock_mixing mix_clocks

current_test_mode SHORT     ;# short scan chains
set_scan_configuration -chain_count 5
```

You can use the `-test_mode` option to directly apply a scan specification to a particular test mode at any time, regardless of the current test mode, as shown in [Example 41](#).

Example 41 Applying Scan Specifications Using the `-test_mode` Option

```
set_scan_configuration -test_mode LONG \
                      -chain_count 2 -clock_mixing mix_clocks
set_scan_configuration -test_mode MEDIUM \
                      -chain_count 3 -clock_mixing mix_clocks
```

```
set_scan_configuration -test_mode SHORT \
    -chain_count 5
```

To apply a scan specification to all test modes after previously defining some test modes, use the `-test_mode all` option, as shown in [Example 42](#).

Example 42 Applying Scan Specification to All Test Modes

```
set_dft_signal -view spec -test_mode all \
    -type ScanDataIn -port {SI1 SI2 SI3}
```

Note:

The `-test_mode` option also accepts the `all_dft` keyword, which is equivalent to `all`.

When multiple scan specification commands are applied to a test mode, they are applied cumulatively. A new scan specification command overwrites a previous scan specification according to the same precedence rules used in a single test-mode flow.

[Example 43](#) shows how to apply clock mixing to all test modes except for the `SHORT` test mode:

Example 43 Overwriting a Previous Scan Specification Setting for a Test Mode

```
set_scan_configuration -test_mode all \
    -clock_mixing mix_clocks
set_scan_configuration -test_mode LONG \
    -chain_count 2
set_scan_configuration -test_mode MEDIUM \
    -chain_count 3
set_scan_configuration -test_mode SHORT \
    -chain_count 5 -clock_mixing no_mix ;# overwrites mix_clocks
```

The default for the `-clock_mixing` option is `no_mix`. In this example, the first command applies the `-clock_mixing no_clocks` option to all test modes. The subsequent two commands configure the chain counts for the first two test modes. Because the `-chain_count` option does not overwrite the `-clock_mixing` option, the `mix_clocks` specification remains in place for the `LONG` and `MEDIUM` test modes. In the last command, the `-clock_mixing` option reapplies the default clock mixing behavior of `no_mix` to the `SHORT` test mode to overwrite the `mix_clocks` behavior previously applied to all test modes.

Recommended Ordering of Global and Mode-Specific Commands

When applying scan specifications in a multiple test-mode environment, perform these steps:

1. Define TestMode signals with the `set_dft_signal -test_mode all` command.
2. Define all test modes, their usage, and optional encodings with the `define_test_mode` command.
3. Define clock and asynchronous DFT signals and constants that are common to all test modes with the `set_dft_signal -test_mode all` command.
4. Define any mode-specific DFT signals with the `set_dft_signal -test_mode test_mode_name` command.
5. Specify any scan specifications to be used in all test modes using the `-test_mode all` option of the `set_scan_configuration` and `set_scan_path` commands.
6. Specify any scan specifications to be used in specific test modes using the `-test_mode test_mode_name` option of the `set_scan_configuration` and `set_scan_path` commands.

Note:

If you reference any test modes using the `-test_mode` option of any DFT configuration commands, you must define those test modes with the `define_test_mode` command before referencing them.

Using Multiple Test Modes in Hierarchical Flows

In hierarchical flows, different cores might have different test modes defined. In this case, DFT Compiler creates as many top-level test modes as needed to accommodate all of the core-level test modes.

By default, the relationship between core-level and top-level test modes is determined by test mode name according to the following rules:

- For each core-level test mode, a top-level test mode of the same name is created.
- If multiple cores share a test mode with the same name, those core-level test modes are included in a top-level test mode of the same name.
- If a core does not have a test mode that matches a top-level test mode name, it is excluded from that top-level test mode.

The `preview_dft` and `insert_dft` commands report the core-level test modes used in each of the top-level test modes.

Consider an example where coreA has two test modes named SHORT and MEDIUM, and coreB has two test modes named MEDIUM and LONG. At the top level, DFT Compiler creates all three test modes. The `preview_dft` and `insert_dft` commands report the core-level test modes as shown in [Example 44](#).

Example 44 Top-Level Test Mode Report for Default Name-Based Relationships

```
Control signal value - Integration Test Mode
Core Instance - Test Mode
-----
00 SHORT - InternalTest
    coreA - SHORT: InternalTest

01 MEDIUM - InternalTest
    coreA - MEDIUM: InternalTest
    coreB - MEDIUM: InternalTest

10 LONG - InternalTest
    coreB - LONG: InternalTest
```

Note that coreB does not participate in top-level test mode SHORT, and coreA does not participate in top-level test mode LONG.

At the top level, you can override the default name-based association of core-level test modes by using the `define_test_mode -target` command. The `-target` option takes a list of core and test mode pairs that should be included in that top-level test mode.

The previous example can be modified to use the closest matches for the missing core-level test modes, as shown in [Example 45](#).

Example 45 Specifying User-Defined Core-Level Test Mode Relationships

```
# top-level test mode definitions
define_test_mode SHORT \
    -target {coreA:SHORT    coreB:MEDIUM}
define_test_mode MEDIUM \
    -target {coreA:MEDIUM   coreB:MEDIUM}
define_test_mode LONG \
    -target {coreA:MEDIUM   coreB:LONG}
```

The `preview_dft` and `insert_dft` commands report the core-level test modes as shown in [Example 46](#).

Example 46 Top-Level Test Mode Report for User-Defined Test Mode Relationships

```
Control signal value - Integration Test Mode
Core Instance - Test Mode
-----
00 SHORT - InternalTest
    coreA - SHORT: InternalTest
    coreB - MEDIUM: InternalTest
```

```

01 MEDIUM - InternalTest
coreA - MEDIUM: InternalTest
coreB - MEDIUM: InternalTest

10 LONG - InternalTest
coreA - MEDIUM: InternalTest
coreB - LONG: InternalTest

```

Note:

If you use the `-target` option of the `define_test_mode` command, you must completely define the core test mode relationships for all cores and test modes. When the `-target` option is used, name-based test mode association is no longer performed for any core or test mode.

Supported Test Specification Commands for Test Modes

This topic lists the commands and options you can use to configure DFT insertion for specific test modes. These commands and options honor the `-test_mode` option or the current test-mode focus. Other DFT configuration commands and options apply to all test modes.

set_dft_signal

You can use the `set_dft_signal -test_mode` command to declare different DFT signals for different test modes. For example, each test mode can have different scan-in and scan-out ports.

Keep in mind that pre-DFT DRC only analyzes the global `all` test mode; it does not consider mode-specific signals applied to other modes. As a result,

- Mode-specific signals defined with the `-view spec` option, such as scan-in and scan-out signals, can be safely made, as these signals do not yet exist during pre-DFT DRC.
- Mode-specific signals defined with the `-view existing` option, such as constants and reset signals, must be made with care, as these signals are not considered during pre-DFT DRC. However, they are incorporated into the mode-specific test protocols used by post-DFT DRC.

Note that you can apply a baseline set of signals to the `all` test mode to be used by pre-DFT DRC, along with mode-specific signals to be used by post-DFT DRC.

All test modes must share the same scan-enable signals. You cannot specify different scan-enable signals for different test modes.

If the `-test_mode` option is not specified, this command applies to the current test mode.

set_scan_configuration

The following `set_scan_configuration` options can be applied to specific test modes:

- `-chain_count`
- `-clock_mixing`
- `-exclude_elements`

Note:

Shared wrapper cells are not supported for per-test-mode exclusion.

- `-max_length`

If the `-test_mode` option is not specified, these options apply to the current test mode. Other options of the `set_scan_configuration` command apply to all test modes.

set_scan_path

The following `set_scan_path` options can be applied to specific test modes:

- `-scan_master_clock`
- `-exact_length`

Use the `set_scan_path -test_mode` command to provide scan chain specifications for the test modes in your design. The scan path specification can be given for any chains in any defined test mode. It can include pin access information provided with the `-scan_data_in`, `-scan_data_out`, `-scan_enable`, `-scan_master_clock`, and `-scan_slave_clock` options. The specification can also specify a list of design elements to be included. When specifying pins with the `-scan_data_in` or `-scan_data_out` options, the signals must be previously defined with the `set_dft_signal` command using the `ScanDataIn` or `ScanDataOut` signal types. If the scan path specification applies to a test mode which has the usage specified as `scan`, both the port and hookup arguments of the `set_dft_signal` command can be specified for the `ScanDataIn` and `ScanDataOut` signals. If the scan path specification applies to a test mode which has the usage specified as `scan_compression`, then only the `-hookup` option of the `set_dft_signal` command can be specified for the `ScanDataIn` and `ScanDataOut` signals. A port argument must not be used for compressed scan mode scan path definitions, as these codec-connected compressed chains have no direct access from the ports.

If the `-test_mode` option is not specified, the specification applies to the current test mode. To apply the specification to all test modes, you must use the `-test_mode all` option.

Multiple Test-Mode Scan Insertion Script Examples

This topic provides examples of basic scan, DFTMAX compressed scan, and core wrapping scripts in a multiple test-mode environment.

[Example 47](#) 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 47 Basic Scan Multiple Test-Mode Script

```
## Define the pins for use in any test mode with "-test_mode all"
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 -verbose

current_test_mode scan2
dft_drc -verbose

current_test_mode scan3
dft_drc -verbose

current_test_mode scan4
dft_drc -verbose

list_test_modes

list_licenses
change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output vg/top_scan.v
write_test_protocol -test_mode scan1 \
    -output stil/scan1.stil -names verilog
write_test_protocol -test_mode scan2 \
    -output stil/scan2.stil -names verilog
write_test_protocol -test_mode scan3 \
    -output stil/scan3.stil -names verilog
write_test_protocol -test_mode scan4 \
    -output stil/scan4.stil -names verilog

exit

```

Example 48 shows a DFTMAX compression 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 48 Basic DFTMAX Compressed Scan Multiple Test-Mode Script

```

## Define the pins for compression/base_mode using "-test_mode all"
## 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 DFTMAX compression
set_dft_configuration -scan_compression enable

#Configure DFTMAX compression
set_scan_compression_configuration -base_mode my_base1 -chain_count 32 \
    -test_mode scan_compression1 -xtolerance high
set_scan_compression_configuration -base_mode my_base2 -chain_count 256 \
    -test_mode scan_compression2 -xtolerance 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 -verbose

current_test_mode scan_compression2
report_dft_signal
dft_drc -verbose

current_test_mode my_base1
report_dft_signal
dft_drc -verbose

current_test_mode my_base2
report_dft_signal
dft_drc -verbose

current_test_mode burn_in
report_dft_signal
dft_drc -verbose

change_names -rules verilog -hierarchy
write -format verilog -hierarchy \
    -output vg/10x_xtol_moxie_top_scan_mm.v
write_test_protocol -test_mode scan_compression1 \
    -output stil/scan_compression1.stil -names verilog
write_test_protocol -test_mode scan_compression2 \
    -output stil/scan_compression2.stil -names verilog
write_test_protocol -test_mode my_base1 \
    -output stil/my_base1.stil -names verilog
write_test_protocol -test_mode my_base2 \

```

```

        -output stil/my_base2.stil -names verilog
write_test_protocol -test_mode burn_in \
        -output stil/10x_xtol_moxie.burn_in.stil -names verilog

exit

```

Example 49 shows a core wrapper script. This script defines all the modes that are created in a wrapper insertion process, which supports at-speed test and shared wrapper cells.

Example 49 Basic Core Wrapper Multiple Test-Mode 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 -verbose

#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_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_burn_in.rpt

current_test_mode domain
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_domain.rpt

current_test_mode my_wrp_of
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_of.rpt

current_test_mode my_wrp_of_delay
report_scan_path -view existing_dft -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_dft -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_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_my_wrp_if.rpt

current_test_mode my_wrp_if_delay

```

```

report_scan_path -view existing_dft -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_dft -cell all > \
    reports xg_wrap_dedicated_delay_my_wrp_if_scl_delay.rpt

report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchy -output db/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_model -output db/des_unit.scan.ctldb
write_test_protocol -test_mode burn_in -output stil/burn_in.spf
write_test_protocol -test_mode domain -output stil/domain.spf
write_test_protocol -test_mode my_wrp_if_delay \
    -output stil/my_wrp_if_delay.spf
write_test_protocol -test_mode my_wrp_if_scl_delay \
    -output stil/my_wrp_if_scl_delay.spf
write_test_protocol -test_mode my_wrp_if -output stil/my_wrp_if.spf
write_test_protocol -test_mode my_wrp_of -output stil/my_wrp_of.spf
write_test_protocol -test_mode my_wrp_of_delay \
    -output stil/my_wrp_of_delay.spf
write_test_protocol -test_mode my_wrp_of_scl_delay \
    -output stil/my_wrp_of_scl_delay.spf
write_test_protocol -test_mode wrp_if -output 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 50](#).

Example 50 report_scan_path Output Example

```

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)

```

Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces

Normally, the test mode of a design is controlled by one or more test-mode ports. You can use the IEEE 1500 insertion feature to control the test mode of a design through standard IEEE 1500 or IEEE 1149.1 interfaces instead. This feature allows you to

- Insert IEEE 1500 test-mode control logic, which provides a standard interface for test mode control, at the core level and chip level
- Insert server control logic, which provides complete access to all on-chip IEEE 1500 controllers through an IEEE Std 1149.1 (JTAG) interface, at the chip level
- Integrate cores that use either IEEE 1500 or test-mode ports for test mode control
- Create test protocols that set up a design's test mode using the core-level or chip-level logic at that level

Note:

A DFTMAX or TestMAX DFT license is required to use the IEEE 1500 test-mode control feature.

Test-mode control through IEEE 1500 is described in the following topics:

- [IEEE 1500 Test Mode Control Architecture](#)
- [Inserting IEEE 1500 at the Core Level](#)
- [Inserting IEEE 1500 and IEEE 1149.1 at the Chip Level](#)
- [Customizing the IEEE 1500 Architecture](#)
- [Writing Test Protocols](#)
- [Script Examples](#)
- [Limitations](#)

IEEE 1500 Test Mode Control Architecture

The architecture used for test-mode control through IEEE 1500 depends on whether you are inserting control logic at the core level or chip level, and whether you are integrating cores, as described in the following topics:

- [Core-Level Test-Mode Control](#)
- [Core Integration With IEEE 1500 Test-Mode Control](#)
- [Chip-Level Test-Mode Control](#)

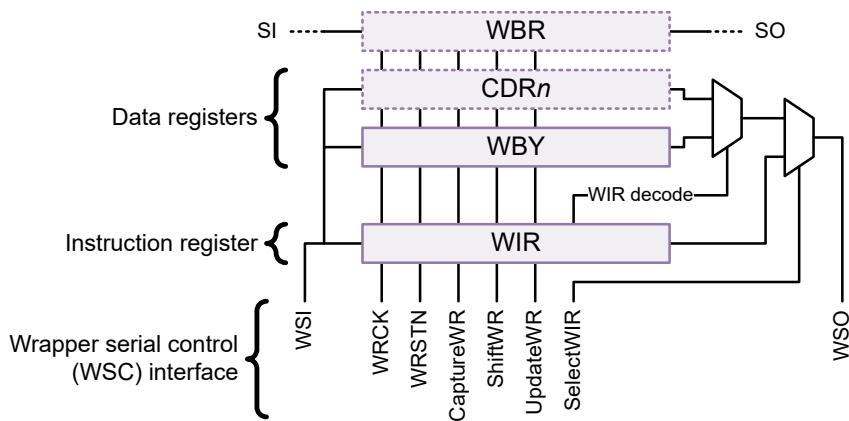
For more information about IEEE 1500, see *IEEE Std 1500-2005 - IEEE Standard Testability Method for Embedded Core-based Integrated Circuits*, available at the following address:

<http://standards.ieee.org/>

Core-Level Test-Mode Control

An IEEE 1500 controller presents a standardized interface for accessing the test capabilities of a design. Figure 143 shows the basic implementation used by the TestMAX DFT tool.

Figure 143 IEEE 1500 Controller Logic



The IEEE 1500 controller logic is accessed primarily through a mandatory serial interface, known as the wrapper serial control (WSC) interface. The WSC interface uses the following wrapper controller signals to provide access to the instruction and data registers:

- WSI - scan path input
- WRSTN - active-low reset
- WRCK - controller clock
- CaptureWR - data capture enable
- ShiftWR - shift enable
- UpdateWR - update enable
- SelectWIR - shift path selection between controller instruction or data registers
- WSO - scan path output

These signals provide access to the instruction and data registers. When SelectWIR is asserted, a new instruction can be shifted into the WIR. When SelectWIR is de-asserted,

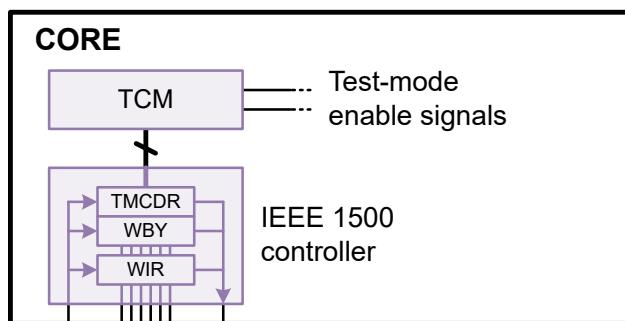
a new data value can be shifted into the data register selected by the WIR. A single-bit wrapper bypass register (WBY) is required for compliance.

The wrapper boundary register (WBR) represents the wrapper chain present in wrapped cores. It can be implemented as one or more register segments. Typically a minimum of two segments are used: one for inputs and one for outputs.

The IEEE 1500 specification also allows one or more core data registers (CDRs) to be implemented. These CDRs allow design-specific functionality to be accessed from the WSC.

When IEEE 1500 test-mode control is enabled, DFT insertion creates a CDR to control the test mode. This test-mode CDR (TMCDR) drives the test control module (TCM), as shown in [Figure 144](#). The TMCDR in the IEEE 1500 controller takes the place of the test-mode input ports that typically drive the test-mode signals. By shifting different values into the TMCDR, the core can be placed into different test modes.

Figure 144 Core-Level IEEE 1500 Test-Mode Control



In addition to driving the test-mode control signals of the TCM, the TMCDR also drives on-chip clocking (OCC) controller enable and PLL bypass signals.

Note:

If you have test-mode signals that control other testability features, such as AutoFix or clock gating, you must define them as port-driven signals. They are not controlled by the TMCDR.

Note the following aspects of the DFTMAX implementation of the IEEE 1500 controller:

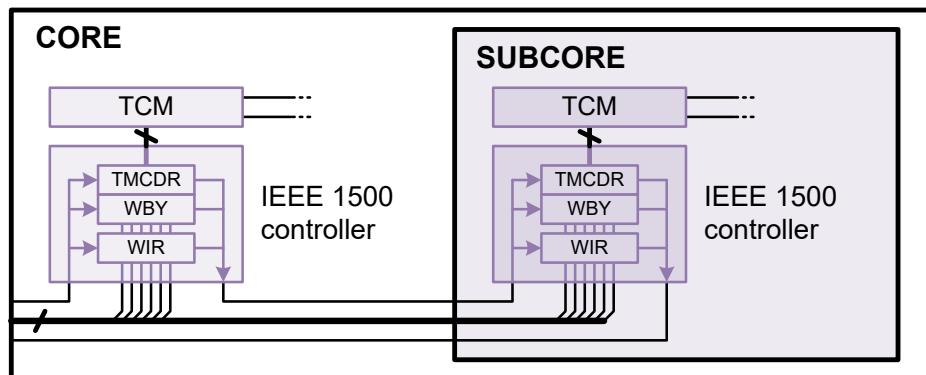
- The WBR is optional; you can enable IEEE 1500 test-mode control when creating unwrapped cores.
- The WSC interface provides the WBR shift, capture, and update control signals, but normal scan input and output signals provide the WBR shift data. This allows the WBR to be controlled using the normal WSC signals, while still allowing the WBR to use scan compression.

Core Integration With IEEE 1500 Test-Mode Control

When you use IEEE 1500 test-mode control, you can integrate cores with or without IEEE 1500 test-mode control.

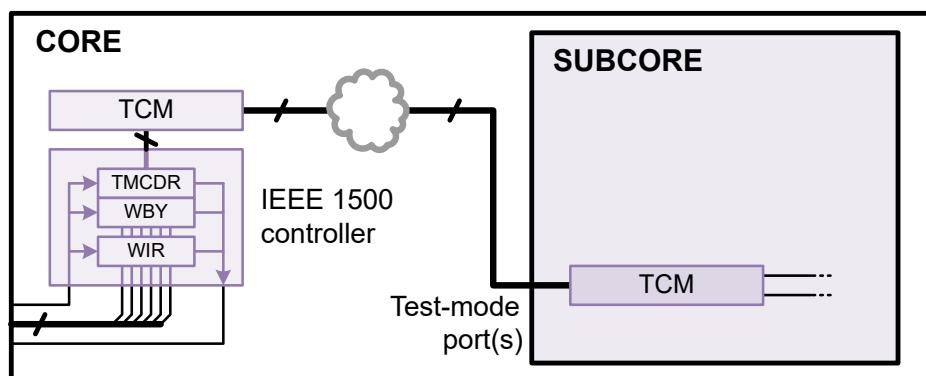
When you integrate a core with IEEE 1500 test-mode control, the IEEE 1500 controller inside that core is daisy-chained with the IEEE 1500 controller in the current design, as shown in [Figure 145](#).

Figure 145 Integrating a Core That Uses IEEE 1500 Test-Mode Control



When you integrate a core with test-mode ports, the test-mode ports of that core are driven by the test control module (TCM) of the current design, as shown in [Figure 146](#). Glue logic might be added, depending on how the core-level test modes map to the test modes of the current design.

Figure 146 Integrating a Core That Uses Test Mode Ports



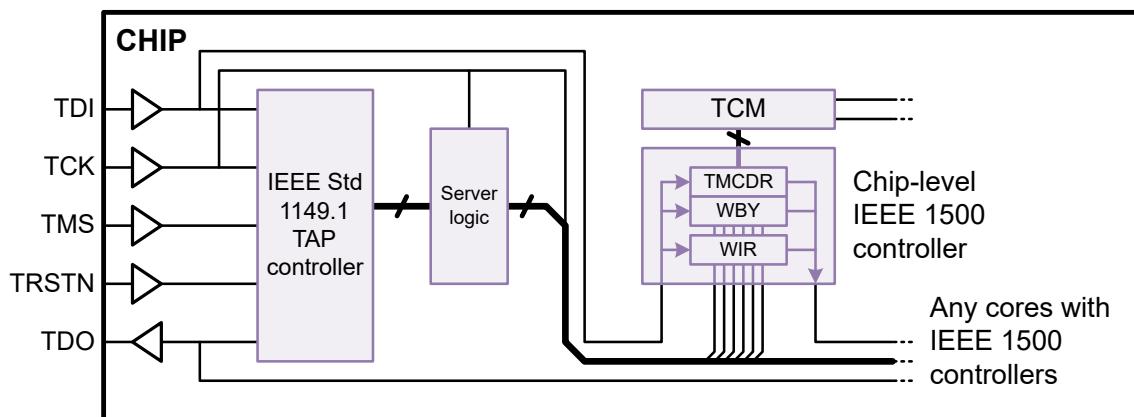
You can integrate a mix of cores with and without IEEE 1500 test-mode control. All cores with IEEE 1500 test-mode control are daisy-chained off the IEEE 1500 controller of the current design, and all cores with test-mode ports are driven by the TCM of the current design.

When you use IEEE 1500 test-mode control for a core, you must also use IEEE 1500 test-mode control for each higher level of DFT insertion up to and including the chip level.

Chip-Level Test-Mode Control

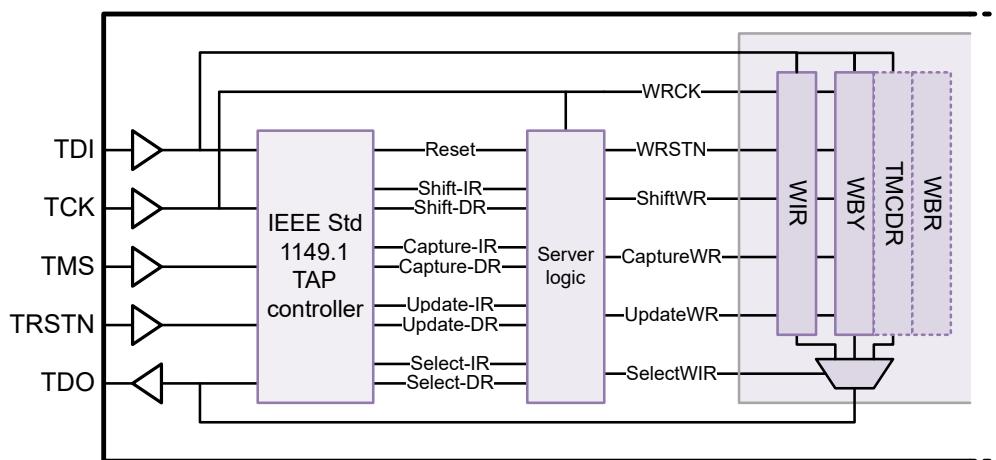
At the chip level, the tool inserts an IEEE 1500 controller and integrates all cores, just as at the core level. In addition, it inserts glue logic, called *server logic*, that connects the IEEE 1500 controller logic to the chip-level IEEE Std 1149.1 interface, as shown in [Figure 147](#).

[Figure 147](#) Integrating a Core That Uses IEEE 1500 Test-Mode Control



By design, the control and data signals used by IEEE 1500 are similar to those used by IEEE Std 1149.1. [Figure 148](#) shows a summary of the signal connectivity provided by the server logic.

[Figure 148](#) Summary of Control and Data Signals Used by the Server Logic



Inserting IEEE 1500 at the Core Level

To use a DFT-inserted IEEE 1500 controller for test-mode control at the core level,

1. Enable IEEE 1500 test-mode controller insertion with the following command:

```
dc_shell> set_dft_configuration -ieee_1500 enable
```

The tool automatically recognizes cores with IEEE 1500 test-mode control logic; you do not need to specify them.

2. If you have existing placeholder ports for the IEEE 1500 wrapper serial control (WSC) interface signals, define them with the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type WSI -port port
dc_shell> set_dft_signal -view spec -type WRSTN -port port \
           -active_state 0
dc_shell> set_dft_signal -view spec -type WRCK -port port
dc_shell> set_dft_signal -view spec -type CaptureWR -port port
dc_shell> set_dft_signal -view spec -type ShiftWR -port port
dc_shell> set_dft_signal -view spec -type UpdateWR -port port
dc_shell> set_dft_signal -view spec -type SelectWIR -port port
dc_shell> set_dft_signal -view spec -type WSO -port port
```

If you do not define these signals, the tool creates them automatically.

3. When you define your DFT signals, do not define the following test-mode signals:

- Test-mode signals for multiple test-mode selection
- On-chip clocking (OCC) controller test-mode signals and PLL bypass signals

When the tool automatically creates these signals, it creates test-mode core data register (TMCDR) bits to drive them. These TMCDR bits take the place of traditional port-driven test mode signals.

Note:

If you have test-mode signals that control other testability features, such as AutoFix or clock gating, define them in the usual way. They are not controlled by the TMCDR.

4. Continue with DFT insertion in the usual way.

After DFT insertion, the tool places information about the IEEE 1500 interface signals into the CTL model. This allows the interface to be recognized during chip-level integration.

Inserting IEEE 1500 and IEEE 1149.1 at the Chip Level

To use a DFT-inserted IEEE 1500 controller for test-mode control at the chip level where IEEE Std 1149.1 logic will also be inserted,

1. Enable IEEE 1500 test-mode controller insertion and IEEE Std 1149.1 insertion with the following command:

```
dc_shell> set_dft_configuration -ieee_1500 enable -bsd enable
```

The tool automatically recognizes cores that have IEEE 1500 test-mode control logic described in their CTL models; you do not need to specify these cores.

2. Define the ports or pad hookup pins for the IEEE Std 1149.1 interface signals with the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type TDI -port port \
           -hookup_pin pin
dc_shell> set_dft_signal -view spec -type TRSTN -port port \
           -hookup_pin pin -active_state 0
dc_shell> set_dft_signal -view spec -type TCK -port port \
           -hookup_pin pin
dc_shell> set_dft_signal -view spec -type TMS -port port \
           -hookup_pin pin
dc_shell> set_dft_signal -view spec -type TDO -port port \
           -hookup_pin pin
dc_shell> set_dft_signal -view spec -type TDO_EN -port port \
           -hookup_pin pin
```

All signals except TRSTN are required. If these signals do not exist, the tool does not automatically create them.

3. When you define your DFT signals, do not define the following test-mode signals:

- Test-mode signals for multiple test-mode selection
- On-chip clocking (OCC) controller test-mode signals and PLL bypass signals

When the tool automatically creates these signals, it creates test-mode core data register (TMCDR) bits to drive them. These TMCDR bits take the place of traditional port-driven test mode signals.

Note:

If you have test-mode signals that control other testability features, such as AutoFix or clock gating, define them in the usual way. They are not controlled by the TMCDR.

4. Continue with DFT insertion and boundary-scan insertion in the usual way.

For more information about the boundary-scan insertion capability provided by the TestMAX DFT tool, see the *TestMAX DFT Boundary Scan User Guide*.

Note:

When IEEE 1500 test-mode control is enabled, the DFT-inserted TAP controller implements only the instructions needed for IEEE 1500 control. To implement mandatory IEEE Std 1149.1 instructions, boundary-scan logic, or user-defined instructions, use the 2-pass method described in [SolvNet article 039402, “How Can I Use Additional Boundary-Scan Features With IEEE 1500 Test-Mode Control?”](#)

Customizing the IEEE 1500 Architecture

In the IEEE 1500 architecture, test-mode signals are generated from wrapper instruction register (WIR) and test-mode core data register (TMCDR) decoding logic. This topic describes the options used to configure this decoding logic.

The following topics describe how you can customize the IEEE 1500 architecture:

- [Configuring the WIR](#)
- [Configuring the DFT-Inserted TMCDR](#)
- [Using an Existing TMCDR](#)
- [Using WIR Test-Mode Decoding With No TMCDR](#)
- [Controlling the Test-Mode Encoding Style](#)
- [Reporting the Test Mode Encodings](#)
- [Specifying WIR Opcodes for CDRs](#)

Configuring the WIR

By default, the tool sizes the WIR automatically, based on the opcodes that select the core data registers. For the default case, which is a TMCDR with no user-defined CDRs, the tool uses a single-bit WIR. If you define additional CDRs or specify particular opcodes to be used, the tool creates a wider WIR as needed.

To specify a particular WIR size, use the following command:

```
dc_shell> set_ieee_1500_configuration -wir_width width
```

Configuring the DFT-Inserted TMCDR

By default, the tool sizes the TMCDR to contain all test-mode encodings using the specified encoding mode. To specify a wider TMCDR size, use the `-exact_length` option. For example,

```
dc_shell> set_scan_path TMCDR -class ieee_1500 \
           -view spec -test_mode all \
           -exact_length tmcdr_width
```

The `-class 1500` option indicates that you are defining a CDR register (in this case, the TMCDR) that is selected by the WIR. The scan path name can be anything except the reserved values of `WIR` and `WBY`.

Using an Existing TMCDR

Instead of using a DFT-inserted TMCDR, you can use an existing TMCDR for test-mode control of your design.

[Example 51](#) defines an existing TMCDR.

Example 51 Defining an Existing Test-Mode Core Data Register

```
# define the control and scan-in/scan-out signal pins of the TMCDR
set_dft_signal -view spec -type WSI -hookup_pin MY_TMCDR/wsi
set_dft_signal -view spec -type WRCK -hookup_pin MY_TMCDR/wrck
set_dft_signal -view spec -type WRSTN -hookup_pin MY_TMCDR/wrstn \
               -active_state 0
set_dft_signal -view spec -type ShiftWR -hookup_pin MY_TMCDR/shiftwr
set_dft_signal -view spec -type CaptureWR -hookup_pin MY_TMCDR/capturewr
set_dft_signal -view spec -type UpdateWR -hookup_pin MY_TMCDR/updatewr
set_dft_signal -view spec -type WSO -hookup_pin MY_TMCDR/wso

# define the TMCDR scan path design
set_scan_path MY_TMCDR_SCANPATH -class ieee_1500 \
               -view spec -test_mode all \
               -ordered_elements {MY_TMCDR/DOUT[3] \
                                  MY_TMCDR/DOUT[2] \
                                  MY_TMCDR/DOUT[1] \
                                  MY_TMCDR/DOUT[0]} \
               -hookup {MY_TMCDR/wsi \
                        MY_TMCDR/wrck \
                        MY_TMCDR/wrstn \
                        MY_TMCDR/shiftwr \
                        MY_TMCDR/capturewr \
                        MY_TMCDR/updatewr \
                        MY_TMCDR/wso}

# prevent scan insertion/stitching on the CDR block
set_scan_element false MY_TMCDR
set_scan_configuration -exclude_elements MY_TMCDR
```

Specify the leaf or hierarchical register output pins with the `-ordered_elements` option, ordered from WSI to WSO. Specify the remaining input and output control and scan data pins used to access the TMCDR with the `-hookup` option. All pins should be unconnected or tied to ground because the tool connects them during DFT insertion. (The signal and scan path specifications are defined with the `-view spec` option because the tool makes these connections during DFT insertion.)

The following signal types are required for an existing TMCDR specification: WSI, WRCK, ShiftWR, WSO. Other signal types are optional.

To use specific TMCDR register output bits for specific DFT signals, specify the register outputs as hookup pins with the `set_dft_signal` command. For example,

```
set_dft_signal -view spec -type pll_bypass \
    -hookup_pin MY_TMCDR/DOUT[3]
set_dft_signal -view spec -type pll_reset \
    -hookup_pin MY_TMCDR/DOUT[2]
set_dft_signal -view existing_dft -type Constant -active_state 0 \
    -hookup_pin MY_TMCDR/DOUT[0]
```

You can specify hookup pins for some or all of the DFT signals in the design. Unspecified DFT signals will use the remaining available TMCDR bits in the usual way.

If the existing TMCDR does not contain enough bits for all DFT signals in the design, the tool creates and uses a DFT-inserted TMCDR for the additional signals.

Using WIR Test-Mode Decoding With No TMCDR

By default, the tool uses a TMCDR, selected by the WIR, to generate the test-mode signals. The advantage of this method is that only a single WIR instruction is consumed by test functionality; all of the test-mode encodings are contained in the TMCDR data encoding space.

However, you can omit the TMCDR register and use the WIR directly for test-mode decoding. In this case, WIR instruction encodings are created for all test modes. To do this, define the TMCDR with a zero width specified:

```
dc_shell> set_scan_path TMCDR -class ieee_1500 \
    -view spec -test_mode all \
    -exact_length 0
```

When the TMCDR is omitted, DFTMAX sizes the WIR to contain all test-mode encodings using the specified encoding mode. To specify a wider WIR size, use the following command:

```
dc_shell> set_ieee_1500_configuration -wir_width width
```

Controlling the Test-Mode Encoding Style

The tool automatically chooses encodings for each test mode that can be enabled by the TMCDR. By default, binary encodings are used, which provide the most compact encodings but require the use of decoding logic to generate the test-mode enable signals.

You can also specify one-hot test-mode encoding, which requires simplified decoding logic at the expense of more test-mode encoding bits, by using the following command:

```
dc_shell> set_dft_configuration -mode_decoding_style one_hot
```

Reporting the Test Mode Encodings

When the tool determines the WIR opcode that selects the TMCDR, the `preview_dft` and `insert_dft` commands report the chosen opcode as follows:

```
Info: CDR Opcode is set to '1' (WSO-->WSI)
```

The `preview_dft` command also provides information about the test-mode encodings. The following report example is for a design using the default TMCDR test-mode decoding mode:

```
dc_shell> preview_dft
...
=====
Test Mode Controller Information
=====

Test Mode Controller Ports
-----
test_mode: BLK_Test_Controller_1500_inst/CDR[1]
test_mode: BLK_Test_Controller_1500_inst/CDR[0]

Test Mode Controller Index (WSO --> WSI)
-----
BLK_Test_Controller_1500_inst/CDR[1],
BLK_Test_Controller_1500_inst/CDR[0]

Control signal value - Test Mode
-----
00 Mission_mode - Normal
01 wrp_of - ExternalTest
10 wrp_if - InternalTest
11 ScanCompression_mode - InternalTest
```

The following report example is for a design using the optional WIR test-mode decoding mode:

```
dc_shell> preview_dft
...
Test Mode Controller Ports
-----
```

```

test_mode: BLK_Test_Controller_1500_inst/WIR[1]
test_mode: BLK_Test_Controller_1500_inst/WIR[0]

Test Mode Controller Index (WSO --> WSI)
-----
BLK_Test_Controller_1500_inst/WIR[1],
BLK_Test_Controller_1500_inst/WIR[0]

Control signal value - Test Mode
-----
00 Mission_mode - Normal
01 wrp_of - ExternalTest
10 wrp_if - InternalTest
11 ScanCompression_mode - InternalTest

```

Specifying WIR Opcodes for CDRs

By default, the tool chooses WIR opcodes for all CDRs selectable by the WIR. This includes

- The WBY register
- TMCDR registers
- User-defined CDR registers

To specify the WIR opcode that selects a particular CDR, use the `-opcode` option of the `set_scan_path` command.

For the default DFT-inserted TMCDR that does not normally have a scan path specification, specify only the opcode with no other information. For example,

```
dc_shell> set_scan_path TMCDR -class ieee_1500 \
           -view spec -test_mode all \
           -opcode opcode_string
```

For a CDR that already has a scan path specification, include the `-opcode` option in the specification.

If you have not specified the WIR size, the tool sizes the WIR to accommodate your opcodes. If you have specified the WIR size, choose your opcode encodings accordingly.

Writing Test Protocols

When you perform DFT insertion, the tool creates a test protocol for each newly created test mode. When you use IEEE 1500 test-mode control, each test protocol configures the test mode through the appropriate interface—IEEE 1500 at the core level and IEEE Std 1149.1 at the chip level. You can use the `write_test_protocol` command to write out these test protocols in STIL format.

[Example 52](#) shows the test_setup section of a core-level test protocol that initializes the test mode through the IEEE 1500 interface.

Example 52 Core-Level Test Protocol Using IEEE 1500 Test-Mode Control

```
"test_setup" {
    ...
    Ann { * Reset 1500 Logic *} V {
        "WSI" = 0;
        "WRSTN" = 0;
        "SelectWIR" = 0;
    }
    Ann { * Unreset 1500 Logic *} V {
        "WRSTN" = 1;
    }
    Ann { * Prepare To Load WIR *} V {
        "WRCK" = P;
        "ShiftWR" = 1;
        "SelectWIR" = 1;
    }
    Ann { * Load 1500 WIR *} Macro "wir_load";
    Ann { * Update 1500 WIR *} V {
        "ShiftWR" = 0;
        "UpdateWR" = 1;
    }
    Ann { * Prepare To Load CDR *} V {
        "ShiftWR" = 1;
        "UpdateWR" = 0;
        "SelectWIR" = 0;
    }
    Ann { * Load 1500 CDR *} Macro "cdr_load";
    Ann { * Update 1500 CDR *} V {
        "ShiftWR" = 0;
        "UpdateWR" = 1;
    }
    Ann { * 1500 Wrapper Clock Off *} V {
        "WRCK" = 0;
        "UpdateWR" = 0;
    }
}
```

[Example 53](#) shows the test_setup section of a chip-level test protocol that initializes the test mode through the IEEE Std 1149.1 interface. The test_setup sections of the test protocols use macros to simplify the initialization of the WIR and CDR.

Example 53 Chip-Level Test Protocol Using IEEE 1500 Test-Mode Control

```
"test_setup" {
    ...
    Ann { * Test-Logic-Reset *} V {
        "tdi" = 0;
        "trstn" = 0;
    }
```

```

}
Macro "reset_to_shift_ir";
Macro "SELECT_WIR";
Macro "exit1_ir_to_shift_dr";
Macro "wir_load_CORE2_ScanCompression_mode";
Macro "wir_load_CORE1_ScanCompression_mode";
Macro "wir_load_top_CORES1AND2";
Macro "exit1_dr_to_shift_ir";
Macro "SELECT_CDR";
Macro "exit1_ir_to_shift_dr";
Macro "cdr_load_CORE2_ScanCompression_mode";
Macro "cdr_load_CORE1_ScanCompression_mode";
Macro "cdr_load_top_CORES1AND2";
Ann { * Update-DR *} V {
    "tck" = P;
    "tms" = 1;
    "trstn" = 1;
}
Ann { * Run-Test-Idle *} V {
    "tms" = 0;
}
Ann { * JTAG TCK Off *} V {
    "tck" = 0;
}
}

```

If a test mode contains an untested core with an IEEE 1500 controller, the corresponding test protocol loads the WS_BYPASS instruction into the wrapper instruction register (WIR) of the untested core.

Script Examples

[Example 54](#) shows a script that inserts compressed scan, core wrapping, and IEEE 1500 test-mode control logic at the core level.

Example 54 Core-Level Insertion of IEEE 1500 Test-Mode Control

```

# initial compile
current_design core
link
create_clock -period 10 CLK
compile -scan

# enable DFT clients
set_dft_configuration \
    -scan_compression enable -wrapper enable -ieee_1500 enable

# define DFT signals
set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port CLK
set_dft_signal -view spec -type ScanDataIn -port [get_ports SI*]

```

```

set_dft_signal -view spec -type ScanDataOut -port [get_ports SO*]

set_dft_signal -view spec -type WSI           -port WSI
set_dft_signal -view spec -type WRSTN        -port WRSTN
set_dft_signal -view spec -type WRCK          -port WRCK
set_dft_signal -view spec -type CaptureWR    -port CaptureWR
set_dft_signal -view spec -type ShiftWR       -port ShiftWR
set_dft_signal -view spec -type UpdateWR      -port UpdateWR
set_dft_signal -view spec -type SelectWIR     -port SelectWIR
set_dft_signal -view spec -type WSO           -port WSO

# configure scan, scan compression, core wrapping
set_scan_configuration -chain_count 2
set_scan_compression_configuration -chain_count 8
set_wrapper_configuration -class core_wrapper -maximize_reuse enable

# insert DFT and write out design
create_test_protocol
dft_drc
insert_dft
write -format ddc -output core.ddc

```

Example 55 shows a script that inserts IEEE 1500 test-mode control, along with the server and IEEE Std 1149.1 TAP controller logic, at the chip level.

Example 55 Chip-Level Insertion of IEEE 1500 Test-Mode Control

```

# initial compile
current_design chip
link
compile -scan -incremental

# enable DFT clients
set_dft_configuration \
    -bsd enable -scan enable -scan_compression enable -ieee_1500 enable

# define DFT signals
set_dft_signal -view existing_dft -type ScanClock \
    -port TCK -timing {45 55}
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}

for {set i 1} {$i <= 2} {incr i} {
    set_dft_signal -view spec -type ScanDataIn \
        -port SI${i} -hookup_pin U_SI${i}_PAD/Z
    set_dft_signal -view spec -type ScanDataIn \
        -port SI${i} -hookup_pin U_SI${i}_PAD/Z
}
set_dft_signal -view spec -type ScanEnable -port SE -hookup_pin U_SE/Z

set_dft_signal -view spec -type TDI      -port TDI      -hookup_pin U_TDI/Z
set_dft_signal -view spec -type TRST    -port TRST_N   -hookup_pin U_TRSTN/Z
set_dft_signal -view spec -type TCK      -port TCK      -hookup_pin U_TCK/Z

```

```

set_dft_signal -view spec -type TMS      -port TMS      -hookup_pin U_TMS/Z
set_dft_signal -view spec -type TDO      -port TDO      -hookup_pin U_TDO/A
set_dft_signal -view spec -type TDO_EN   -port TDO      -hookup_pin U_TDO/E

# configure wrapped compressed scan core integration
define_test_mode CORES_STD -usage scan \
    -target {CORE1:wrp_if}                  CORE2:wrp_if}
define_test_mode CORES_COMP -usage scan_compression \
    -target {CORE1:ScanCompression_mode CORE2:ScanCompression_mode}
define_test_mode ONLY_TOP -usage scan \
    -target {chip}

set_scan_configuration -test_mode CORES_STD \
    -chain_count 4 -clock_mixing mix_clocks
set_scan_configuration -test_mode ONLY_TOP \
    -chain_count 4 -clock_mixing mix_clocks
set_scan_compression_configuration \
    -test_mode CORES_COMP -base_mode CORES_STD \
    -integration_only true

# insert DFT and write out design
create_test_protocol
dft_drc
insert_dft
change_names -rules verilog -hierarchy
write -format ddc -output chip.ddc

```

Limitations

Note the following requirements and limitations:

- When IEEE 1500 test-mode control is enabled, the DFT-inserted TAP controller implements only the instructions needed for IEEE 1500 control. To implement mandatory IEEE Std 1149.1 instructions, boundary-scan logic, or user-defined instructions, use the 2-pass method described in [SolvNet article 039402, “How Can I Use Additional Boundary-Scan Features With IEEE 1500 Test-Mode Control?”](#)
- The chip level must have multiple test modes to control.
- At least one core must have IEEE 1500 logic inserted.
- You can only integrate cores containing IEEE 1500 logic up one level, into a chip level that uses IEEE 1500 and IEEE Std 1149.1 test-mode control. Nested integration of IEEE 1500 cores is not supported.
- The test-mode control data register (TMCDR) drives only the test-mode selection signals and the OCC controller test-mode and pll_bypass signals. If you have test-mode signals that control other testability features, such as AutoFix or clock gating, you must define them as port-driven signals. They are not controlled by the TMCDR.

- All IEEE 1500 signal and scan specifications must be applied to the global test mode; specifications applied to specific user-defined test modes are ignored.
- The WBR boundary scan register does not support WSI-to-WSO scan shifting through the IEEE 1500 controller.
- You cannot specify user-defined test mode encodings when using DFT-inserted IEEE 1500 test-mode control.

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 topic describes the methodology for running DFT insertion in designs containing multiple voltages.

The following topics describe multivoltage support:

- [Configuring Scan Insertion for Multivoltage Designs](#)
- [Configuring Scan Insertion for Multiple Power Domains](#)
- [Mixture of Multivoltage and Multiple Power Domain Specifications](#)
- [Reusing Multivoltage Cells](#)
- [Scan Path Routing and Isolation Strategy Requirements](#)
- [Using Domain-Based Strategies for DFT Insertion](#)
- [DFT Considerations for Low-Power Design Flows](#)
- [Previewing a Multivoltage Scan Chain](#)
- [Scan Extraction Flows in the Presence of Isolation Cells](#)
- [Limitations](#)

Configuring Scan Insertion for Multivoltage Designs

The following command instructs the DFT insertion process to build scan chains that can cross different voltage regions:

```
dc_shell> set_scan_configuration -voltage_mixing true
```

When set to `true`, DFT insertion attempts to minimize voltage crossings to reduce the number of level shifters added. The default of this option is `false`.

If you use any commands, such as the `set_scan_path` command, 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 can cross different power domains:

```
dc_shell> set_scan_configuration \
           -power_domain_mixing true
```

When set to `true`, DFT insertion attempts to minimize power domain crossings to reduce the number of isolation cells added. DFT insertion does not check or remove existing isolation cells. The default of this option is `false`.

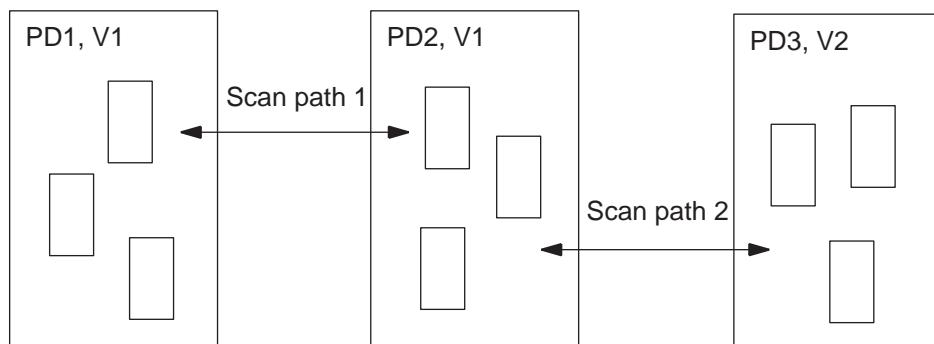
If you use any commands, such as the `set_scan_path` command, that violate the power domain mixing specification, the `preview_dft` and `insert_dft` commands issue the following warning message and continue with scan insertion:

Warning: elements are supplied by different power domains. (TEST-1029)

Mixture of Multivoltage and Multiple Power Domain Specifications

The interaction between the `-voltage_mixing` and `-power_domain_mixing` options is as shown in [Figure 149](#).

Figure 149 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 44](#) shows the allowable scan paths with the various combinations 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

Note:

The behavior of DFT insertion in an always-on synthesis environment is such that you must run an incremental compile after running the `insert_dft` command to insert any missing multivoltage cells that might be needed.

Reusing Multivoltage Cells

By default, the `insert_dft` command reuses existing level shifters and isolation cells if they are on the scan path. It reuses only combinational multivoltage cells and does not reuse sequential multivoltage cells, such as latch-based isolation cells. If you do not want the `insert_dft` command to reuse existing multivoltage cells (level shifters and isolation cells), use the following command:

```
dc_shell> set_scan_configuration \
           -reuse_mv_cells false
```

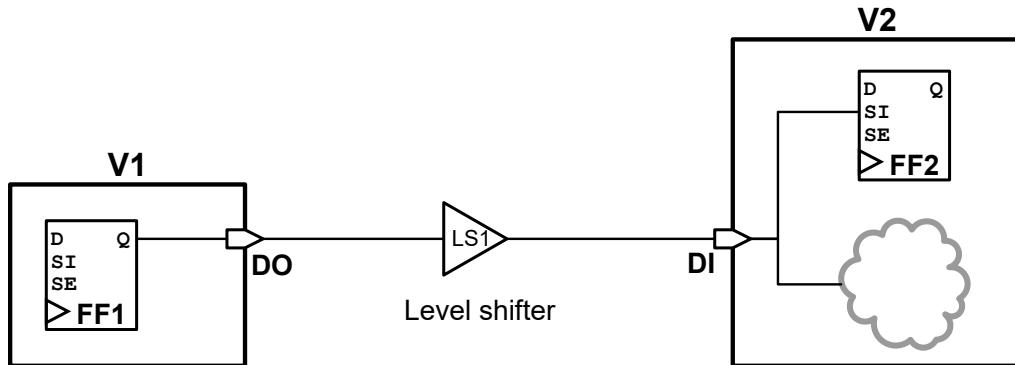
This topic covers the following:

- [Reusing Level Shifters in Scan Paths](#)
- [Reusing Isolation Cells in Scan Paths](#)

Reusing Level Shifters in Scan Paths

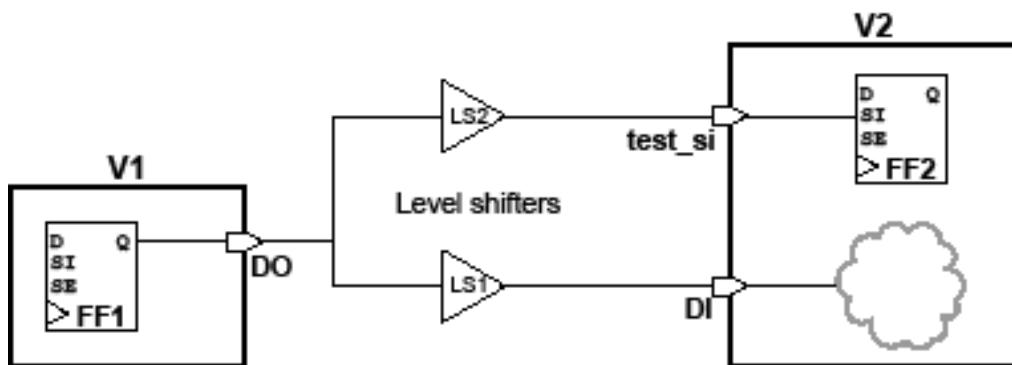
If a scan path goes through a level shifter and you enable multivoltage cell reuse, scan insertion connects the scan chain at the output side of the level shifter, as shown in [Figure 150](#).

Figure 150 Shared Level Shifter Along Scan Path



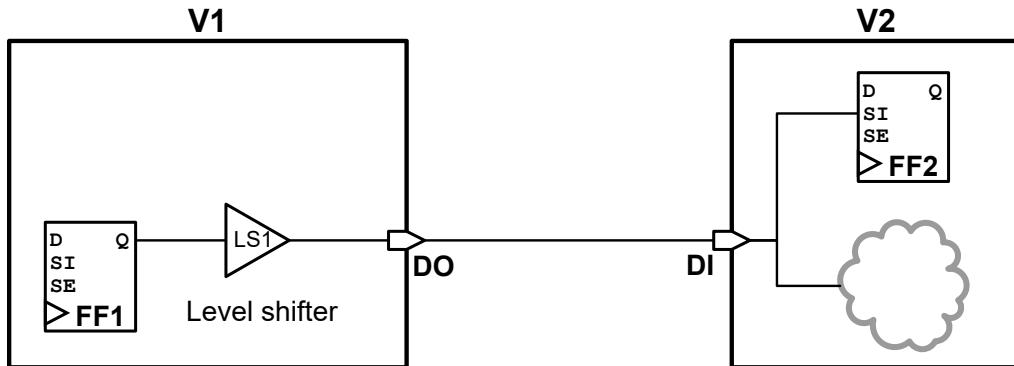
If the scan path goes through a level shifter and you disable multivoltage cell reuse, a new level shifter is added to connect to the scan path. See [Figure 151](#).

Figure 151 Separate Level Shifters Along Scan Path



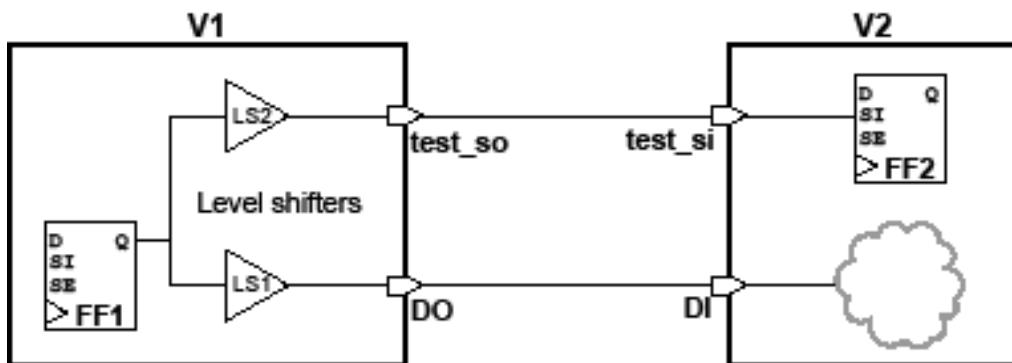
If the level shifter is within a block and you enable multivoltage cell reuse, then scan insertion reuses the existing level shifter and hierarchical port, as shown in [Figure 152](#).

Figure 152 Shared Level Shifter in a Block Along Scan Path



If you disable multivoltage cell reuse 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 153](#).

Figure 153 Separate Level Shifters in a Block Along Scan Path

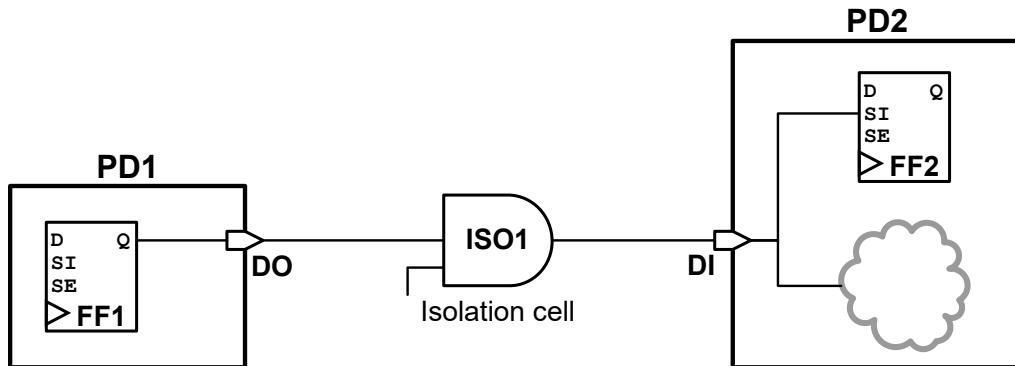


Reusing Isolation Cells in Scan Paths

In the following examples, the scan path and functional path for a given signal route to the same downstream power domain, which is different from the source power domain.

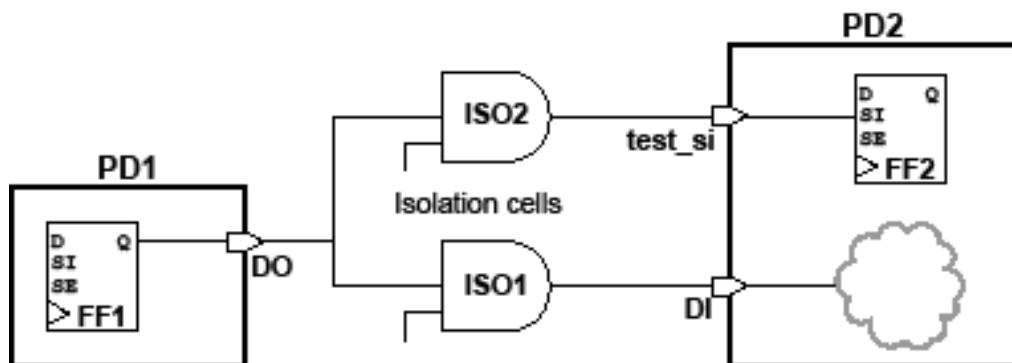
If a scan path goes through an isolation cell in a parent power domain, and you enable multivoltage cell reuse, then scan insertion connects the scan chain at the output side of the isolation cell, as shown in [Figure 154](#).

Figure 154 Shared Isolation Cell Along Scan Path in Parent Domain



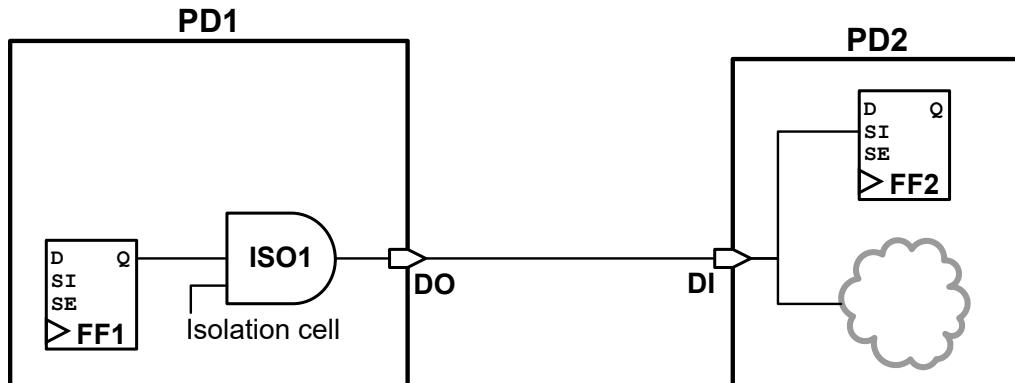
If you disable multivoltage cell reuse, then the scan path is connected to the net before the isolation cell and a new hierarchical port and isolation cell are added, as shown in [Figure 155](#).

Figure 155 Separate Isolation Cells Along Scan Path in Parent Domain



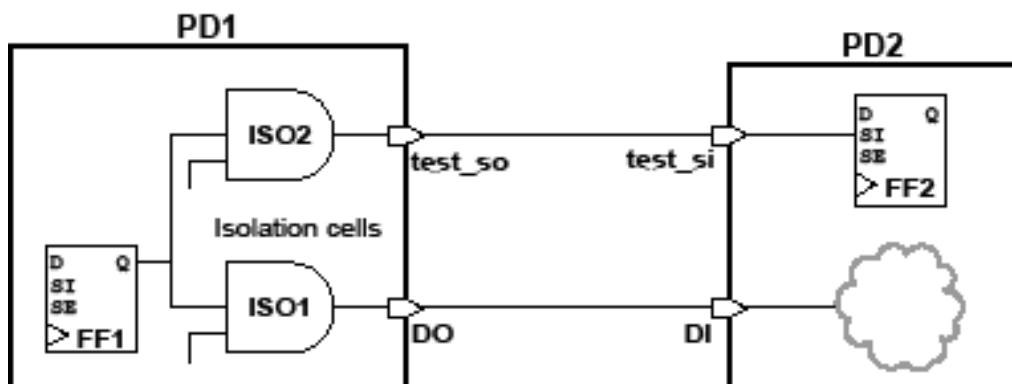
If the isolation cell is within the source power domain, and you enable multivoltage cell reuse, then scan insertion reuses the existing hierarchical port and isolation cell, as shown in [Figure 156](#).

Figure 156 Shared Isolation Cell Along Scan Path in Source Domain



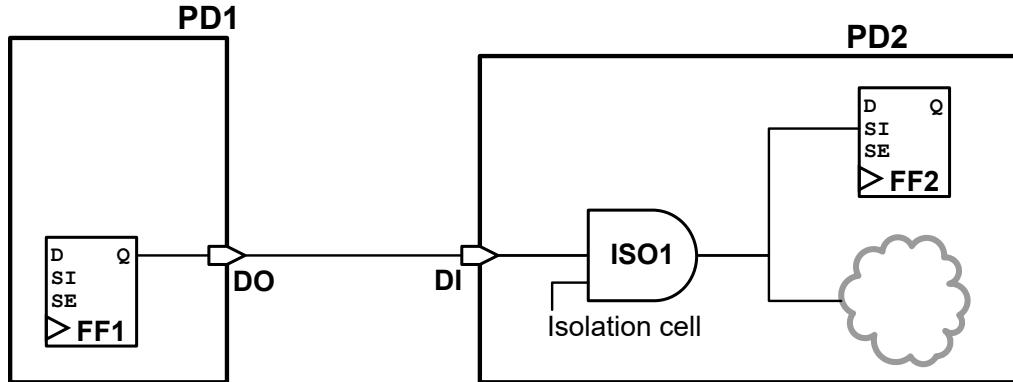
If you disable multivoltage cell reuse, and the existing isolation cell is within the source power domain, then a new hierarchical port and isolation cell are added, as shown in [Figure 157](#).

Figure 157 Separate Isolation Cells Along Scan Chain in Source Domain



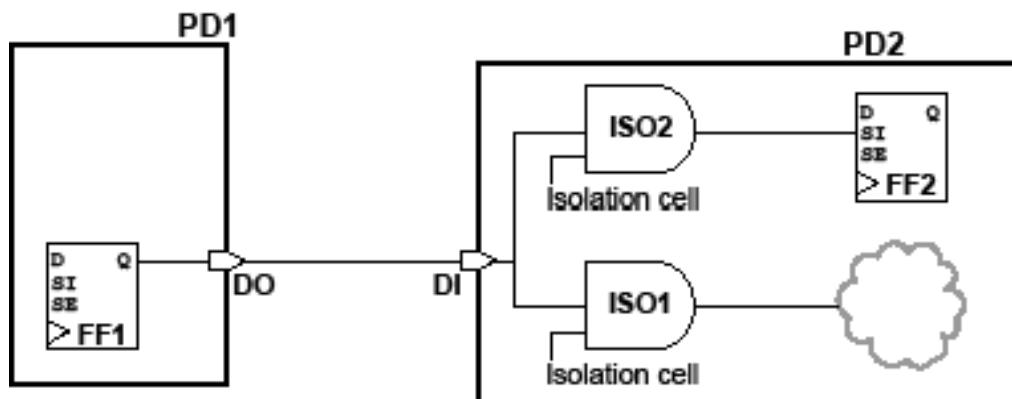
If the isolation cell is within the fanout power domain, and you enable multivoltage cell reuse, then scan insertion reuses the existing hierarchical port and isolation cell, as shown in [Figure 158](#).

Figure 158 Shared Isolation Cell Along Scan Path in Fanout Domain



If you disable multivoltage cell reuse, and the existing isolation cell is within the fanout power domain, then a new hierarchical port and isolation cell are added, as shown in Figure 159.

Figure 159 Separate Isolation Cells Along Scan Path in Fanout Domain



If the scan path routes to a different downstream power domain than the functional path, an existing isolation cell can be reused if the isolation strategy allows both the original power domain connection and the new scan path power domain connection to be driven by the isolation cell.

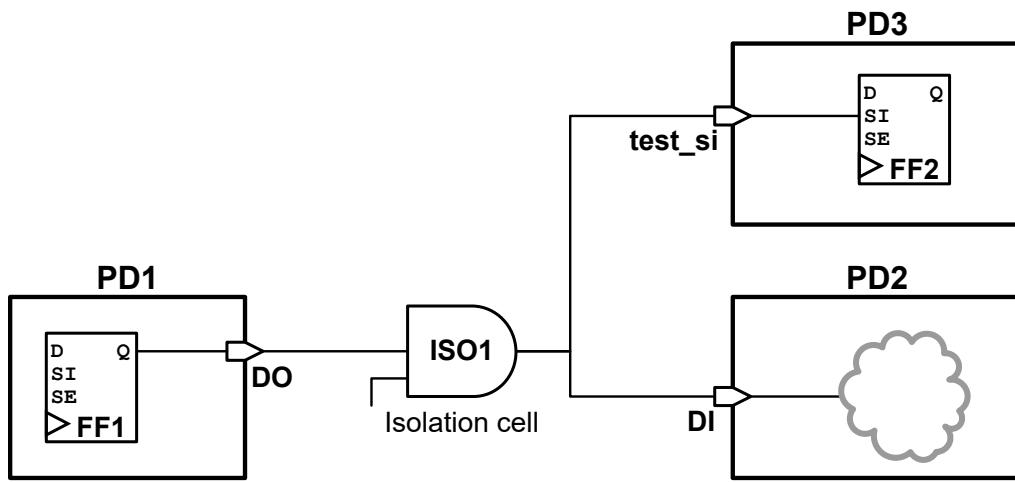
Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added to the parent power domain of PD1 with the `-location parent` option:

```
set_isolation iso_PD1 \
  -domain PD1 \
  -diff_supply_only true \
  -applies_to outputs
set_isolation_control iso_PD1 \
```

```
-domain PD1 \
-location parent
```

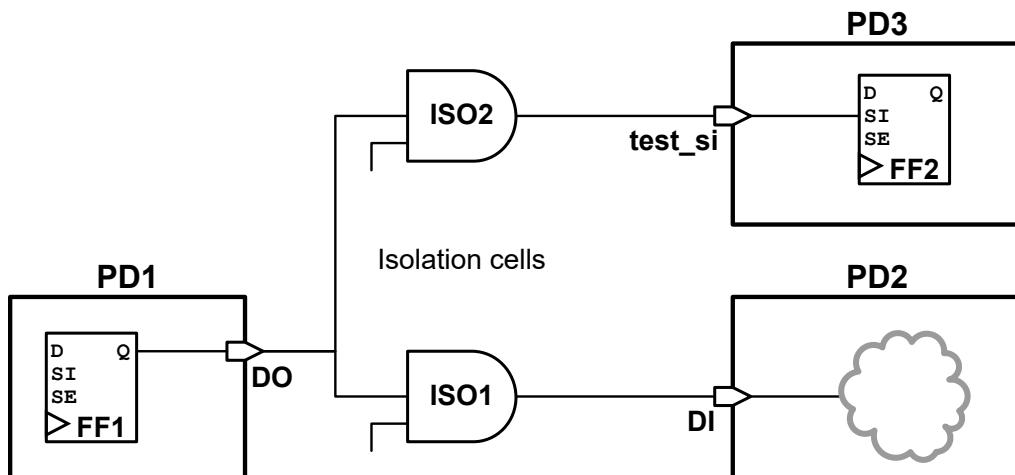
The `-diff_supply_only` strategy allows an isolation cell in the source domain to isolate multiple fanout power domains, if they differ from the source domain. If you enable multivoltage cell reuse, then scan insertion reuses the existing isolation cell and hierarchical port, as shown in [Figure 160](#).

Figure 160 Shared Isolation Cell in Parent Domain With Multiple Sink Domains



If you disable multivoltage cell reuse, then a new isolation cell is added, as shown in [Figure 161](#):

Figure 161 Separate Isolation Cells in Parent Domain With Multiple Sink Domains

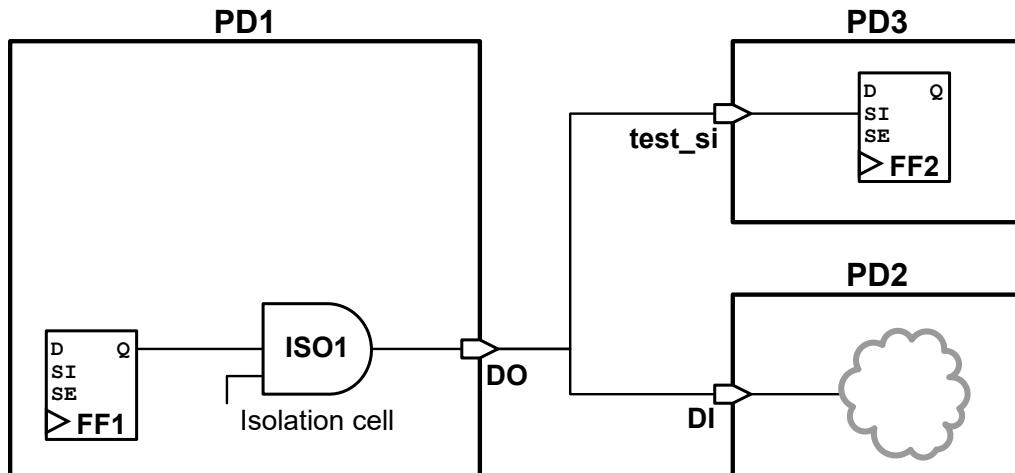


Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added within the power domain PD1 with the `-location self` option:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location self
```

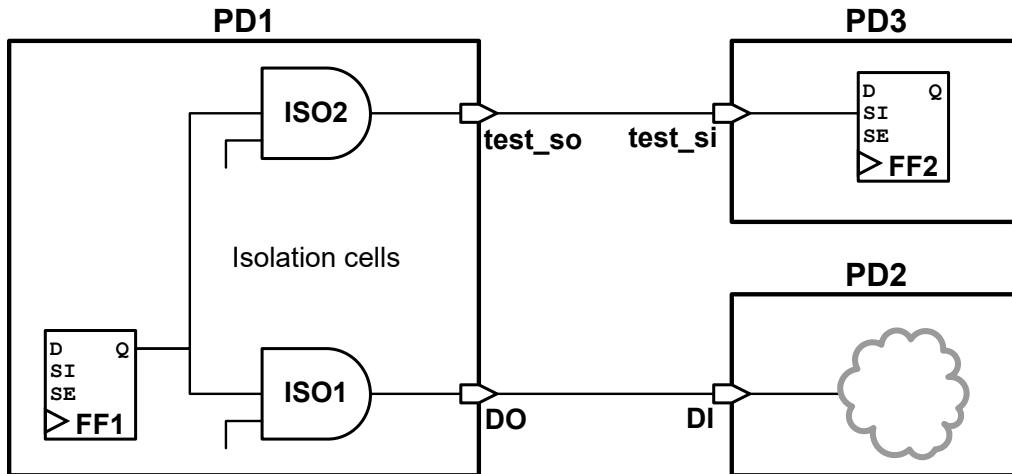
If you enable multivoltage cell reuse, then scan insertion reuses the existing isolation cell and hierarchical port, as shown in [Figure 162](#):

Figure 162 Shared Block Isolation Cell With Multiple Sink Domains



If you disable multivoltage cell reuse, and the existing isolation cell is within a block, then a new isolation cell and hierarchical port are added, as shown in [Figure 163](#):

Figure 163 Separate Parent Isolation Cells With Multiple Sink Domains



Scan Path Routing and Isolation Strategy Requirements

The isolation strategy applied to a cross-domain net might restrict the power domain connections allowed for that net:

- A `set_isolation -diff_supply_only` isolation strategy allows multiple fanout power domains to be driven by the same isolated net, if they differ from the source domain.
- A `set_isolation -source -sink` isolation strategy requires that any specified source and sink power domains connected to an isolated net match the isolation strategy.

When the `insert_dft` command routes a scan chain along an existing hierarchical output port to a different downstream power domain, the isolation strategy requirements of the existing net might require new isolation cells and hierarchical ports to be added along the scan path.

Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added to the parent power domain of PD1:

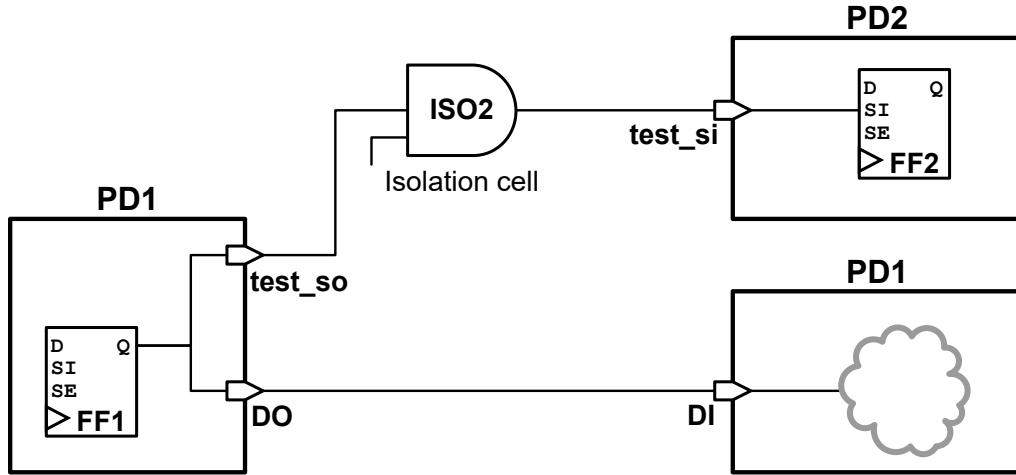
```

set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location parent

```

If the existing functional path is routed through the same power domain, but the scan path is routed to a different power domain, an isolation cell is added within the parent power domain as shown in Figure 164.

Figure 164 Isolation Cell in Parent Domain With Differing Isolation Requirements

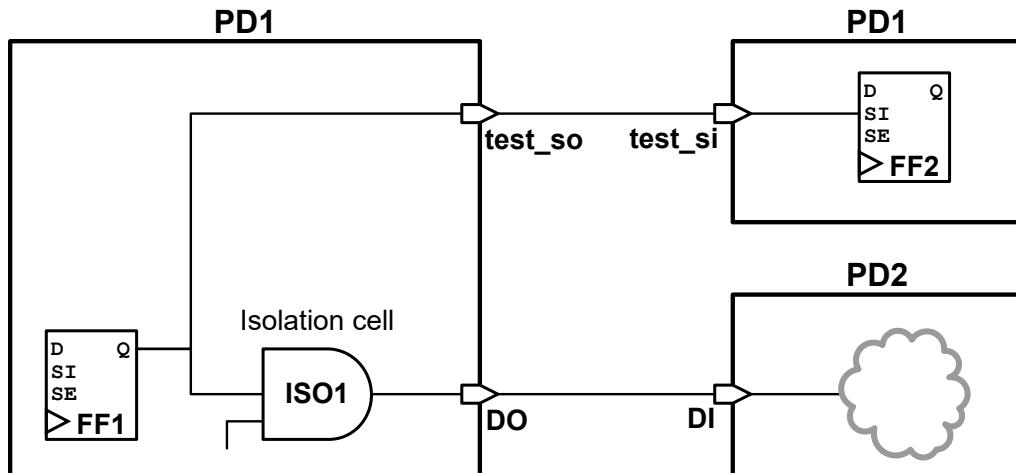


Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added within the source power domain PD1:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location self
```

If the existing functional path is routed to a different power domain but the scan path is routed through the same power domain, a hierarchical port is added to bypass the isolation cell as shown in Figure 165.

Figure 165 Isolation Cell in Source Domain With Differing Isolation Requirements

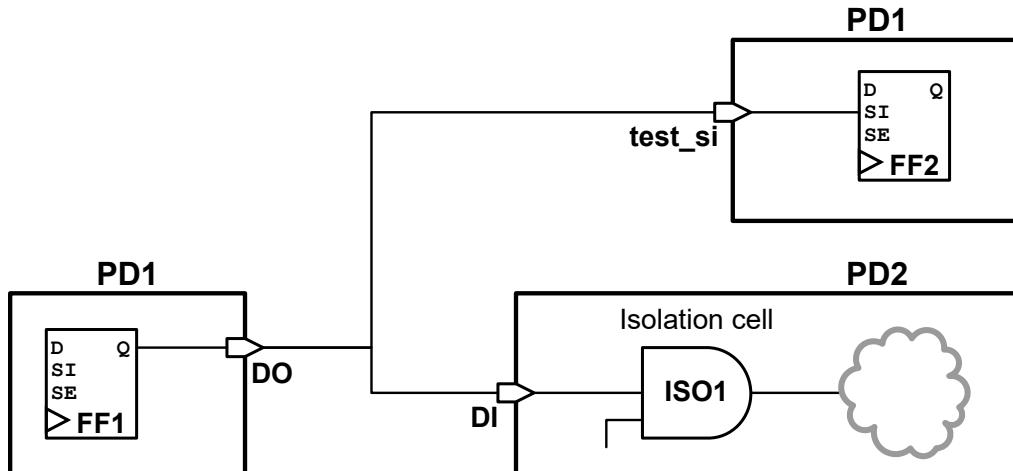


You can configure the isolation cell strategy to place the isolation cells in the fanout domains to avoid the creation of additional hierarchical scan pins. Consider the following example, modified to use the `-location fanout` option:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location fanout
```

In this modified example, the isolation cells and resulting isolated nets are now located in the fanout power domain, as shown in [Figure 166](#). No additional hierarchical scan pins are needed to meet the isolation strategy requirements.

Figure 166 Isolation Cell in Fanout Domain With Differing Isolation Requirements



Consider the `-source -sink` isolation strategy defined in the following example, which specifies two isolation strategies: one that requires isolation cells for nets that fan out to sink power domain SS2, and one that requires isolation cells for nets that fan out to sink power domain SS3.

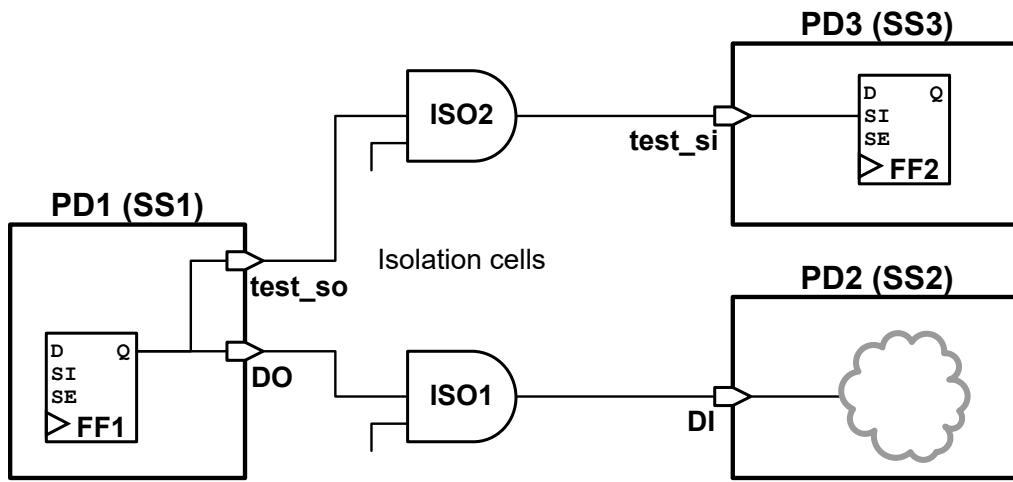
```
set_isolation iso1_PD1 \
    -domain PD1 \
    -source SS1 -sink SS2
set_isolation_control iso1_PD1 \
    -domain PD1 \
    -location parent

set_isolation iso2_PD1 \
    -domain PD1 \
    -source SS1 -sink SS3
set_isolation_control iso2_PD1 \
```

```
-domain PD1 \
-location parent
```

If you enable multivoltage cell reuse, the `insert_dft` command is unable to reuse the existing isolation cell ISO1 for the scan path connection, because the existing cross-domain isolated net cannot drive two different power domains. Instead, a new isolation cell and hierarchical port are added, as shown in [Figure 167](#).

Figure 167 Separate Isolation Cells in Parent Domain With Differing Isolation Requirements



Using Domain-Based Strategies for DFT Insertion

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 the blocks and individual ports. If you only specify the strategies on the blocks and 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 example, if your power intent specification is applied to all outputs with the `-applies_to outputs` option:

```
create_power_domain PD1 -elements U_block

set_isolation iso_PD1 \
    -domain PD1 \
    -isolation_power_net VDD -isolation_ground_net VSS \
    -clamp_value 1 \
    -applies_to outputs
```

and if the `insert_dft` command creates new output pins on the `blk_a` block that requires isolation, then isolation cells are automatically added where they are needed.

However, if your power intent specification is applied to selected existing design elements with the `-elements` option:

```
create_power_domain PD1 -elements U_block

set_isolation PD1 \
    -domain BLOCK \
    -isolation_power_net VDD -isolation_ground_net VSS \
    -clamp_value 1 \
    -elements {Z}
```

The isolation strategy applies only to existing output pin Z. Any new output pins that are created by the `insert_dft` command are not isolated.

The same behavior applies to level shifter insertion strategies specified by the `set_level_shifter` command.

DFT Considerations for Low-Power Design Flows

Low-power designs often use the following special cells:

- Isolation cells
- Retention registers
- Power switches

You must configure these special cells such that the data shifts through the scan chains during test operations. After DFT insertion using the `insert_dft` command, the `dft_drc` command can identify design rule violations on isolation cells and retention registers, if any, that would prevent scan shifting through such cells. However, the command cannot identify design rule violations on isolation cells and retention registers before the `insert_dft` command is run.

The control signals for these special cells are typically driven by a power controller. If the power controller is located off-chip, you can constrain the control signals at the ports for correct shift operation. If the power controller is located on-chip and does not include testability logic, the tool can insert power controller override logic for you. For more information, see [Inserting Power Controller Override Logic on page 411](#).

You must keep any DFT constraints on special cells in place up to the `write_scan_def` command when you generate a SCANDEF file for scan chain reordering.

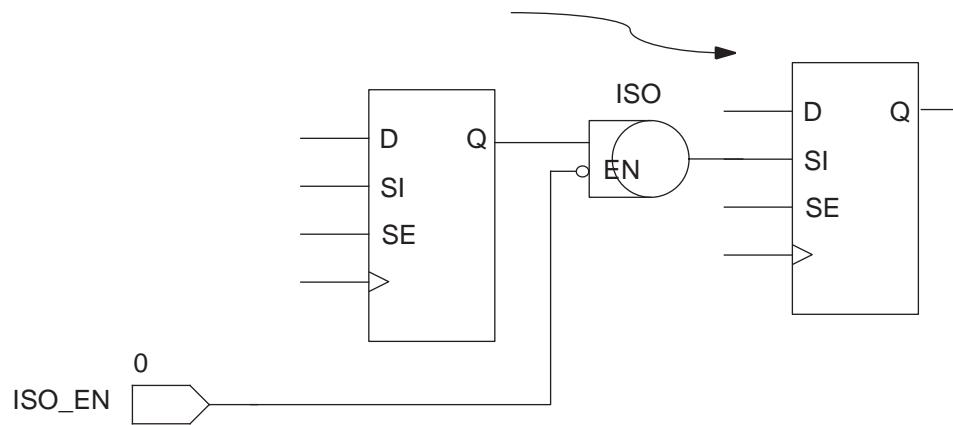
Also note that the `dft_drc` command cannot detect design rule violations on power switches in pre- or post-DFT insertion.

A multivoltage-aware verification tool such as MVSIM or MVRC can detect such violations if there are any. For further information, see the MVSIM and MVRC documentation.

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 168](#).

Figure 168 Proper Configuration of a Scan Chain That Includes an Isolation Cell



Retention Registers

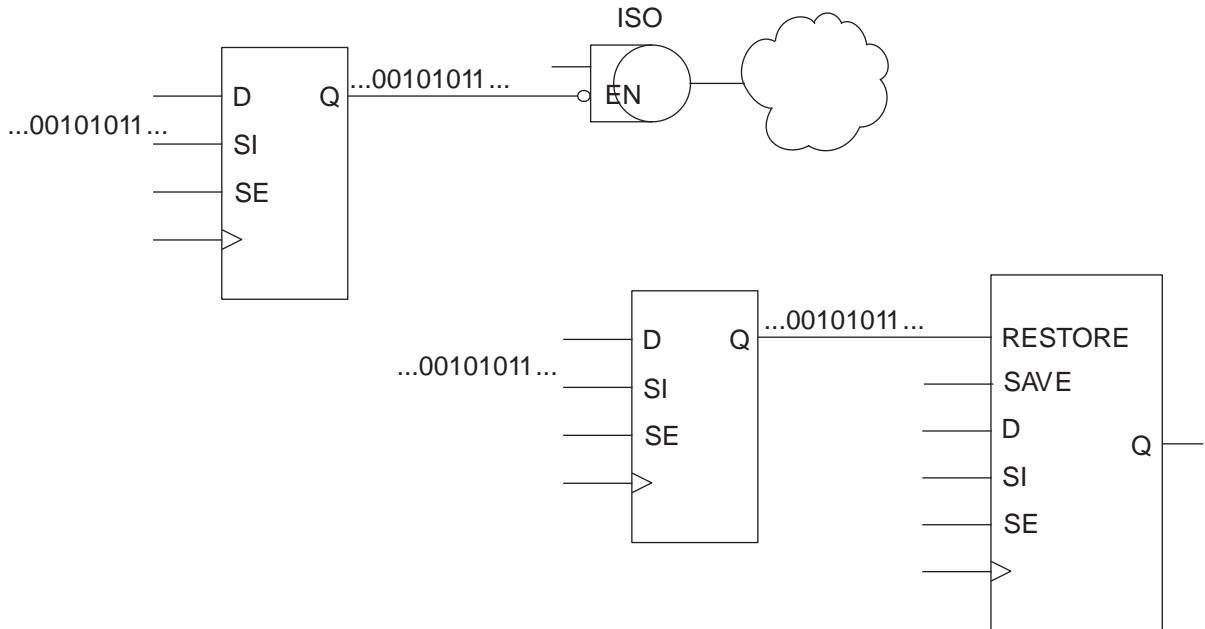
The design’s retention registers must be in normal or power-up mode during the test operation. You should check that any save or restore signals are at their correct states.

Registers That Drive 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 169](#) shows the consequences of putting such registers on the scan chains.

These registers must also drive the low power control signals to a constant state so that the controls cannot be toggled during test operations. You can check for this during the `dft_drc` command by ensuring that these registers are included in the TEST-504 and TEST-505 violations.

Figure 169 Consequences of Putting Registers That Drive Low-Power Control Signals on the Scan Chains



Previewing a Multivoltage Scan Chain

The `preview_dft -show all` command reports 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 56](#), (v) indicates that the scan cell drives a level shifter and (i) indicates that the scan cell drives an isolation cell.

Example 56 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]
...

```

See Also

- [Previewing the DFT Logic on page 603](#) for more information about previewing scan chain structures

Scan Extraction Flows in the Presence of Isolation Cells

If you need to run the scan extraction flow on a netlist for a design that is already scan-inserted and contains isolation cells, you must specify any constraints that are needed at the enable pin of isolation cells in addition to the constraints that might be required for DFT signals. If this is not done, you might fail to extract scan chains. DFT Compiler does not check the validity of isolation logic.

Limitations

The following limitations apply to multivoltage and multipower domains:

- 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, and HSS
 - DFTMAX compressed scan
- Multivoltage and multipower domains are not supported in the following flows:
 - BSD insertion
 - Core integration

Controlling Power Modes During Test

Power-sensitive designs often contain multiple power domains. This allows power supplies for inactive logic to be switched off, reducing power consumption during operation. These designs typically have a *power controller* block, which supplies the necessary power supply control signals to power switches, isolation cells, and retention registers.

When the device is being tested, the power control signals must be controlled to ensure that the design is testable. This topic describes the features provided by DFT Compiler to automate the control of power modes during test.

Inserting Power Controller Override Logic

The power controller block generates control signals for the power supply control logic that exists throughout the design. The power control signals at the block outputs are defined using commands that are part of the IEEE 1801 specification, also known as the Unified Power Format (UPF) specification.

DFT Compiler does not create the power controller block, but it can insert testability logic at the block outputs to override the power control signals during test mode. This logic is known as the *power controller override* logic. This logic allows the signals to be controlled during test mode without manually-created power controller initialization vectors. It also provides observability of the power controller outputs for improved test coverage.

To use this feature, enable the power controller override feature, and specify the power controller hierarchical cell instance with the following commands:

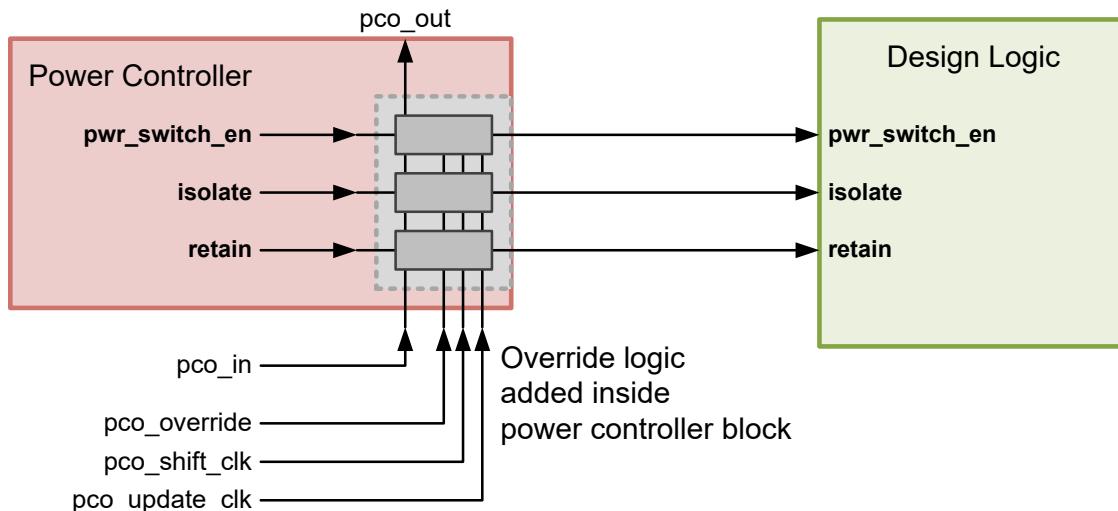
```
set_dft_configuration -power_control enable
set_dft_power_control power_controller_hier_cell
```

The control signal outputs of the power controller block must be defined using the signal specification options of the applicable UPF commands:

- `create_power_switch -control_port`
- `set_isolation_control -isolation_signal`
- `set_retention_control -save_signal`

When a power controller block is configured, the `insert_dft` command inserts wrapper chain override logic at the control signal outputs inside the specified power controller block. [Figure 170](#) shows the structure of the power controller override logic.

Figure 170 Power Controller Override Logic



The power controller override wrapper chain is composed of control-observe cells along the power controller outputs. The wrapper chain is a separate chain that cannot be combined with other scan chains. For designs with scan compression, the power controller override wrapper chain exists outside the compressor-decompressor logic.

The following signals provide control of the power controller override logic:

- pco_in and pco_out

These signals are the scan-in and scan-out signals for the wrapper chain cells.

- pco_override

This signal is asserted in test mode to override the power controller's control signals with the wrapper chain override values.

- pco_shift_clk

When clocked with ScanEnable de-asserted, this signal captures the current output values from the power controller to improve observability. When clocked with ScanEnable asserted, this signal shifts data through the shadow registers of the wrapper chain. When the override is asserted, the current override state is not affected when data is shifted through the wrapper chain shadow registers.

- pco_update_clk

This signal is clocked to transfer the control signal values from the wrapper cell shadow registers to the wrapper cell output registers.

By default, the tool creates these signals and ports. To use existing placeholder ports for these signals, define them as follows:

```
dc_shell> # PCO control signals
dc_shell> set_dft_signal -view spec -type pco_override -port port_name
dc_shell> set_dft_signal -view spec -type pco_shift_clk -port port_name
dc_shell> set_dft_signal -view spec -type pco_update_clk -port port_name

dc_shell> # PCO scan data signals
dc_shell> set_dft_signal -type ScanDataIn -port pco_in_port_name
dc_shell> set_dft_signal -type ScanDataOut -port pco_out_port_name
dc_shell> set_scan_path -view spec -class pco_wrapper MY_PCO_CHAIN \
           -scan_data_in pco_in_port_name \
           -scan_data_out pco_out_port_name
```

During power controller override logic insertion, the tool updates the test_setup procedure with test vectors that place the power control signals into a stable state. If multiple test modes have been defined, they are all updated.

This stable state has the following characteristics:

- All controllable power supply switches are enabled.
- All controllable isolation cells are set to a pass-through state.
- All controllable retention cells are placed into save mode.

If you have defined all power controller signals in the UPF specification, the resulting test protocol can be used directly in the TestMAX ATPG tool with no editing. If any power controller signals have not been captured in the UPF specification, you must add the required signal values to the `test_setup` procedure.

Limitations

Note the following limitations of the power controller override feature:

- Only one power controller instance can be specified.
- If the power controller logic is distributed across several blocks, you must first group it into a single block.
- It is not possible to use an existing scan segment as the power controller override wrapper chain.
- Power controller override signals cannot be internally driven in the internal pins flow.

Reducing Shift Power Using Functional Output Pin Gating

During scan testing, while scan data shifts through scan chains, the functional logic driven by the scan flip-flops also toggles. This can cause increased power dissipation during testing, which could damage the design under test.

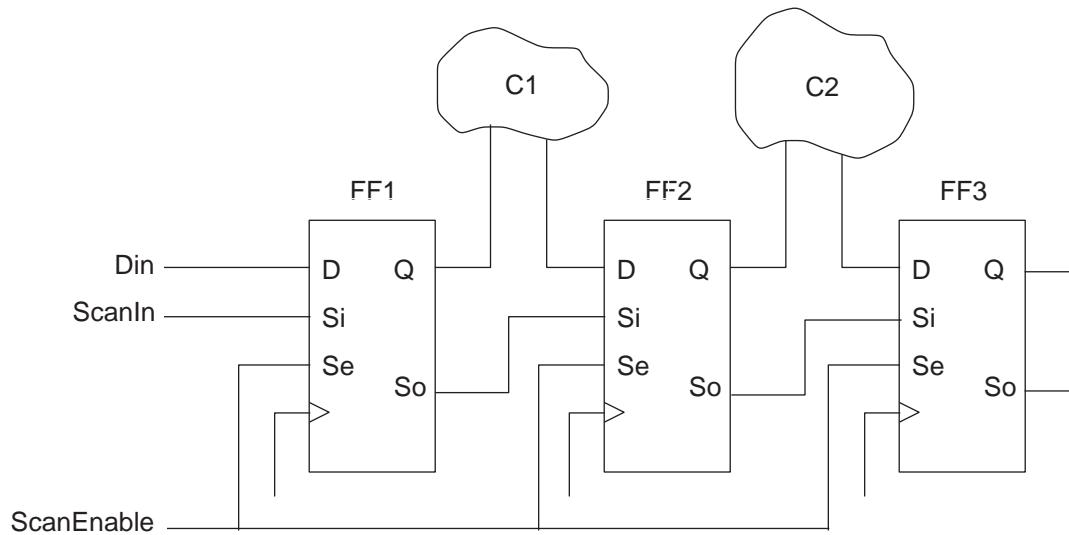
You can use the `set_scan_suppress_toggling` command to enable functional output gating. When this command is used, the tool inserts gating logic to suppress toggling on the functional output of scan flip-flops that either you specify or the tool automatically selects. The tool uses AND-gating or OR-gating logic, depending on which constant value most reduces toggling from other ungated signals entering the fanout logic cone.

Note:

This feature only works with designs that use the multiplexed flip-flop scan style.

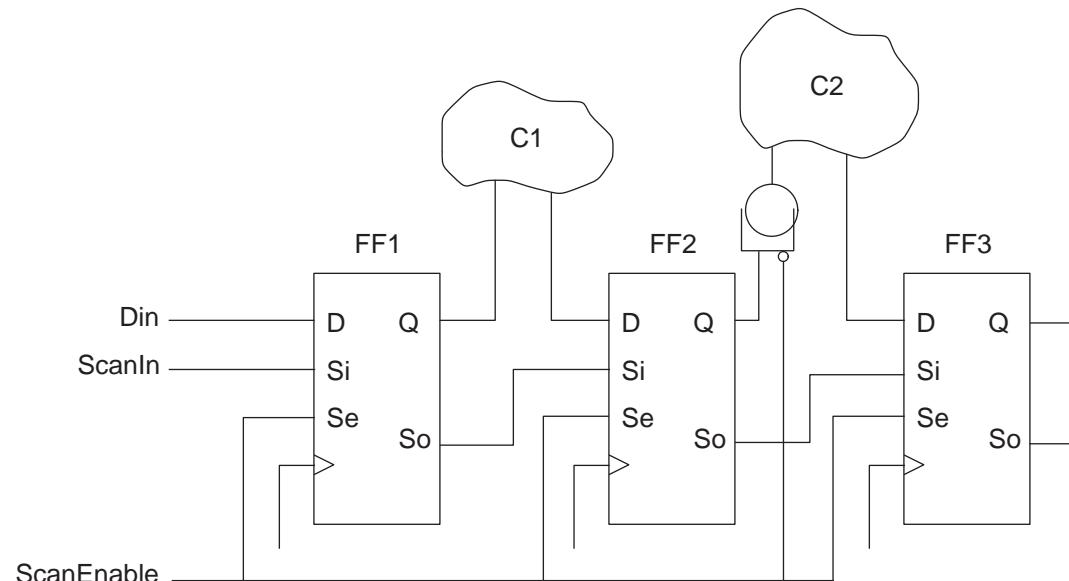
[Figure 171](#) shows the design after DFT insertion but without any gating logic inserted on the functional output.

Figure 171 Design After DFT Insertion Without Gating Logic on the Functional Output



[Figure 172](#) shows the design after the tool has inserted AND-gating logic on the output pin of the FF2 flip-flop.

Figure 172 Design After DFT Insertion With AND Clock Gating Inserted on the FF2 Functional Output



In the [Figure 172](#) 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 is gated. [Figure 173](#) and [Figure 174](#) show this case for both AND-gating logic and OR-gating logic, respectively.

Figure 173 Design With AND-Gating Logic Inserted on the Functional Output of FF2, Where FF2 Has a Single Output Pin Shared Between Functional and Scan Output

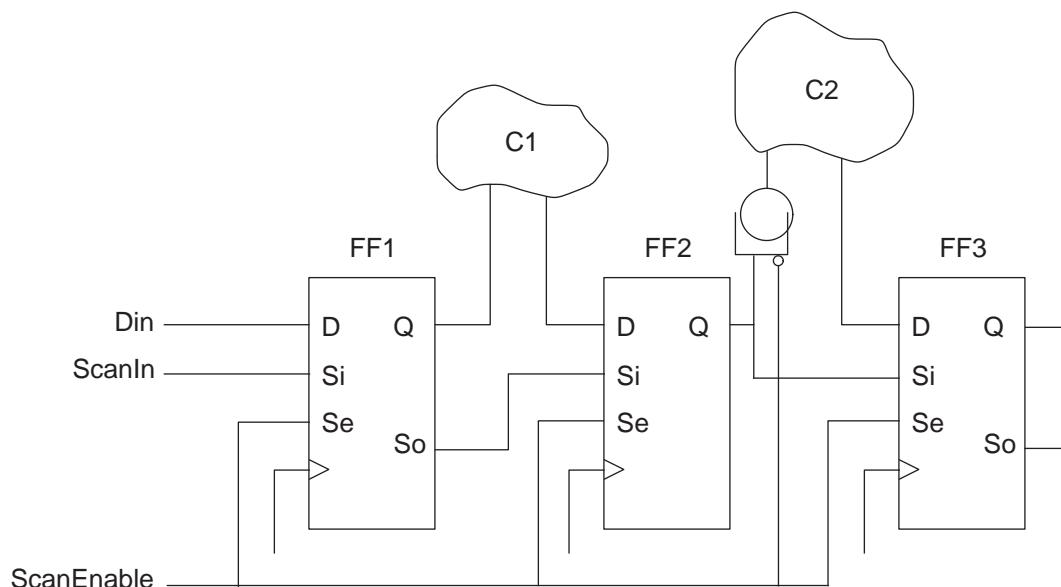
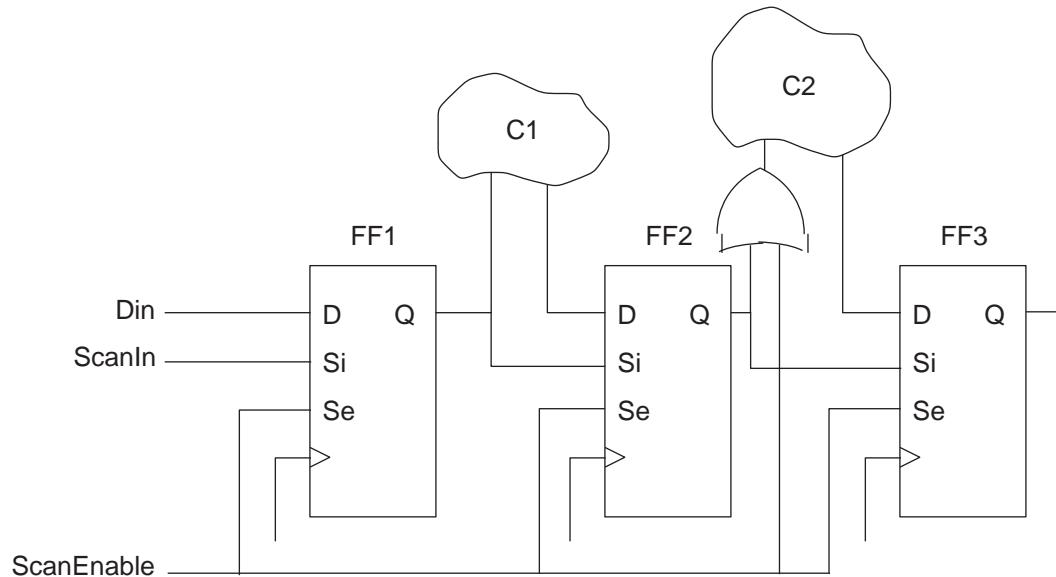


Figure 174 Design With OR-Gating Logic Inserted on the Functional Output of FF2, Where FF2 Has a Single Output Pin Shared Between Functional and Scan Output



To use this feature, run the `set_scan_suppress_toggling` command as part of your DFT specifications before you run the `insert_dft` command.

The `set_scan_suppress_toggling` command has the following syntax:

```
set_scan_suppress_toggling
    -selection_method [manual|auto|mixed]
    -include_elements collection_of_design_objects
    -exclude_elements collection_of_design_objects
    -ignore_timing_impact [true|false]
    -min_slack [0_to_1000]
    -total_percentage_gating [0.001_to_100]
```

Argument	Description
<code>-selection_method</code> manual auto mixed	This option defaults to <code>manual</code> , which applies gating only for objects manually specified with the <code>-include_elements</code> option. The <code>auto</code> value enables automatic selection of flip-flops for gating using power-based heuristics. The <code>mixed</code> value enables a combined approach, allowing the manual specification to be supplemented with automatic selection by the tool.

Argument	Description
<code>-include_elements collection_of_design_objects</code>	This option is used to manually specify a collection of design objects for gating. Flip-flop instance names, hierarchical cell names, and flip-flop and design references can be specified. A hierarchical cell specification includes all scan flip-flops in the cell.
<code>-exclude_elements collection_of_design_objects</code>	This option is used to manually specify a collection of design objects that should be excluded from gating. Flip-flop instance names, hierarchical cell names, and flip-flop and design references can be specified. A hierarchical cell specification includes all scan flip-flops in the cell.
<code>-ignore_timing_impact true false</code>	This option is <code>false</code> by default. When this option is set to <code>true</code> , the minimum slack requirement specified by the <code>-min_slack</code> option is ignored. When set to <code>true</code> , the <code>-exclude_elements</code> option might be useful for guiding automatic selection.
<code>-min_slack num_0_to_1000</code>	This option specifies the required minimum slack value after gating has been added. The default is 0, which means the slack should be nonnegative after gating.
<code>-total_percentage_gating num_0.001_to_100</code>	This option specifies the target percentage of scan flip-flops to be automatically selected for gating when the <code>-selection_method</code> option is set to <code>auto</code> or <code>mixed</code> . The default for this option is 5.

Functional gating logic can be inserted if the scan flip-flop is part of a scan chain. A flip-flop is excluded from gating consideration if it meets any of the following criteria:

- The flip-flop is manually excluded with the `-exclude_elements` option.
- The flip-flop is part of a shift register segment identified by the `compile_ultra` command.
- 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` attribute on the block.
- The Q or QN pin of the flip-flop connects only to the scan-in signal. (They can be gated if they are functionally connected.)
- The logic does not meet the minimum slack limit or would not meet the minimum slack limit if gating is inserted.

When the `-selection_method` option is set to `auto` or `mixed`, flip-flops are automatically selected for gating using propagated switching power analysis. A percentage value between 5 and 30 percent is typically specified with the `-total_percentage_gating`

option. Consider the trade-offs between test-mode power consumption, functional power consumption, and functional timing when choosing a gating percentage value. As the gating percentage value is increased, leakage power increases, and the potential for layout congestion and timing closure difficulty increases. While considering timing and power trade-offs, you should also consider that TestMAX ATPG has several power-aware algorithms that seek to reduce shift and capture flip-flop toggle rates during the ATPG process.

When the `-selection_method` option is set to `manual` or `mixed`, all flip-flops specified with the `-include_elements` option are gated, except those meeting the exclusion criteria defined earlier.

When the `-selection_method` option is set to `mixed`, all flip-flops specified with the `-include_elements` option are gated, even if the `-total_percentage_gating` target is exceeded. If the `-total_percentage_gating` target is not met by the manually specified flip-flops, additional flip-flops are selected automatically to meet the goal.

Use the `report_scan_suppress_toggling` command to confirm the option settings you specified with the `set_scan_suppress_toggling` command. Use the `remove_scan_suppress_toggling` command to remove the previous toggling suppression settings applied.

As described in [Previewing Additional Scan Chain Information on page 604](#), you can use the `preview_dft -show {qgates}` command before DFT insertion to see which scan cells will be gated. This command generates a preview report that annotates the gated scan cells with the `(g)` attribute. For example:

```
dc_shell> preview_dft -show {qgates}
...
(g) shows cell scan-out drives a toggle suppressing gate

Scan chain '1' (SI1 --> SO1) contains 48 cells
    CORE/Dstrobe_reg           (CLK, 45.0, rising)
    CORE/R1_reg[0]  (g)
    CORE/R1_reg[1]  (g)
    ...

```

When the `-selection_method` option is set to `auto` or `mixed`, the `preview_dft` command reports the number of scan flip-flops automatically selected for gating, the number considered for gating, the number that you manually included, and the number rejected due to timing considerations.

During DFT insertion, the combinational gating cell instances are named using the `compile_instance_name_prefix` variable, by default. To specify a different instance name prefix for the gating cells, set the `test_suppress_toggling_instance_name_prefix` variable to the desired prefix. When this variable is set, the scan cell leaf name and output pin name are also appended to the

specified prefix. For example, if you set the variable to QGATE, a gating cell inserted at ENAB_reg/Q is named QGATE_ENAB_reg_Q.

After DFT insertion completes, you can use the `test_scan_suppress_toggling cell` attribute to find the gating cells. For example,

```
dc_shell> set cells [get_cells -hierarchical * \
                     -filter {test_scan_suppress_toggling == true}]
{CORE/U181 CORE/U182 U224 U225 U226}
```

See Also

- [Previewing the DFT Logic on page 603](#) for more information about previewing scan chain structures

Controlling Clock-Gating Cell Test Pin Connections

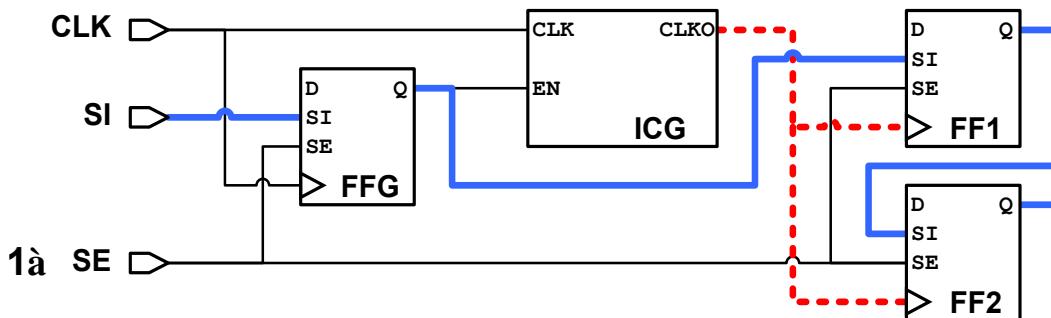
To insert clock-gating cells in a design, you can use the following methods:

- Automatic insertion of clock-gating cells by Power Compiler, using the `-gate_clock` option of the `compile` or `compile_ultra` commands
- Manual instantiation or insertion of clock-gating cells

A clock-gating cell propagates the clock signal to downstream logic only when the enable signal is asserted. During scan shift, if the enable signal is controlled by one or more scan flip-flops, the shifting test data values cause the clock-gating signal to toggle during scan shift. As a result, the clock signal does not reliably propagate through the clock-gating cell to downstream scan flip-flops during scan shift, resulting in scan shift violations.

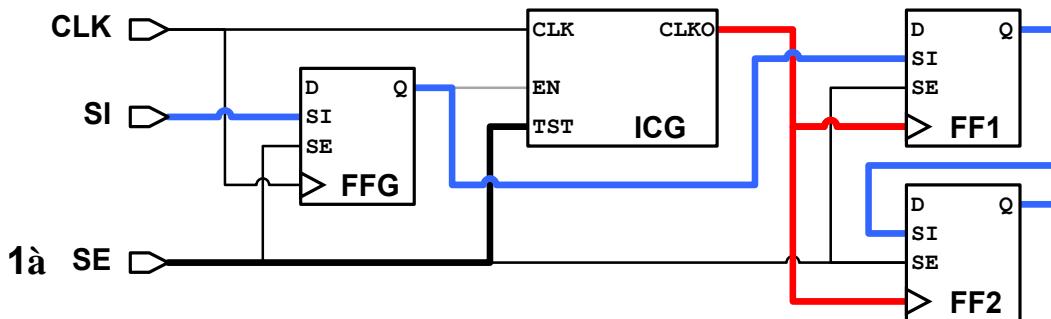
[Figure 175](#) shows an example where the enable signal of an integrated clock-gating cell is driven by a scan flip-flop. The scan chain path is highlighted in blue. During scan shift, the clock-gating enable signal driven by FFG toggles, interrupting the scan shift clocks needed for FF1 and FF2.

Figure 175 Clock-Gating Cell Enabled by Scan Flip-Flop Output



To remedy this, most clock-gating cells have test pins that force the clock signal to pass through when the test pin is asserted. This ensures that downstream scan cells successfully receive the clock signal and shift values along the scan chain. [Figure 176](#) shows a test-aware clock-gating cell with its test pin hooked up to the global scan-enable signal. During scan shift, FF1 and FF2 receive the clock signal and successfully shift data.

Figure 176 Clock-Gating Cell Enabled by Scan Flip-Flop Output



This topic describes the methods provided by DFT Compiler to control clock-gating cell test pin connections.

Connecting User-Instantiated Clock-Gating Cells

You can use the `insert_dft` command to connect user-instantiated clock-gating cells, that is, to connect to the clock-gating cells that have not been inserted by Power Compiler. You use the `set_dft_clock_gating_pin` command to specify the unconnected test pin of the clock-gating cells in your design. Then you run the `insert_dft` command to connect these pins to the test ports.

Connecting user-instantiated clock-gating cells with the `set_dft_clock_gating_pin` and `insert_dft` commands has the following advantages:

- Provides flexibility
- Does not require setting Power Compiler attributes
- Has no dependency on the `identify_clock_gating` command and the `power_cg_auto_identify` variable
- Works with multiple ScanEnable and TestMode signal connectivity

Note:

When clock-domain-based connections are specified, using the `set_dft_signal -connect_to` command, user-instantiated clock-gating pins are not connected by domain. For this feature, only clock-gating cells recognized and inserted by Power Compiler are supported.

The syntax for the `set_dft_clock_gating_pin` command is

```
set_dft_clock_gating_pin object_list -pin_name instance_pin_name
    [-control_signal ScanEnable | TestMode]
    [-active_state 1 | 0]
```

object_list

List of clock-gating cell instances for which test pins are specified. The argument is mandatory.

-pin_name

Name of the test pin on the specified instances. This pin name must be common to all specified instances. The argument is mandatory.

-control_signal

Specifies the type of control signal required by the test pin. The argument is optional. The default is `ScanEnable`.

-active_state

Specifies the active state of the test pin. The argument is optional. The default is 1.

The command is cumulative.

The specified cells and test pin are not checked. Verify that you have specified the actual clock-gating cells and test pin in the design and that they were not identified by Power Compiler. Specifying cells that are not clock-gating cells can cause undesired results when you run the `dft_drc` and `insert_dft` commands.

Using the `set_dft_clock_gating_pin` Commands: Examples

- To specify the test enable (TE) pin on instances U1 and U2 as the clock-gating pin, using the default signal type `ScanEnable`:

```
set_dft_clock_gating_pin [list U1 U2] -pin_name TE
```

- To specify the `test_mode` pin as the clock-gating pin of control signal type `TestMode` on the hierarchical design `des_a`:

```
set_dft_clock_gating_pin \
    [get_cells * -hierarchical -filter "@ref_name == des_a"] \
    -control_signal TestMode -pin_name test_mode
```

- To specify pin A as a clock-gating pin of control signal type `ScanEnable` with active state 1 for all instances of the unique library cell CGC1:

```
set_dft_clock_gating_pin \
    [get_cells * -hierarchical -filter "@ref_name == CGC1"] \
    -control_signal scan_enable -pin_name A
```

Use the `report_dft_clock_gating_pin` command to report the specifications you made with the `set_dft_clock_gating_pin` command. To remove the DFT clock-gating pin specifications, use the `remove_dft_clock_gating_pin` command.

Interaction With Other Commands

Clock-gating cells identified with the `set_dft_clock_gating_pin` command can be used or specified in the following commands:

- In the hook-up-only flow, using the following command:

```
set_dft_configuration -scan disable -connect_clock_gating enable
```

- Using the `set_dft_signal -connect_to` command

The connection is not made when doing clock-domain-based connections.

- Using the `set_dft_clock_gating_configuration -exclude_elements` command.

Script Example

[Example 57](#) and [Example 58](#) show a hookup-clock-gating only flow and a complete DFT insertion flow, respectively.

Example 57 Hookup-Clock-Gating Only Flow

```
read verilog test.v
link
set_dft_signal -view existing_dft -type ScanClock -port clk \
    -timing {45 55}
set_dft_signal -type ScanEnable -port SE1 -view spec
set_dft_clock_gating_pin {clk_gate_out1_reg sub1/clk_gate_out1_reg} \
    -pin_name TE
set_dft_configuration -scan disable -connect_clock_gating enable
report_dft_clock_gating_pin
insert_dft
```

Example 58 Complete DFT Insertion Flow

```
read verilog test.v
link
set_dft_signal -view existing_dft -type ScanClock -port clk \
    -timing {45 55}
set_dft_signal -type ScanEnable -port SE1 -view spec
set_dft_clock_gating_pin {clk_gate_out1_reg sub1/clk_gate_out1_reg} \
    -pin_name TE
set_dft_configuration -scan enable -connect_clock_gating enable
report_dft_clock_gating_pin
create_test_protocol
dft_drc -verbose
insert_dft
```

Limitations

Note the following limitations:

- If you use the `set_dft_signal -connect_to` command to make clock-domain-based connections to clock-gating cells, only Power Compiler clock-gating cells are considered; user-instantiated clock-gating cells are not considered.

For more information, see [Specifying Signals for Clock-Gating Cell Test Pin Connections](#).

- The feature is not supported for pipelined scan-enable signals.

Excluding Clock-Gating Cells From Test-Pin Connection

You might have a portion of the design that is excluded from scan testing, and you do not want DFT Compiler to connect the test pins of those clock-gating cells to test-mode or scan-enable signals. To prevent the `insert_dft` command from connecting the test pins of some clock-gating cells, use the `-exclude_elements` option of the `set_dft_clock_gating_configuration` command:

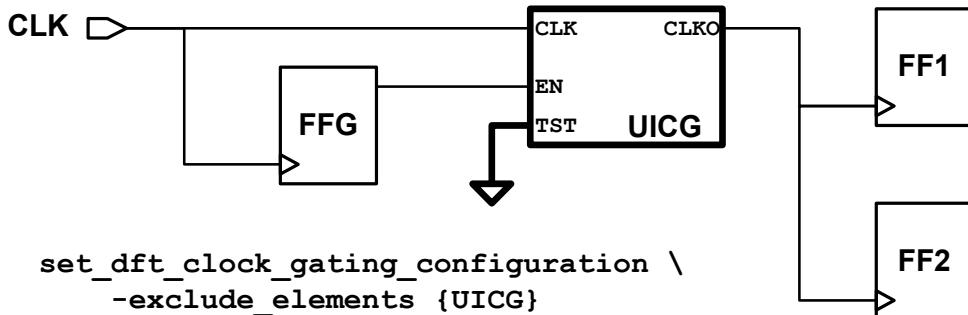
```
set_dft_clock_gating_configuration -exclude_elements object_list
```

The `object_list` can include the following object types:

- Clock-gating cell leaf instances
- Clock-gating observation cell leaf instances
- Hierarchical cell instances – All clock-gating cells within the instances are included in the specification.
- Clock-gating library cell – All instances of that cell are included in the specification.
- Clocks – All clock-gating cells in the clock domains are included in the specification.

Instead of connecting the test pins of excluded clock-gating cells to test-mode or scan-enable signals, DFT Compiler leaves the existing connections in place, which are typically constant drivers that de-assert the test-mode bypass. [Figure 177](#) shows an example of an excluded clock-gating cell.

Figure 177 Directly Specified Excluded Clock-Gating Cell



The `dft_drc` command does not report any TEST-130 unconnected test-pin messages for excluded clock-gating cells. However, any downstream scan cells driven by the excluded clock-gating cells will result in D1 or D9 violations if their clock is uncontrolled.

If you do not know the clock-gating cell instances, but you do know the nonscan flip-flops whose upstream clock-gating cells should not be connected, you can use the `-dont_connect_cgss_of` option of the `set_dft_clock_gating_configuration` command:

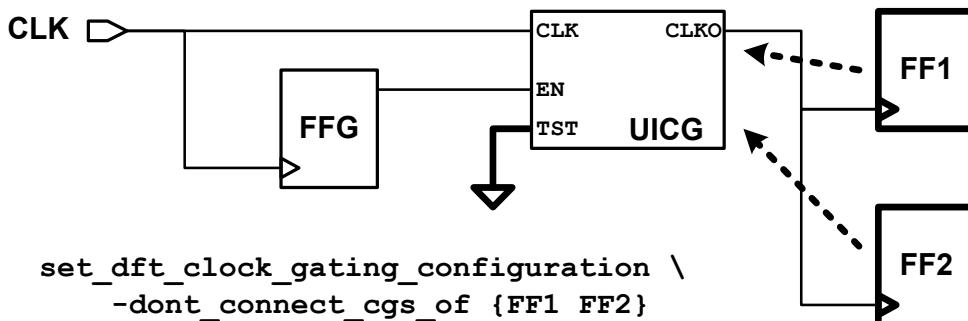
```
set_dft_clock_gating_configuration -dont_connect_cgss_of object_list
```

The `object_list` can include the following object types:

- Flip-flop leaf instances
- Hierarchical cell instances – All flip-flops within the instances are included in the specification.

With the `-dont_connect_cgss_of` option, DFT Compiler identifies any upstream clock-gating cells from these registers, and prevents their test pin connections. Figure 178 shows an example of an excluded clock-gating cell.

Figure 178 Upstream Excluded Clock-Gating Cell

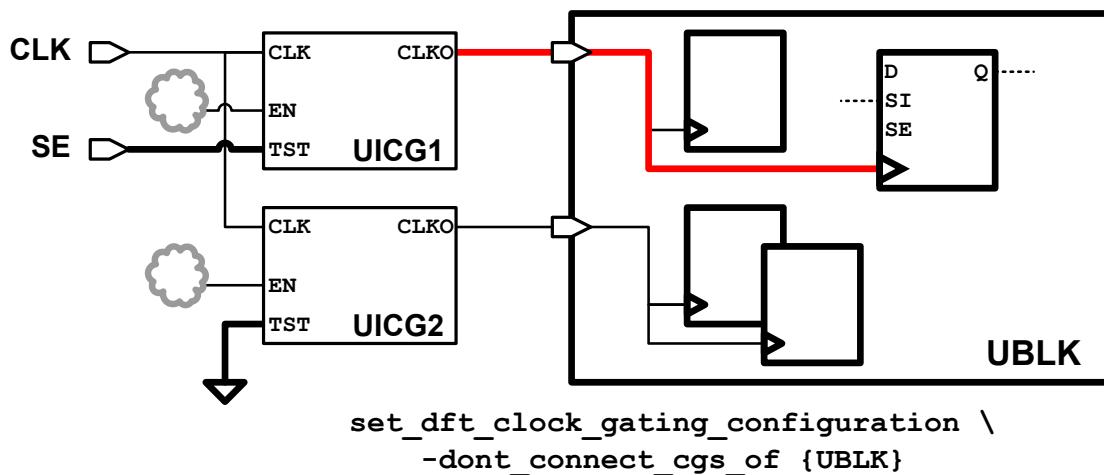


The upstream clock-gating cell can only drive nonscan cells or scan-replaced cells that are excluded from scan stitching. If the clock-gating cell drives any valid scan cells that

are incorporated into scan chains, the test pin is connected to ensure proper scan shift clocking.

Figure 179 shows an example where hierarchical block UBLK is specified with the `-dont_connect_cgss_of` option. The four flip-flops inside the block are included in the specification. However, one of the flip-flops is a valid scan flip-flop that will be included on a scan chain. As a result, the test pin of upstream clock-gating cell UICG1 is tied to a test signal. The test pin of clock-gating cell UICG2 is left tied to its de-asserted constant value.

Figure 179 Upstream Excluded Clock-Gating Cell Due to Downstream Scan Flip-Flops



If DFT Compiler cannot find the upstream clock-gating cells for flip-flops during the `preview_dft` or `insert_dft` commands, it issues TEST-154 information messages for those flip-flops.

For both exclusion methods, use the `report_dft_clock_gating_configuration` command to report the specifications that you previously set. Use the `reset_dft_clock_gating_configuration` command to remove the specifications.

Exclusion works by leaving the default de-asserted constant value connected to the test pins of excluded clock-gating cells. If an exclusion applies to a clock-gating cell with an existing test pin connection other than the default de-asserted constant value, that existing test pin connection remains in place.

The clock-gating cell test pin connection control methods have the following precedence, in order of highest priority to lowest priority:

- `set_dft_signal -connect_to` (highest priority)
- `set_dft_clock_gating_configuration -exclude_elements`
- `set_dft_clock_gating_configuration -dont_connect_cgss_of` (lowest priority)

When the `-dont_connect_cgss_of` option is used, upstream clock-gating cells of the specified registers are excluded from clock-gating cell test pin connection only if none of the downstream registers are valid scan cells. When the `-exclude_elements` option is used, the test pin connections of the specified clock-gating cells are suppressed even if there are downstream valid scan cells.

Note:

These `set_dft_clock_gating_configuration` options work with, but are not intended to be used for, user-instantiated clock-gating cells. If the test pin of a user-instantiated clock-gating cell should not be hooked up, do not include it in a `set_dft_clock_gating_pin` specification. Registers driven by user-defined clock gating cells are traced through simple buffer and inverter logic only.

Connecting Clock-Gating Cell Test Pins Without Scan Stitching

You can use the `insert_dft` command to connect the test pins of clock-gating cells to scan-enable or test-mode signals without also performing scan insertion or scan stitching. Note, however, that only clock-gating cells with Power Compiler attributes are considered.

To use this feature, you must disable scan insertion and enable the clock-gating connection before running the `insert_dft` command. This is accomplished by issuing the following command:

```
set_dft_configuration -scan disable -connect_clock_gating enable
```

When using this flow, do not run the `create_test_protocol` or `dft_drc` commands. If you do, it will prevent the `insert_dft` command from making the clock-gating cell test pin connections.

[Example 59](#) shows how to implement this feature.

Example 59 Using the `insert_dft` Command to Connect the Test Pins of Clock-Gating Cells

```
read ddc design.ddc
set_clock_gating_style -control_signal scan_enable
create_clock clk -name clk
compile_ultra -scan -gate_clock

set_dft_signal -type ScanEnable -view spec -port ICG_SE

# Disable scan insertion, enable only clock-gating cell
# test pin connections
set_dft_configuration -scan disable -connect_clock_gating enable

# Run insert_dft to connect the clock-gating cell test pins only
insert_dft
```

The `insert_dft` command issues an information message indicating that clock-gating cell test pins are being connected to the specified test signal:

```
Routing clock-gating cells
Information: Routing clock-gating cell test pins with no specified
driver to scan enable 'ICG_SE'
1
```

Note:

This capability is not meant to be used as part of a scan-stitching flow. The feature is solely intended to allow you to connect the test pins of clock-gating cells, separate from any scan synthesis run. To connect clock-gating test pins to a different scan-enable signal during scan stitching, use the `-usage` option of the `set_dft_signal` command. For more information, see [Specifying Signals for Clock-Gating Cell Test Pin Connections](#).

The following scenarios show how to use the `insert_dft` command to connect clock-gating cell test pins to various types of test signal ports and pins:

- To connect to the default `test_se` port:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
insert_dft
```

- To connect to an existing scan-enable port:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
set_dft_signal -view spec -type ScanEnable -port SE
insert_dft
```

- To connect to an existing test-mode port:

```
set_clock_gating_style -control_point after -control_signal TestMode
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
set_dft_signal -view spec -type TestMode -port TM
insert_dft
```

- To connecting to an input pad cell hookup pin:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
set_dft_signal -view spec -type ScanEnable -port SE \
    -hookup_pin UPAD_SE/IO_Q
insert_dft
```

- To connect to an internal pin—for example, to a black-box output:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
# Enable internal pins flow
```

```
set_dft_drc_configuration -internal_pins enable
set_dft_signal -view spec -type ScanEnable -port SE \
    hookup_pin IP_CORE/SE_OUT
insert_dft
```

The following features apply to automatic connection of clock-gating cell test pins:

- Connections are made only to the test pins of valid clock-gating cells that have been correctly identified and have the required Power Compiler attributes.
- Only connections to clock-gating cells are made unless boundary scan insertion is enabled, in which case boundary scan insertion takes precedence.
- Connections specified with the `set_dft_signal -connect_to` command are honored.
- The internal pins flow is supported.

The following features or capabilities do not work or have limited capability:

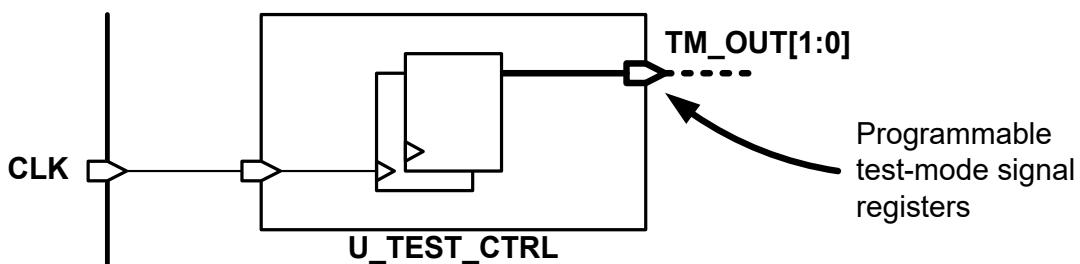
- If test models need to be connected, you must specify the core-level ScanEnable pin to be connected to the top-level ScanEnable signal, using the `set_dft_signal -connect_to` command.
- Partially incomplete flows (in which you have performed the clock-gating cell connections to the blocks but not performed the rest of the DFT flow) should not be used for top-level DFT insertion.

Internal Pins Flow

Normally, DFT signals (such as clocks, resets, scan-ins, and scan-outs) are defined on ports of the current design. Even when defined with a hookup pin, these signals are ultimately connected to a port.

However, advanced DFT architectures might require some DFT signals to connect to an internal pin, with no straightforward path to a port. The following figure shows programmable registers intended to drive TestMode signals:

Figure 180 Signals Connected to Internal Pins Instead of Ports



In these cases, you can use the *internal pins* flow to define such signals. It is called a flow because it requires consideration both before and after DFT insertion, as described below.

See Also

- [SolvNet article 040136, “Using the Internal Pins Flow With Internal Test Registers”](#) for an example that drives DFT signals from design registers

Defining Signals on Internal Pins

To define signals on internal pins, do the following:

1. In your global DFT configuration, enable the internal pins flow:

```
set_dft_drc_configuration -internal_pins enable
```

2. When defining your DFT signals, define each internal-pins signal by using the `-hookup_pin` option *without* the `-port` option:

```
set_dft_signal -view spec -type TestMode \
  -hookup_pin U_TEST_CTRL/TM_OUT[1]
set_dft_signal -view spec -type TestMode \
  -hookup_pin U_TEST_CTRL/TM_OUT[0]
```

The `-hookup_pin` option accepts only a single pin object; use multiple commands for multiple pins.

You can define a mix of port-driven and internal-pins signals as needed.

The following signal types can be defined as internal-pins signals.

Supported Internal-Pins Signal Types

ScanMasterClock

ScanMasterClock

MasterClock

ScanClock

Reset

Constant

TestMode

ScanEnable

Supported Internal-Pins Signal Types

ScanDataIn
ScanDataOut
pll_reset
pll_bypass
LOSPipelineEnable
lbistCaptureCycleEnable

Writing Out the Test Protocol

After DFT insertion, the CTL model for the design contains information about internal pins. This information is understood by DFT Compiler, but it cannot be used by TestMAX ATPG.

Before writing out the STIL protocol file (SPF) for TestMAX ATPG, disable the internal pins flow:

```
dc_shell> set_dft_drc_configuration -internal_pins disable
dc_shell> write_test_protocol -test_mode Internal_scan \
           -output Internal_scan_needs_modification.spf
```

This prevents the `write_test_protocol` command from including information about internal pins in the SPF. (This does not affect post-DFT DRC or writing the design in .ddc format.)

Before using the protocol in TestMAX ATPG, you must make any modifications needed for the assumptions in the test protocol to be true.

Limitations of the Internal Pins Flow

Note the following limitations of the internal pins flow:

- In most cases, the output protocol is not accurate and cannot be used in the TestMAX ATPG tool unless modified.
- Boundary scan and scan extraction flows do not support the internal pins flow.
- You cannot integrate cores created using the internal pins flow unless you set the `test_allow_internal_pins_in_hierarchical_flow` variable to `true` during both core creation *and* core integration. For details, see the man page.

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 topic covers the following:

- [Configuring Scan Grouping](#)
 - [Scan Group Flows](#)
 - [Known Limitations](#)
-

Configuring Scan Grouping

DFT Compiler includes the following 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 is 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 `-serial_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 is `false`.

Note the following:

- There is no `-view` option to the `set_scan_group` command, because the options only works 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 before the `insert_dft` command 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

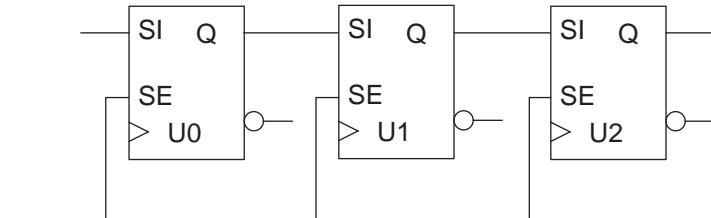
If you have existing serial routed segments at the current design level that you want to incorporate as part of a longer scan chain, you can use the `set_scan_group -serial_routed true` command to accomplish this. When you run the `insert_dft` command, it will then connect to this segment while keeping the segment intact.

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 the `insert_dft` command should connect to this segment.

Consider the existing scan chain shown in [Figure 181](#).

Figure 181 Integrating an Existing Scan Chain



In this example, the scan chain connects the flip-flops U0 through 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. 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.

See Also

- [Performing Scan Extraction on page 439](#) for more information about how to extract pre-existing scan chains in your design

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 are reported.

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 logic 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.

Shift Register Identification

The following topics describe shift register identification in DFT flows:

- [Simple Shift Register Identification](#)
- [Synchronous-Logic Shift Register Identification](#)
- [Shift Register Identification in an ASCII Netlist Flow](#)

For more information about how DC Ultra identifies shift registers during a test-ready compile, see “Sequential Mapping” chapter in the Design Compiler User Guide.

Simple Shift Register Identification

By default, DC Ultra identifies simple shift registers, in which each flip-flop directly captures the output of the previous flip-flop. When you perform a test-ready compile with the `compile_ultra -scan` command, DC Ultra identifies simple shift registers and performs scan replacement only on the first register of each identified shift register.

Information about the identified shift registers is stored in the design database. DFT Compiler uses this information during scan stitching to efficiently incorporate the shift registers into the scan chain. This flow reduces the area overhead for scan replacement.

Shift register identification is not performed across hierarchical boundaries when only a 2-bit shift register would be created. This typically occurs with registered interfaces where a single register at the output of one design is connected to a single register at the input of another design. This behavior reduces port punching when DFT insertion connects scan chains to shift registers that cross hierarchical boundaries, which can help improve congestion and scan wire length results. Shift register identification is performed across hierarchical boundaries for 3-bit and longer shift registers.

The `insert_dft` command uses the stored shift register information from DC Ultra to optimize the scan path stitching process. For each shift register, the first scan-replaced cell provides scan controllability for the entire shift register. Simple shift registers are used

directly in the scan path. If needed, identified shift registers are broken up during DFT insertion to meet scan chain balancing or maximum scan chain length requirements.

Use the following methods to determine the shift registers that were identified by the `compile_ultra -scan` command:

- Use the `preview_dft -show {segments}` command to report the identified shift registers, which are treated as scan segments after being identified. For details on previewing scan segments, see [Previewing Additional Scan Chain Information on page 604](#).
- Use the `shift_register_head` and `shift_register_flop` cell attributes to identify the leading shift register scan cells and subsequent nonscan cells, respectively:

```
dc_shell> get_cells -hierarchical * -filter \
           {shift_register_head==true || shift_register_flop==true}
```

For best results, write out the design in .ddc format to preserve the identified shift register attributes.

To disable shift-register identification, set the following variable:

```
dc_shell> set_app_var compile_seqmap_identify_shift_registers false
```

Synchronous-Logic Shift Register Identification

During a test-ready compile, DC Ultra can optionally identify *synchronous-logic shift registers*, in which each flip-flop captures a combinational logic function that includes the output of the previous flip-flop. DFT insertion updates the combinational logic function between the flip-flops so that the shift register logically degenerates to a simple shift register when the scan-enable signal is asserted.

To enable synchronous-logic shift register identification, set the following variable:

```
dc_shell> set_app_var \
           compile_seqmap_identify_shift_registers_with_synchronous_logic \
           true
```

The default is `false`, which preserves multibit registers better between the `compile` and `insert_dft` steps and can provide additional area savings. However, you can enable synchronous-logic shift register identification if it provides an area benefit for your particular design.

Shift Register Identification in an ASCII Netlist Flow

When you read in a test-ready (scan-replaced) ASCII netlist (Verilog or VHDL), you use the `set_scan_state` command to indicate that the netlist is test-ready:

```
read_verilog block_test_ready.vg
current_design block
link_design
set_scan_state test_ready
```

If the netlist contains shift registers previously identified by the `compile_ultra -scan` command, the attributes for those shift registers are not stored in the netlist file. As a result, the `set_scan_state` command re-identifies them and reapplies their attributes to the design in memory.

Simple Shift Registers

The `set_scan_state test_ready` command always re-identifies any simple shift registers in the design. In other words, it recognizes any structures where a leading scan cell drives a series of one or more same-clocked nonscan cells.

This feature does not require any variables or additional licenses, and it is performed regardless of the value of the `compile_seqmap_identify_shift_registers` variable.

Synchronous-Logic Shift Registers

Synchronous-logic shift registers are not re-identified by default. To re-identify them, set the following variables before running the `set_scan_state` command:

```
# ...read in netlist...
set_app_var compile_seqmap_identify_shift_registers true ;# default is
true
set_app_var
  compile_seqmap_identify_shift_registers_with_synchronous_logic true
set_app_var
  compile_seqmap_identify_shift_registers_with_synchronous_logic_ascii
true
set_scan_state test_ready
```

These variables enable the `set_scan_state test_ready` command to use the DC Ultra shift register identification code, as indicated by the following message:

```
dc_shell> set_scan_state test_ready
Information: Performing full identification of complex shift registers.
(TEST-1190)
```

Note:

Synchronous-logic shift register identification using the `set_scan_state` command requires a DC Ultra license.

If this code re-identifies shift registers differently than test-ready synthesis did, the tool *restructures the registers, scanning or unscanning registers as needed*. This can improve the quality of results, especially when importing netlists from flows without shift register identification.

When synchronous-logic shift register re-identification is enabled, it re-identifies (and restructures) both simple and synchronous-logic shift registers.

To exclude registers from restructuring, apply the `set_dont_touch` or `set_scan_element false` command before running the `set_scan_state` command.

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` command 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
```

11

Wrapping Cores

This chapter shows you how to add a test wrapper to a core design, which creates a *wrapped core*. A wrapped core provides both test access and test isolation during scan pattern application.

Core wrapping is described in the following topics:

- [Core Wrapping Concepts](#)
- [Wrapping a Core](#)
- [Creating User-Defined Core Wrapping Test Modes](#)
- [Creating Compressed EXTEST Core Wrapping Test Modes](#)
- [Creating an IEEE 1500 Wrapped Core](#)
- [Wrapping Cores With OCC Controllers](#)
- [Wrapping Cores With DFT Partitions](#)
- [Wrapping Cores With Multibit Registers](#)
- [Wrapping Cores With Synchronizer Registers](#)
- [Wrapping Cores With Existing Scan Chains](#)
- [Creating an EXTEST-Only Core Netlist](#)
- [Integrating Wrapped Cores in Hierarchical Flows](#)
- [SCANDEF Generation for Wrapper Chains](#)
- [Core Wrapping Scripts](#)

Core Wrapping Concepts

When you integrate a DFT-inserted core into a top-level design, the core-level scan structures are integrated into the top-level scan structures. However, to test the core-level logic separately from the top-level logic, the core must be a *wrapped core*.

DFT Compiler provides two core wrapping flows. The simple core wrapping flow provides basic core wrapping capabilities. The maximized reuse core wrapping flow minimizes the area and timing impact of core wrapping by reusing more existing functional registers.

Wrapped cores are described in the following topics:

- [Wrapper Cells and Wrapper Chains](#)
- [Wrapper Test Modes](#)
- [The Simple Core Wrapping Flow](#)
- [The Maximized Reuse Core Wrapping Flow](#)
- [Wrapping Three-State and Bidirectional Ports](#)

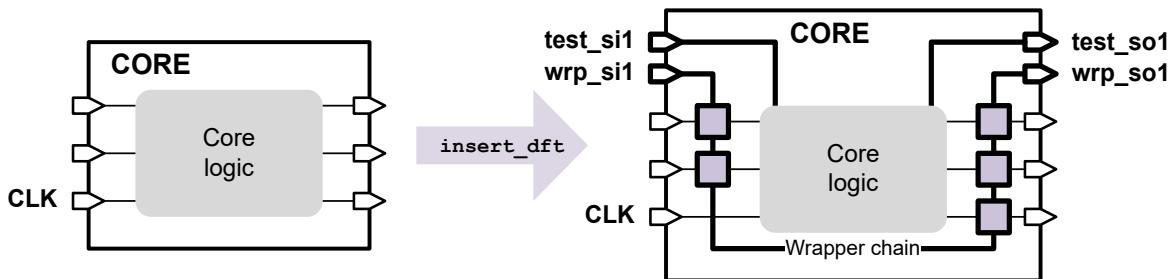
See Also

- [SolvNet article 1918995, “How Do Wrapper Chains and Wrapper Cells Work in Detail?”](#)
for additional reference information about wrapper chains and wrapper cells

Wrapper Cells and Wrapper Chains

A wrapped core has a *wrapper chain* that allows the core to be isolated from the surrounding logic. A wrapper chain is composed of *wrapper cells* inserted between the I/O ports and the core logic of the design. [Figure 182](#) shows an example of a wrapped core.

[Figure 182](#) A Wrapped Core



A wrapper cell consists of a scan cell and MUX logic. It can transparently pass the I/O signal through, or it can capture values at its input and/or launch values at its output. Wrapper chains are shift chains (separate from regular scan chains) that allow known values to be scanned into the wrapper cells and captured values to be scanned out.

Core wrapping is primarily intended to wrap core data ports. The following ports are excluded from wrapping:

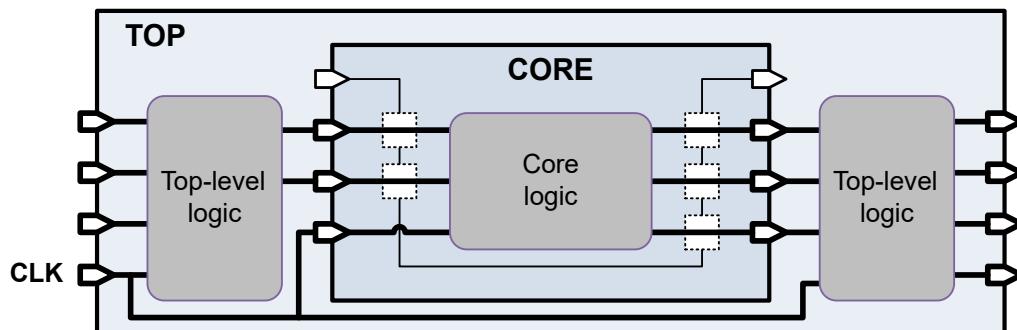
- Functional and test clock ports
- Asynchronous set or reset signal ports
- Scan-input, scan-output, scan-enable, and other global test signal ports
- Wrapper signal ports
- Any port with a constant test signal value defined

The wrapper chain operates in one of four modes—inactive, inward-facing, outward-facing, or safe. These wrapper operation modes behave as follows:

- Inactive mode

The wrapper chain is inactive and I/O signals pass through it. This is the behavior used in mission mode and all non-wrapper test modes.

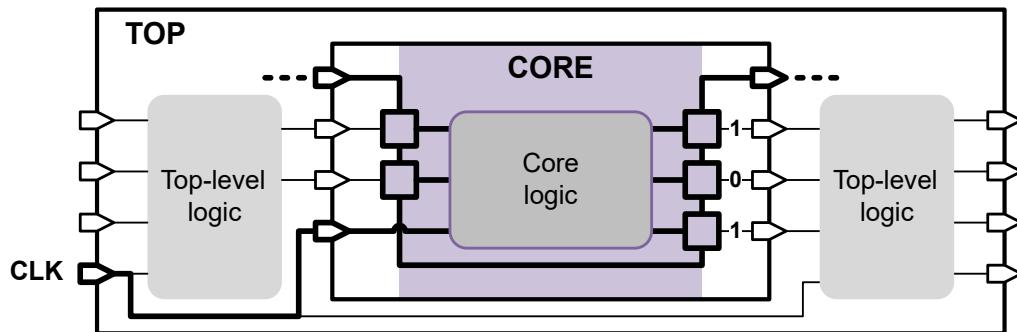
Figure 183 Inactive Mode of Wrapper Chain



- Inward-facing mode (INTEST)

This mode is used to test the core in isolation of the surrounding logic. It includes the wrapper chain and internal chains. The input wrapper cells provide controllability, and the output wrapper cells provide observability. If safe values are specified to protect the surrounding fanout logic from the core output response, they are driven from the output wrapper cells.

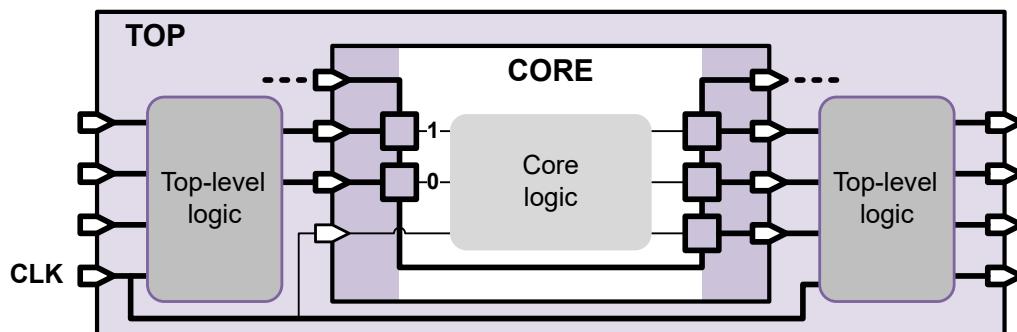
Figure 184 Inward-Facing Mode of Wrapper Chain



- Outward-facing mode (EXTEST)

This mode tests the logic surrounding the core in isolation from the core itself. It includes only the wrapper chain. The input wrapper cells provide observability, and the output wrapper cells provide controllability. If safe values are specified to protect the core inputs from the surrounding fanin logic responses, they are driven from the input wrapper cells. Clock inputs to the core remain unaffected.

Figure 185 Outward-Facing Mode of Wrapper Chain

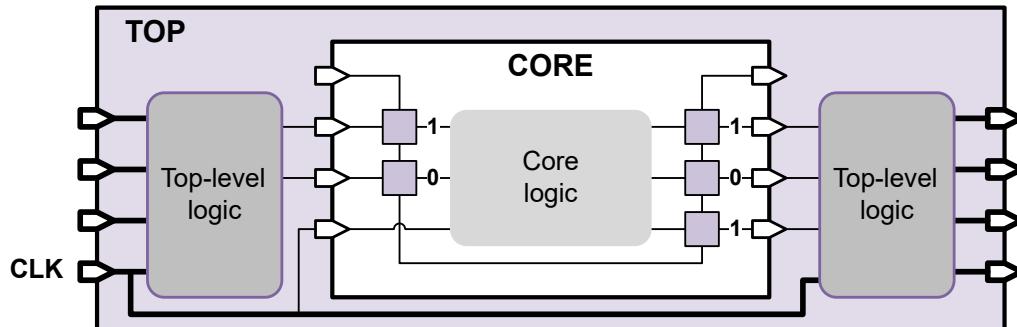


- Safe mode

This optional mode drives safe values from all wrapper cells that have a safe value specified. Safe values are driven into core inputs by any such input wrapper cells, and safe values are driven into the surrounding fanout logic by any such output wrapper

cells. There are no scan chains (internal or wrapper). Clock inputs to the core remain unaffected.

Figure 186 Safe Mode of Wrapper Chain



During core creation, DFT insertion does not mix wrapper chains and regular scan chains (although they can be compressed by the same codec).

During core integration at the top level, DFT insertion can mix core-level wrapper chains, core-level scan chains, and top-level scan chains for length-balancing purposes. Core-level scan chains are not included in top-level test modes that place the core in outward-facing mode.

See Also

- [SolvNet article 3017081, “How Does Core Wrapping Identify Clock Ports for Exclusion?”](#) for more information about how clock ports are identified

Wrapper Test Modes

When core wrapping is enabled, the `insert_dft` command creates the following core wrapping test modes by default:

- `wrp_if`

This is an inward-facing uncompressed scan mode. The wrapper chain is placed in the INTEST mode of operation. Both wrapper chains and internal core chains are active.

- `ScanCompression_mode`

This is an inward-facing compressed scan mode. The wrapper chain is placed in the INTEST mode of operation. Both wrapper chains and internal core chains are active and compressed by the scan compression codec. This mode is created only if scan compression is also enabled.

- wrp_of

This is an outward-facing uncompressed scan mode. Only the wrapper chain is active; it is placed in the EXTEST mode of operation.

- wrp_safe

This mode drives safe values from both input and output wrapper cells according to their safe value specifications. This mode is created only if at least one port has a safe value defined.

When core wrapping is enabled, DFT Compiler does not create the Internal_scan mode by default because it does not provide inward-facing or outward-facing hierarchical test capabilities.

You can also create user-defined core wrapping test modes. For more information, see [Creating User-Defined Core Wrapping Test Modes on page 491](#).

The Simple Core Wrapping Flow

The simple core wrapping flow provides basic core wrapping functionality, as described in the following topics:

- [Simple Core Wrapper Cells](#)
- [Simple Core Wrapper Chains](#)

Simple Core Wrapper Cells

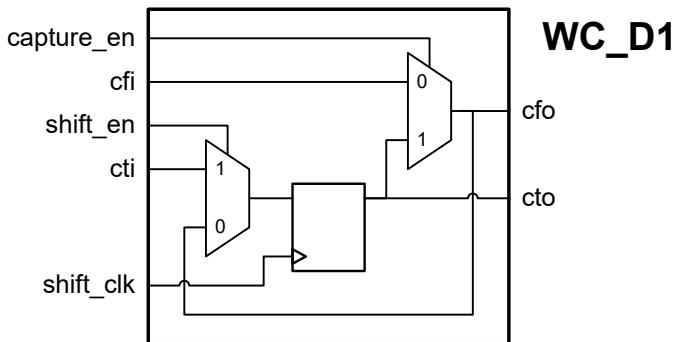
The simple core wrapping flow uses the following wrapper cell types:

- [Dedicated Wrapper Cell](#)
- [Dedicated Safe-State Wrapper Cell](#)
- [Shared-Register Wrapper Cells](#)

Dedicated Wrapper Cell

By default, the simple core wrapping flow uses the WC_D1 *dedicated wrapper cell* for core wrapping. A dedicated wrapper cell is a wrapper cell that uses its own internal dedicated flip-flop to provide controllability, observability, and shift capabilities. [Figure 187](#) shows the internal logic of the WC_D1 dedicated wrapper cell.

Figure 187 WC_D1 Dedicated Wrapper Cell



The interface to the WC_D1 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 wrapper cell in the chain.

cfi – Core functional input

For input wrapper cells, this input is fed from the logic surrounding the core. 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 logic surrounding the core.

shift_clk – Wrapper clock

This is usually driven by the `wrp_clock` signal in the core. It clocks the flip-flop in the wrapper cell.

shift_en – Shift enable

This is like a scan-enable signal for wrapper cells. When the signal is high, the wrapper clock shifts data through the cti and cto scan data pins. When the signal is low, the wrapper clock captures the functional input value or holds the current state, depending on the value of the `capture_en` signal. The shift enable signal can be controlled differently for input and output wrapper cells.

`capture_en` – Capture enable

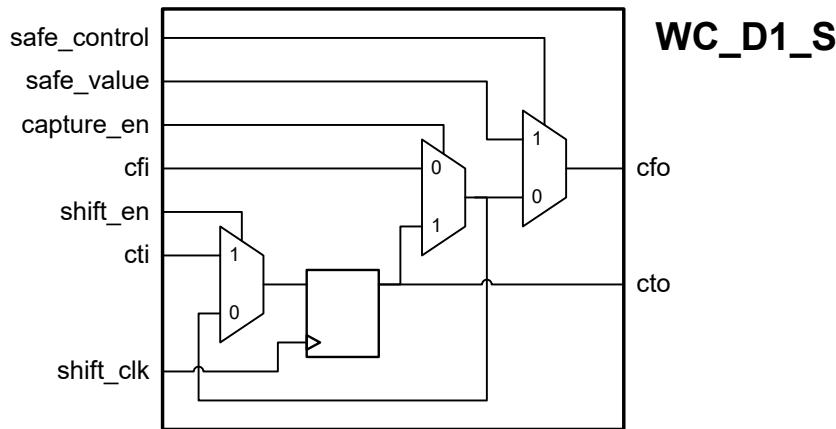
This signal controls what is captured when the wrapper cell is not shifting. When the signal is low, the wrapper clock captures the functional input value. When the signal is high, the wrapper clock holds the current wrapper cell state.

Dedicated Safe-State Wrapper Cell

A wrapper cell provides observability at its input, and controllability at its output. The same WC_D1 wrapper cell is used for both core inputs and core outputs. However, the controlled output of the wrapper cell can toggle as data is shifted through the wrapper chain. In some cases, if edge-triggered or level-sensitive logic exists in the fanout of the wrapper cell, unintended circuit operation can occur.

To avoid this, you can specify a *safe value* for a wrapper cell. DFT Compiler uses the WC_D1_S wrapper cell to implement the safe value capability. It contains an additional multiplexer at its output to drive a static safe logic value, enabled by a safe value control signal. [Figure 188](#) shows the internal logic of the WC_D1_S wrapper cell.

Figure 188 WC_D1_S Dedicated Wrapper Cell



The interface to the WC_D1_S wrapper cell consists of the same signals as the WC_D1 wrapper cell, plus the following additional signals:

`safe_control` – Control signal

This signal determines when the safe state value is driven at the output.

`safe_value` – Logic value

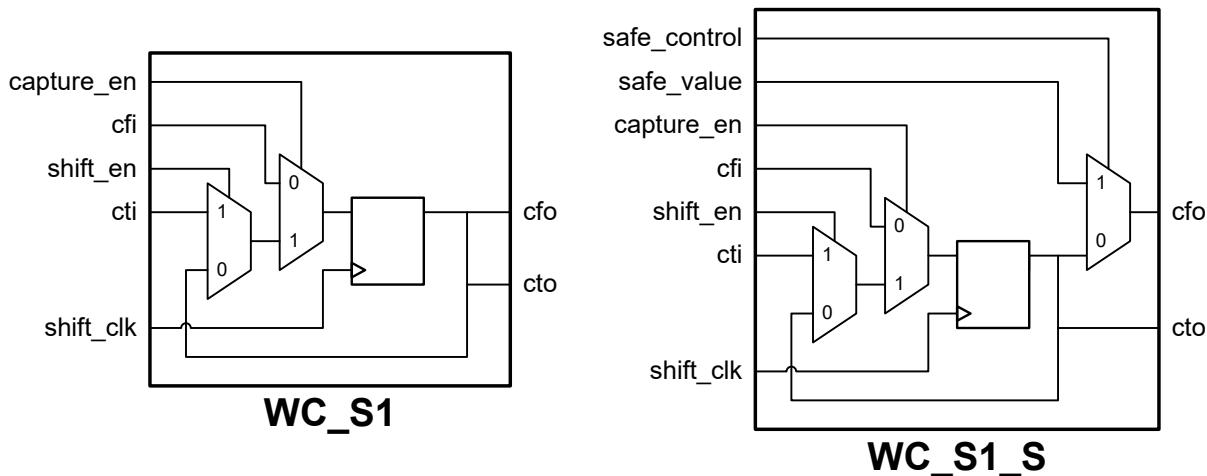
This signal specifies the safe state logic value.

Shared-Register Wrapper Cells

If your design has existing boundary I/O registers by the ports, you can share these functional registers with the wrapper cell logic to reduce the core wrapping area overhead.

Shared wrapper cells replace the existing functional register, providing equivalent functionality in functional mode. [Figure 189](#) shows the internal logic for the WC_S1 and WC_S1_S shared wrapper cells. The signals are identical to the signals used for dedicated wrapper cells.

Figure 189 WC_S1 and WC_S1_S Shared Wrapper Cells



In order for the core wrapping feature to share the existing functional register in the simple core wrapping flow, the I/O register and the port must meet the following conditions:

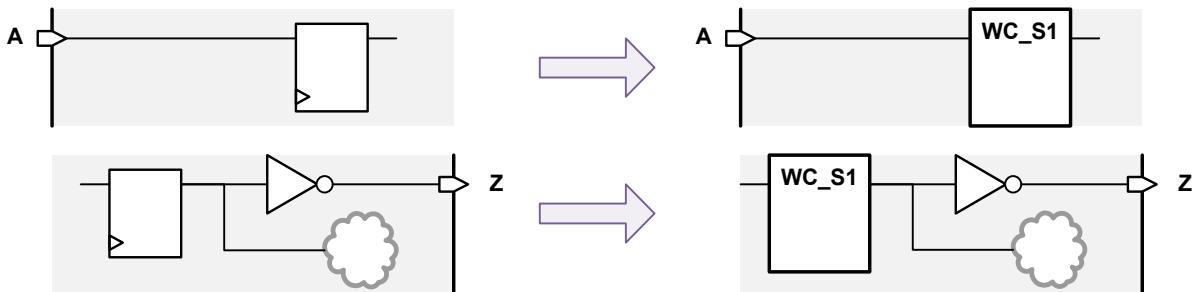
- The register's data input or output must be connected to a boundary port with a wire or logic path. The logic path must be sensitized to produce a buffering or inverting effect, and the sensitization must be controlled by a constant signal type (static value of 0 or 1) on a primary input or output.
- The shared registers must be clocked by a functional clock.

Note:

If the functional clock for the I/O registers also clocks other functional registers internal to the core, it can disturb the internal core logic during wrapper chain operation. You should either provide a separate functional clock for shared wrapper cells, or you can use the `-use_dedicated_wrapper_clock` option. For more information, see [Configuring Simple Core Wrapping on page 464](#).

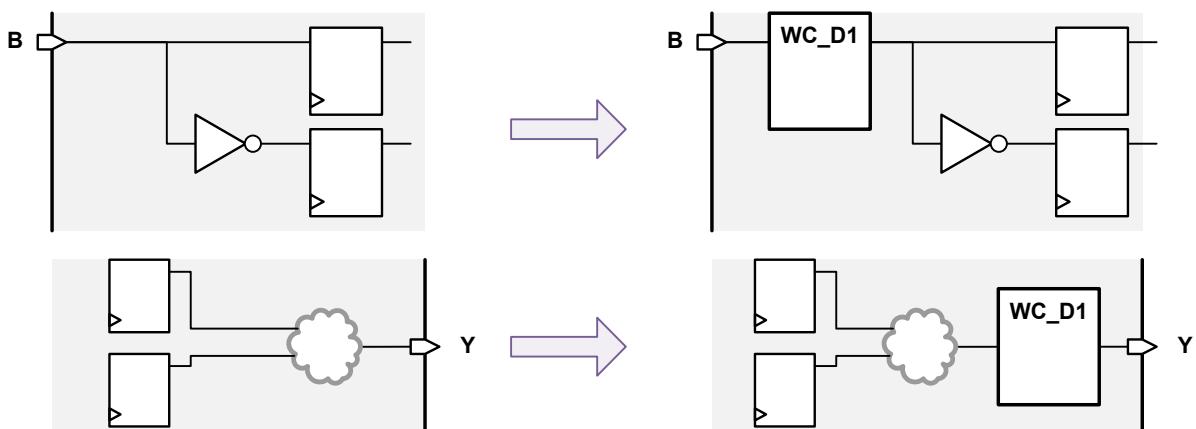
If you enable the shared wrapper cell style, DFT Compiler inserts shared wrapper cells wherever possible. [Figure 190](#) illustrates some cases where shared wrapper cells can be used. The shared wrapper cell is placed at the same hierarchical location as the existing design register.

Figure 190 Examples of Supported Design Register Sharing



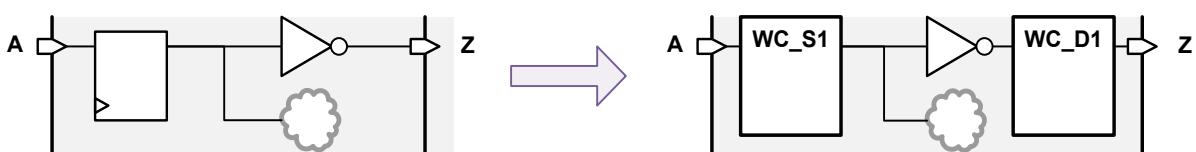
However, if a register does not meet the requirements, a dedicated wrapper cell is used instead. [Figure 191](#) illustrates some cases where shared wrapper cells cannot be used.

Figure 191 Examples of Unsupported Design Register Sharing



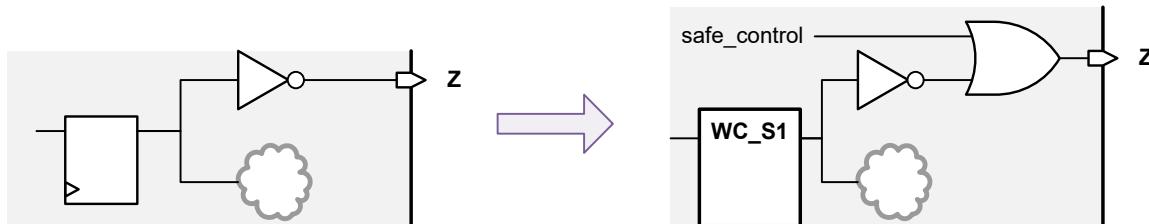
If a register qualifies as both an input shared register and an output shared register, the register becomes an input shared register, and a dedicated wrapper cell is placed at the output port. See [Figure 192](#).

Figure 192 Same Register Qualified as Input and Output Shared Register



If an output shared register has a safe state specified, a WC_S1_S wrapper cell is normally used. However, if the register's output drives internal logic in addition to the output port, the safe state logic inside a WC_S1_S wrapper cell would prevent the register from reliably driving the internal logic. When DFT Compiler detects this situation, it uses a WC_S1 shared wrapper cell and moves the safe state logic to the output port. See [Figure 193](#).

Figure 193 Output Shared Register With Safe State and Internal Fanout Connections

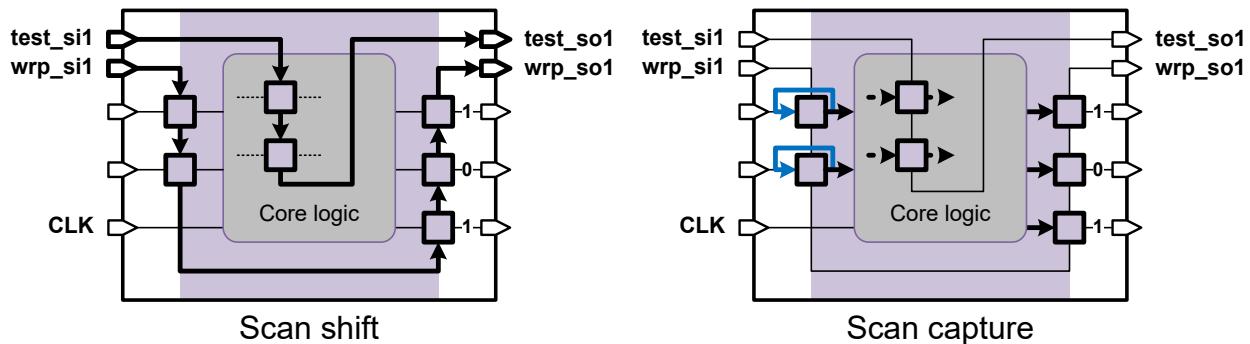


Simple Core Wrapper Chains

In the simple core wrapping flow, input and output wrapper cells can be placed in the same wrapper chain. The design can have a single wrapper chain or multiple wrapper chains.

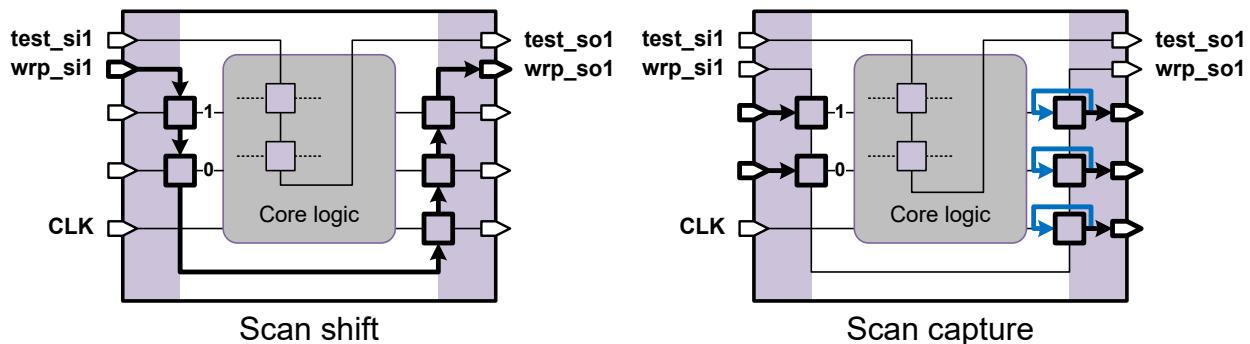
[Figure 194](#) shows the shift and capture behaviors used for inward-facing operation. In scan capture, the wrapper cell state-holding loops (shown in blue) are used to block external values at core inputs.

Figure 194 Inward-Facing Wrapper Chain Behaviors in the Simple Core Wrapping Flow



[Figure 195](#) shows the shift and capture behaviors used for outward-facing operation. In scan capture, the wrapper cell state-holding loops (shown in blue) are used to block core-driven values at core outputs.

Figure 195 Outward-Facing Wrapper Chain Behaviors in the Simple Core Wrapping Flow



The Maximized Reuse Core Wrapping Flow

The simple core wrapping flow adds dedicated wrapper cells when functional I/O registers are not directly connected to I/O ports through simple buffering or inverting logic. To reduce the timing and area impact of core wrapping, the TestMAX DFT tool also provides a *maximized reuse* mode that can share I/O registers that are connected to I/O ports through combinational logic.

The maximized reuse flow is described in the following topics:

- [Maximized Reuse Core Wrapper Cells](#)
- [Maximized Reuse Core Wrapper Chains](#)
- [Maximized Reuse Shift Signals](#)

Note:

A DFTMAX or TestMAX DFT license is required to use the maximized reuse core wrapping feature.

Maximized Reuse Core Wrapper Cells

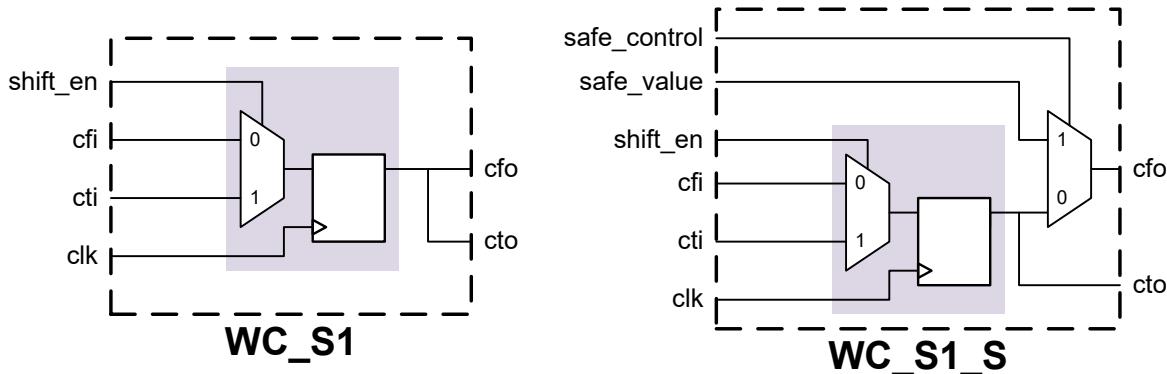
The maximized reuse core wrapping flow uses the following wrapper cell types:

- [Shared-Register Wrapper Cells](#)
- [Dedicated Wrapper Cells](#)

Shared-Register Wrapper Cells

By default, the maximized reuse flow uses *shared wrapper cells* for all ports that meet the sharing criteria. Figure 196 shows the internal logic for the maximized reuse WC_S1 (no safe state) and WC_S1_S (safe state) shared wrapper cells. Their logic structure differs from the same-named wrapper cells used in the simple core wrapper flow.

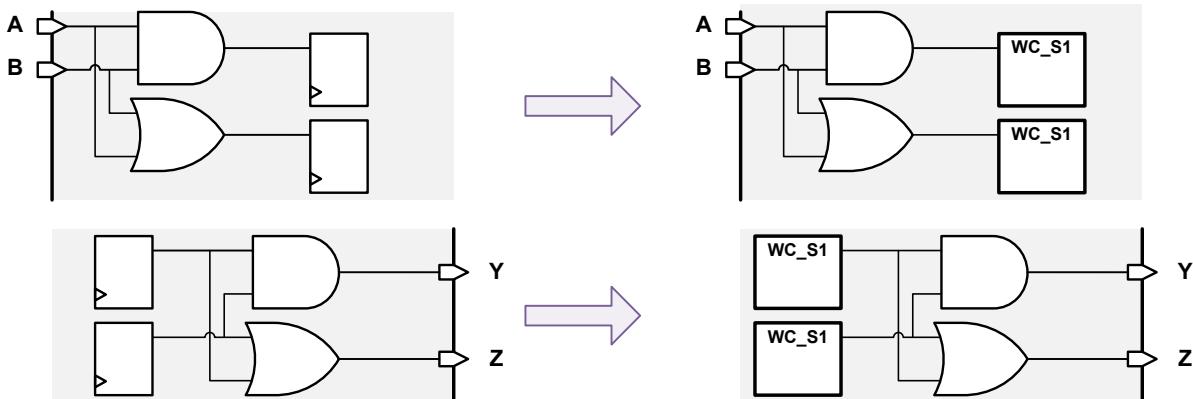
Figure 196 The WC_S1 and WC_S1_S Shared Wrapper Cells in Maximized Reuse Flow



To minimize area overhead, the maximized reuse flow uses an optimized shared wrapper cell design that has no state-holding loop. The wrapper cell uses an in-place wrapper register implementation, which means it has no hierarchy around it. The resulting wrapper cell functionality is implemented by a typical functional scan-equivalent flip-flop.

The maximized reuse flow also allows functional I/O registers connected to I/O ports through combinational logic to be shared. [Figure 197](#) shows functional I/O registers that are replaced by shared wrapper cells when maximized reuse is enabled.

Figure 197 Maximized Reuse Examples



Note:

The diagrams in this section show WC_S1 shared wrapper cell instances for clarity. However, the in-place register implementation used by maximized reuse core wrapping ensures that the names and locations of the wrapped functional registers are not disturbed.

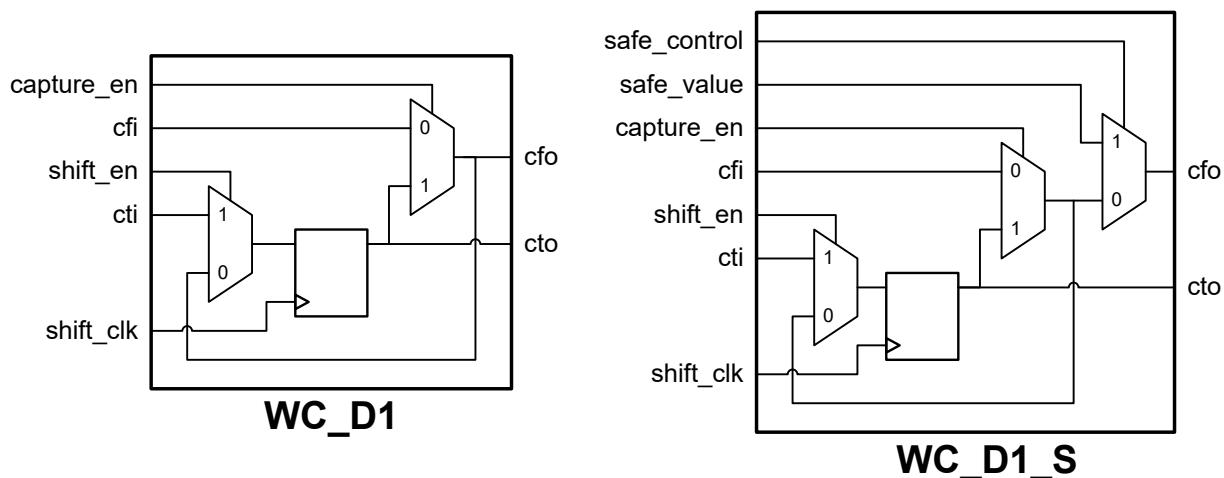
The maximized reuse flow provides count-based and logic-depth-based thresholds to limit how many functional registers can be wrapped for an I/O port.

Using existing functional registers as shared wrapper cells can reduce the area requirements for core wrapping. Any combinational logic between the shared wrapper cell and the ports is effectively placed outside the block as far as core wrapping logic is concerned. This logic must be tested using the EXTEST wrapper mode that exercises the surrounding logic.

Dedicated Wrapper Cells

In the maximized reuse flow, dedicated wrapper cells are used for I/O ports to be wrapped that exceed the sharing thresholds. [Figure 198](#) shows the internal logic for the WC_D1 and WC_D1_S dedicated wrapper cells. These are the same dedicated wrapper cell designs used in the simple core wrapper flow.

Figure 198 WC_D1 and WC_D1_S Wrapper Cells



Maximized Reuse Core Wrapper Chains

In the maximized reuse core wrapping flow, input and output wrapper cells are placed in separate wrapper chains with separate input and output wrapper shift-enable signals. Because shared wrapper cells have no state-holding loop in this flow, wrapper chains are kept in scan shift mode as needed to block values from being captured by wrapper cells.

[Figure 199](#) shows the shift and capture behaviors used for inward-facing operation. In scan capture, input wrapper chains are kept in scan shift (highlighted in blue) to block external values at core inputs.

Figure 199 Inward-Facing Wrapper Chain Behaviors in the Maximized Reuse Core Wrapping Flow

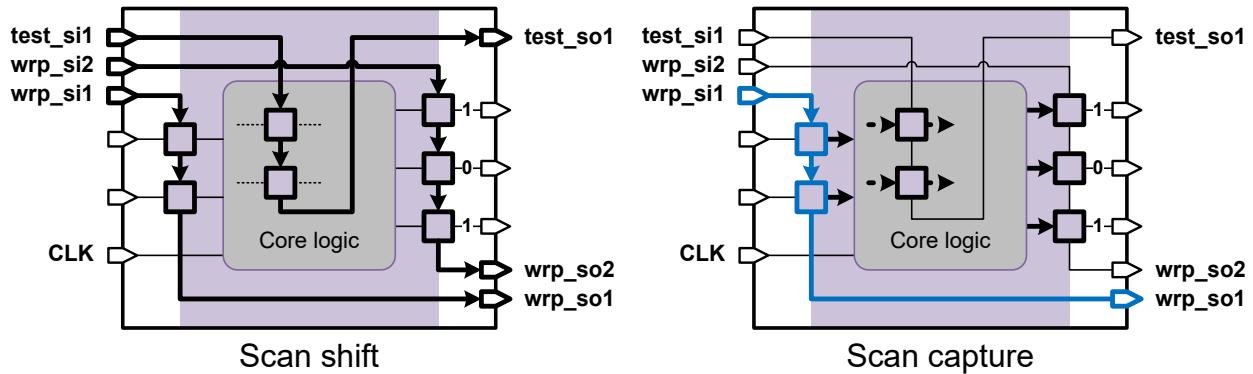
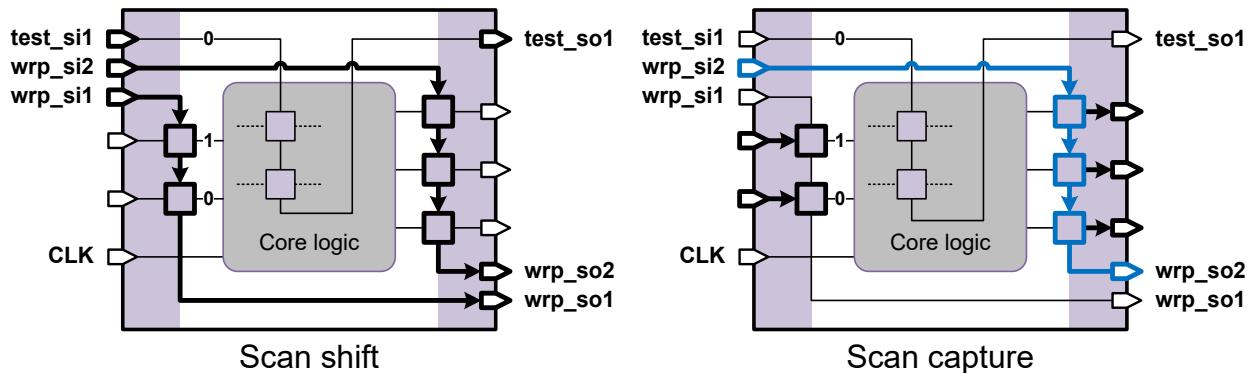


Figure 200 shows the shift and capture behaviors used for outward-facing operation. In scan capture, output wrapper chains are kept in scan shift (highlighted in blue) to block core-driven values at core outputs. Core wrapper chain scan-ins are driven with logic 0 to reduce power consumption.

Figure 200 Outward-Facing Wrapper Chain Behaviors in the Maximized Reuse Core Wrapping Flow



These shift and capture behaviors apply to all wrapper cells, shared and dedicated, in the input and output wrapper chains. The state-holding loops of any dedicated wrapper cells are not used.

In transition-delay ATPG, the highlighted wrapper chains that are kept in scan shift generate transitions by shifting the opposite value into a wrapper cell from the preceding wrapper cell.

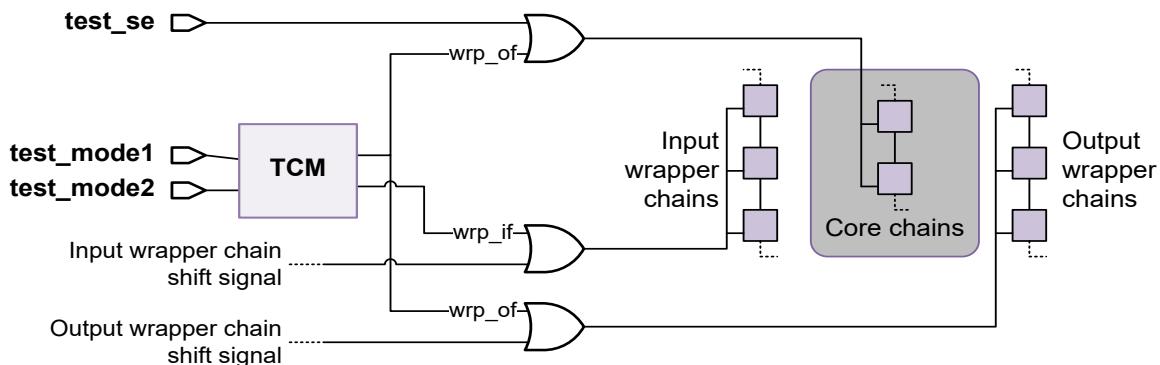
Maximized Reuse Shift Signals

In the maximized reuse flow, the scan-enable and wrapper shift signals are conditioned by the currently active test mode as follows:

- In inward-facing wrapper modes, the input wrapper shift signal is always asserted.
- In outward-facing wrapper modes, the output wrapper shift signal is always asserted.
- In outward-facing wrapper modes, the scan-enable signal for the internal core scan chains is always asserted (to load constant values into all scan chain elements).

[Figure 201](#) shows an example of the conditioning logic.

Figure 201 Scan-Enable and Wrapper Shift Signals in the Maximized Reuse Flow



The wrapper shift signals going to the input and output wrapper chains are conditioned separately as shown in the diagram, even when a single wrapper shift signal is used.

See Also

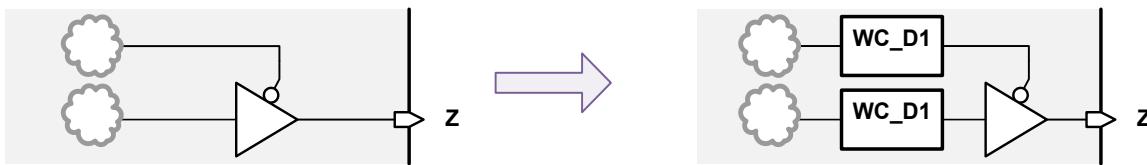
- [Defining Wrapper Shift Signals on page 459](#) for details on defining wrapper shift signals

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 inward from the driver, at the control and data signals.

For a three-state port, as shown in [Figure 202](#), wrapper cells are added to the data-out and control paths of the three-state driver or pad cell connected to the port. When the control path wrapper cell enables the three-state driver, the data-out path wrapper cell controls the output port value; otherwise, the data-out path wrapper cell has no effect.

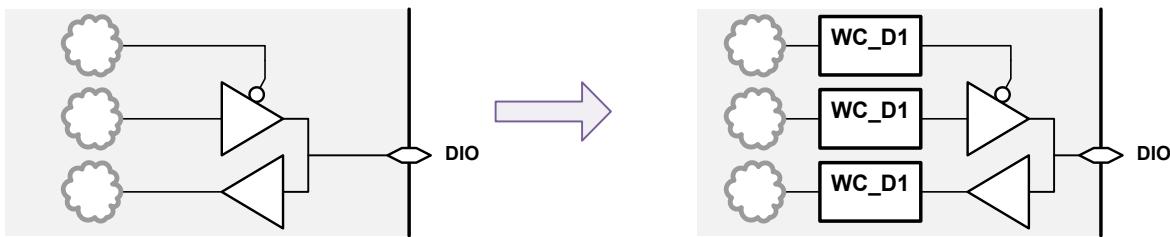
Figure 202 Three-State Port Wrapping



Three-state drivers always have a safe state that drives the output to a safe high-impedance state. Safe state specifications for three-state ports are ignored.

For a bidirectional port, as shown in [Figure 203](#), wrapper cells are added to the data-out and control paths as well as the data-in path of the bidirectional pad connected to the port. When the control path wrapper cell asserts the bidirectional output driver, the data-out path controls the port; otherwise, the data-in path wrapper cell is controlled by the port.

Figure 203 Bidirectional Port Wrapping



Bidirectional drivers always have a safe state that places the driver cell in output mode, with a data value driven by the output data wrapper cell. Safe state specifications for bidirectional ports are ignored.

The three-state and bidirectional control and data paths use shared or dedicated wrapper cells as determined by the current wrapping configuration. In the maximized reuse flow, the tool always uses a threshold of 1 for the driver control signals.

For degenerate three-state and bidirectional ports, in which the driver cell functionality is simplified using constant values, only the nonconstant signals are wrapped.

AutoFixing of three-state and bidirectional drivers is independent of wrapper cell insertion. AutoFixed drivers are still conditioned by the AutoFix logic during any scan shift activity (wrapper or core logic). For more information, see [Uncontrollable Three-State Bus Enable Signals on page 335](#).

Wrapping a Core

Core wrapping is performed during DFT insertion. Core wrapping configuration is described in the following topics:

- [Enabling Core Wrapping](#)
 - [Defining Wrapper Shift Signals](#)
 - [Defining Dedicated Wrapper Clock Signals](#)
 - [Configuring Global Wrapper Settings](#)
 - [Configuring Port-Specific Wrapper Settings](#)
 - [Controlling Wrapper Chain Count and Length](#)
 - [Configuring Simple Core Wrapping](#)
 - [Configuring Maximized Reuse Core Wrapping](#)
 - [Determining Power Domains for Dedicated Wrapper Cells](#)
 - [Using the set_scan_path Command With Wrapper Chains](#)
 - [Previewing the Wrapper Cells](#)
 - [Post-DFT DRC Rule Checks](#)
-

Enabling Core Wrapping

To use core wrapping, you must include the DesignWare dw_foundation.sldb synthetic library in your link library list. This synthetic library contains the wrapper cell designs.

```
dc_shell> set link_library {* my_tech_lib.db dw_foundation.sldb}
```

Then, to enable core wrapping after loading and linking the design, use the following command:

```
dc_shell> set_dft_configuration -wrapper enable
```

Defining Wrapper Shift Signals

The wrapper shift signal enables scan data to shift through wrapper chains, just as a scan-enable signal does for scan chains. You can use any of the following wrapper shift signal configurations:

- [Using a DFT-Created Wrapper Shift Signal](#)
- [Defining a Single Dedicated Wrapper Shift Signal](#)
- [Reusing an Existing Scan-Enable Signal as the Wrapper Shift Signal](#)
- [Defining Separate Input and Output Wrapper Shift Signals](#)
- [Defining Clock-Domain-Based Wrapper Shift Signals](#)
- [Defining Input and Output Clock-Domain-Based Wrapper Shift Signals](#)

Using a DFT-Created Wrapper Shift Signal

By default, DFT Compiler creates a single wrapper shift signal named `wrp_shift`.

Defining a Single Dedicated Wrapper Shift Signal

To define a single dedicated wrapper shift signal, define the signal source as a `wrp_shift` signal:

```
dc_shell> set_dft_signal -view spec -type wrp_shift \
    -port my_wrp_shift
```

Reusing an Existing Scan-Enable Signal as the Wrapper Shift Signal

To reuse an existing scan-enable signal as the wrapper shift signal, define the signal source as both a `ScanEnable` signal and `wrp_shift` signal:

```
dc_shell> set_dft_signal -view spec -type ScanEnable \
    -port my_test_se
dc_shell> set_dft_signal -view spec -type wrp_shift \
    -port my_test_se
```

Defining Separate Input and Output Wrapper Shift Signals

To use separate input and output wrapper shift signals for separate input and output wrapper chains, define two `wrp_shift` signals with the `set_dft_signal` command, then specify them with the `-input_shift_enable` and `-output_shift_enable` options of the `set_wrapper_configuration` command:

```
dc_shell> set_dft_signal -view spec \
    -type wrp_shift -port {wrp_ishift wrp_oshift}
dc_shell> set_wrapper_configuration -class core_wrapper \
    -mix_cells false \
```

```
-input_shift_enable wrp_ishift \
-output_shift_enable wrp_oshift
```

The `-input_shift_enable` wrapper shift signal is used for the input wrapper cells, and the `-output_shift_enable` wrapper shift signal is used for the output wrapper cells. You can specify only a single signal for each option.

Defining Clock-Domain-Based Wrapper Shift Signals

To define a clock-domain-based wrapper shift signal, which is used only for wrapper cells clocked by a particular clock, specify the test clock source with the `-connect_to` option of the `set_dft_signal` command. For example,

```
dc_shell> # define test clocks
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port {CLK1A CLK1B CLK2}

dc_shell> # define per-clock-domain wrapper shift signals
dc_shell> set_dft_signal -view spec -type wrp_shift \
    -port WRP_SHIFT1 -connect_to {CLK1A CLK1B}
dc_shell> set_dft_signal -view spec -type wrp_shift \
    -port WRP_SHIFT2 -connect_to {CLK2}
```

This syntax is similar to that used for clock-domain-based scan-enable signals.

For any wrapper cell clocks not included in a clock-domain-based wrapper shift signal definition, the tool uses the first-defined wrapper shift signal.

See Also

- [Defining Dedicated Scan-Enable Signals for Scan Cells on page 230](#) for related information about clock-domain-based scan-enable signals

Defining Input and Output Clock-Domain-Based Wrapper Shift Signals

In the simple wrapping flow, you cannot define input and output clock-domain-based wrapper shift signals.

In the maximized reuse flow, you can define such signals by using the `input_wrp_shift` and `output_wrp_shift` signal types. For details, see [Defining Input/Output Clock-Domain-Based Wrapper Shift Signals on page 477](#).

Defining Dedicated Wrapper Clock Signals

If a dedicated wrapper clock signal is needed, by default, DFT Compiler creates a wrapper clock signal named `wrp_clock`. The clock defaults to a 10 percent duty cycle (with the rising edge and falling edge at 45 percent and 55 percent of the default clock period, respectively). You can change the default timing by setting the `test_wrapper_new_wrp_clock_timing` variable.

If you have an existing port to use for this wrapper clock signal, you can define it with the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type wrp_clock \
    -port port_name
```

By default, this clock signal uses the default wrapper clock timing (10 percent duty cycle or as specified by the `test_wrapper_new_wrp_clock_timing` variable). You can also specify signal-specific timing by defining an `existing_dft` view of the wrapper clock signal:

```
dc_shell> set_dft_signal -view existing_dft -type wrp_clock \
    -timing {45 55} -port port_name
```

By default, DFT Compiler creates any test-mode ports needed to provide the test-mode encodings for the functional, scan, and wrapper modes. If you have existing ports to use for these test-mode signals, you can use the `set_dft_signal` to define them as `TestMode` signals.

Configuring Global Wrapper Settings

The `set_wrapper_configuration` command allows you to specify global configuration parameters that apply to the entire wrapper chain. You must use the `-class core_wrapper` option to configure core wrapping.

By default, DFT Compiler does not create safe state wrapper cells. To specify a safe state value for all wrapper cells in the wrapper chain, use the `-safe_state` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -safe_state 1
```

Configuring Port-Specific Wrapper Settings

The `set_wrapper_configuration` command applies to all ports in the design. To specify the wrapper cell characteristics of specific ports, you can use the `set_boundary_cell` command, which provides many of the same options as the `set_wrapper_configuration` command.

For example, to specify safe state values for specific ports, use the `-safe_state` option of the `set_boundary_cell` command:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -ports port_list -type WC_D1_S -safe_state 0 | 1
```

Note:

When using the `set_boundary_cell` command, you must explicitly provide the wrapper cell type in the specification. The specified type should match the safe state and shared register characteristics for that port. In the preceding example, a dedicated wrapper cell is used.

To prevent the insertion of wrapper cells for a specific list of ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -ports port_list -type none
```

This might be needed in cases where an output port drives downstream clock pins or asynchronous set or reset signals. If the output port is wrapped, toggle activity in the wrapper cell might cause unintended activity in the downstream logic. Since excluding ports from the wrapper chain reduces test coverage, you should use this capability only when necessary.

To specify a dedicated wrapper cell for ports that would otherwise use a shared wrapper cell, specify a WC_D1 or WC_D1_S wrapper cell with the `set_boundary_cell` command:

```
dc_shell> # no safe state:
dc_shell> set_boundary_cell -class core_wrapper \
    -ports port_list -type WC_D1

dc_shell> # safe state:
dc_shell> set_boundary_cell -class core_wrapper \
    -ports port_list -type WC_D1_S -safe_state safe_value
```

Note:

You cannot use the `set_boundary_cell` command to force a shared wrapper cell type to be used for a port if the I/O register does not meet the requirements for a shared wrapper cell or if sharing has not been enabled with the `-style shared` option.

To specify that a particular wrapper clock is to be used for the dedicated wrapper cells of specific ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper -type WC_D1 \
    -shift_clk clock_name -ports port_name
```

Controlling Wrapper Chain Count and Length

You can use the following options of the `set_wrapper_configuration` command to control the count or maximum length of the wrapper chains:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -chain_count W ;# wrapper chain count
```

```
dc_shell> set_wrapper_configuration -class core_wrapper \
-max_length X ;# wrapper chain maximum length
```

If you also specify a chain count with the `set_scan_configuration` command:

```
dc_shell> set_scan_configuration -chain_count N
```

then this value N is the total chain count for both wrapper chains and internal chains. The tool architects the wrapper chains first, then it architects the internal chains using the remainder of the total allocation N . Internal scan cells and wrapper cells cannot be mixed on the same chain.

If any of the following criteria are true:

- Input and output wrapper cell mixing is disabled (`set_wrapper_configuration -mix_cells false`)
- The maximized reuse flow is used, which disables input and output wrapper cell mixing
- User-specified wrapper scan path specifications exist (`set_scan_path -class core_wrapper`)

then the tool architects the wrapper chains first, then it architects the internal chains. In this case, wrapper chains and internal chains are not length-balanced by default. You should explicitly use both the `set_wrapper_configuration` and `set_scan_configuration` commands to configure an overall length-balanced configuration. In outward-facing modes, use only the `set_wrapper_configuration` command to configure the wrapper chains.

If none of these criteria are true, then the tool architects the wrapper chains and internal chains together. In this case, if you do not specify a count or maximum length specifically for the wrapper chains, the wrapper chains and internal chains are length-balanced together using the `set_scan_configuration -chain_count` or `-max_length` specification. However, you can still use the `set_wrapper_configuration` command to specify a particular wrapper chain count or maximum length.

If a chain count and chain length specification are both applied with the same command, the length requirement is used and the count requirement is ignored.

Wrapper chains follow the same clock mixing requirements as normal scan chains. To allow differently clocked wrapper cells to be mixed in the same wrapper chain, enable clock mixing with the `-clock_mixing` option of the `set_scan_configuration` command. For example,

```
dc_shell> set_scan_configuration -clock_mixing mix_clocks
```

This can improve length balancing. Lock-up latches are inserted between wrapper cells that do not use the same clock. The default for the `-clock_mixing` option is `no_mix`, which creates separate wrapper chains for each wrapper clock domain.

Configuring Simple Core Wrapping

Configuration of the simple core wrapping functionality is described in the following topics:

- [Configuring Dedicated Wrapper Cell Clocks](#)
- [Using Shared Wrapper Cells](#)
- [Configuring Shared Wrapper Cell Clocks](#)
- [Using In-Place Shared Wrapper Cells](#)
- [Creating Separate Input and Output Wrapper Chains](#)

Configuring Dedicated Wrapper Cell Clocks

By default, simple core wrapping uses dedicated wrapper cells that use a dedicated wrapper clock. However, you can use system clocks for dedicated wrapper cells by setting the following option:

```
dc_shell> set_wrapper_configuration \
           -use_system_clock_for_dedicated_wrp_cells enable
```

In this case, the tool attempts to identify and use the dominant clock domain associated with each port using the following rules.

- A port's dedicated wrapper cell uses the same clock as the flip-flops associated with that port.
- If the port is associated with flip-flops of multiple clock domains, the dominant clock is used.
- If a dominant clock is not found, any user-specified wrapper clock, defined using the `set_dft_signal -view spec -type wrp_clock` command, is used.
- If no user-specified wrapper clock has been defined, a dedicated wrapper clock is created and used.

Using Shared Wrapper Cells

By default, the simple core wrapping flow inserts dedicated wrapper cells to wrap input and output ports. If you have existing functional I/O registers that you want to use for shared wrapper cells, specify the `-style shared` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
           -style shared
```

When the shared wrapper cell style is enabled, by default, the tool uses dedicated wrapper cells as a fallback for any ports that do not meet the sharing criteria. To prevent the

fallback insertion of dedicated wrapper cells and leave these ports unwrapped instead, use the `-dedicated_cell_type none` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
           -style shared -dedicated_cell_type none
```

DFT Compiler automatically selects the type of wrapper cell (WC_D1, WC_D1_S, WC_S1, or WC_S1_S) based on the capabilities needed for each wrapper cell.

If the shared wrapper cells span multiple clock domains, the cells are placed in separate wrapper chains unless clock mixing is enabled.

Configuring Shared Wrapper Cell Clocks

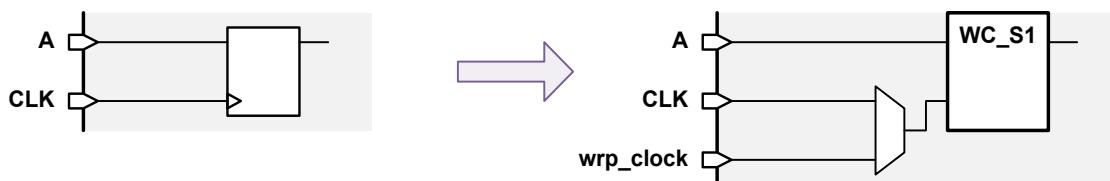
For shared wrapper cells, core wrapping keeps the register's existing functional clock signal. This can disturb the internal core logic during boundary cell operation. You should either provide a separate functional clock for shared wrapper cells, or use the `-use_dedicated_wrapper_clock` option of the `set_wrapper_configuration` or `set_boundary_cell` command:

```
dc_shell> # global:
dc_shell> set_wrapper_configuration -class core_wrapper \
           -style shared \
           -use_dedicated_wrapper_clock true

dc_shell> # per-port:
dc_shell> set_boundary_cell -class core_wrapper \
           -ports port_list -type WC_S1 \
           -use_dedicated_wrapper_clock true
```

When enabled, this option uses the dedicated wrapper clock signal when wrapper test modes are active, but retains the original functional clock signal for other modes. See [Figure 204](#).

Figure 204 MUXing Dedicated Wrapper Clocks for Shared Wrapper Cells



Using In-Place Shared Wrapper Cells

When a shared wrapper cell is used for a port, DFT Compiler replaces, or swaps, the entire I/O register for that port with a shared wrapper cell, as shown in [Figure 190 on page 450](#). This process introduces a level of hierarchy around the register and renames the register cell itself.

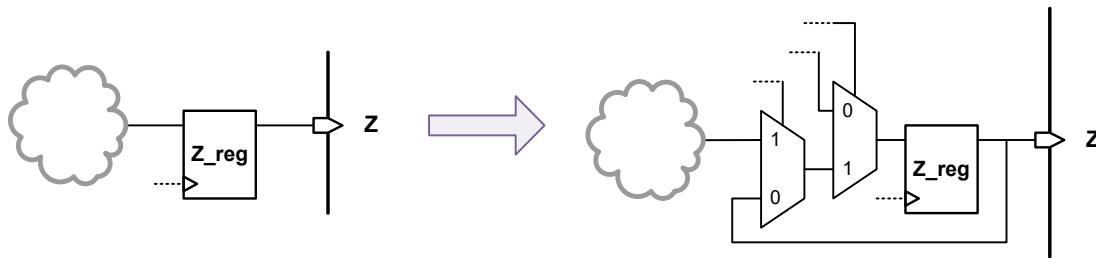
To preserve the original hierarchical instance path of the register, use the `-register_io_implementation in_place` option of the `set_wrapper_configuration` or `set_boundary_cell` command:

```
dc_shell> # global:
dc_shell> set_wrapper_configuration -class core_wrapper \
           -style shared \
           -register_io_implementation in_place

dc_shell> # per-port:
dc_shell> set_boundary_cell -class core_wrapper \
           -ports port_list -type WC_S1 \
           -register_io_implementation in_place
```

This option implements the shared wrapper cell functionality by using discrete logic gates around the existing I/O register, as shown in [Figure 205](#). The location of the original I/O register is not disturbed.

Figure 205 Shared Wrapper Cell Using In-Place Register Implementation



Creating Separate Input and Output Wrapper Chains

By default, the simple wrapper mode inserts a wrapper chain with the following characteristics:

- Input and output wrapper cells can be mixed on the same scan chain.
- Input and output wrapper cells share the same wrapper shift signal.

To prevent input and output wrapper cells from being mixed on the same scan chains, use the `-mix_cells false` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
           -mix_cells false
```

If you use a single wrapper shift signal, it is used for both the input and output wrapper chains. You can also define separate input and output wrapper chain shift signals, as described in [Defining Separate Input and Output Wrapper Shift Signals on page 459](#).

Configuring Maximized Reuse Core Wrapping

The maximized reuse feature considers how many registers exist in the fanout from an input port or the fanin to an output port. A single port might have many registers in its fanin or fanout, which would require many shared wrapper cells to fully wrap the port. If the number of fanin or fanout registers for a port exceeds a *reuse threshold* value, a single dedicated wrapper cell is used for that port.

Configuration of the maximized reuse core wrapping functionality is described in the following topics:

- [Enabling Maximized Reuse Core Wrapping](#)
- [Applying a Register Reuse Threshold](#)
- [Applying a Combinational Depth Threshold](#)
- [Specifying Port-Specific Maximized Reuse Behaviors](#)
- [Special Cases for Register Reuse](#)
- [Using Dedicated Wrapper Cells](#)
- [Configuring Dedicated Wrapper Cell Clocks](#)
- [Defining Input/Output Clock-Domain-Based Wrapper Shift Signals](#)
- [Including Additional Scan Cells in Input and Output Wrapper Chains](#)
- [Using the Pipelined Scan-Enable Feature](#)
- [Low-Power Maximized Reuse Features](#)
- [Hierarchical Core Wrapping](#)
- [Limitations of the Maximized Reuse Flow](#)

Enabling Maximized Reuse Core Wrapping

To enable the maximized reuse feature, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -maximize_reuse enable -reuse_threshold N
```

where the value of *N* defines the reuse threshold. The default of *N* is 1.

With the `-maximize_reuse` option enabled, when the number of registers encountered from an input or to an output exceeds the reuse threshold, a dedicated wrapper cell is added to the port. If the number of registers is less than this threshold, the tool can use these registers for wrapping the core. If a reuse threshold value of zero is specified, the tool converts all I/O registers to shared wrapper cells and no dedicated wrapper cells are

used unless specified by the user. For more information about how the reuse threshold is computed, see [Applying a Register Reuse Threshold on page 468](#).

When the `-maximize_reuse` option is enabled, the tool sets the following options automatically:

- `-style shared`
- `-register_io_implementation in_place`
- `-mix_cells false`
- `-use_system_clock_for_dedicated_wrp_cells enable`

Also, you should not use the following options of the `set_wrapper_configuration` command with the `-maximize_reuse` option:

- `-delay_test`
- `-core`
- `-shared_cell_type`
- `-shared_design_name`

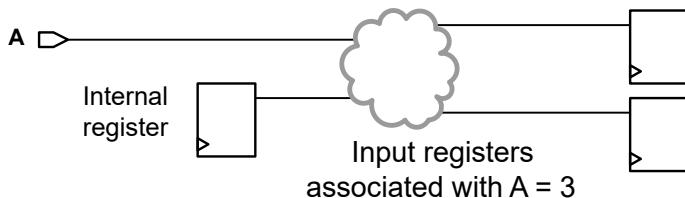
Applying a Register Reuse Threshold

When applying the reuse threshold value to a port, the tool determines the number of registers associated with that port. If the number of registers for a port does not exceed the reuse threshold, the registers are replaced with shared wrapper cells. If the number of registers exceeds the reuse threshold, a dedicated wrapper cell is placed at the port.

Computing Reuse Thresholds for Input Ports

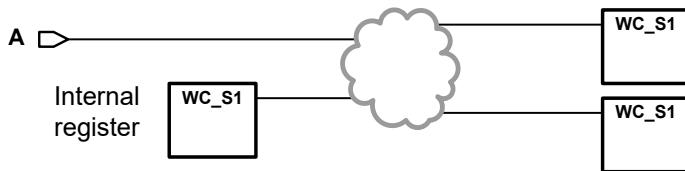
For an input port, the tool determines the number of registers in the fanout of the port. In addition, *it includes any internal registers that feed data pins or synchronous set/reset pins of the fanout registers*. [Figure 206](#) shows an input port with an associated register count of three.

Figure 206 Input Port Register Count Computation Example



In this example, if the reuse threshold is set to a value of three or higher, the tool replaces the registers with shared wrapper cells, as shown in [Figure 207](#).

Figure 207 Input Port Registers After Wrapper Cell Replacement



The tool places the shared wrapper cells for the fanout registers in the input wrapper chain. Because any internal registers feeding these fanout registers capture values from the core logic instead of input ports, the tool places the shared wrapper cells for these internal registers in the output wrapper chain.

In the maximized reuse flow, dedicated wrapper cells are added to input ports according to the following rules:

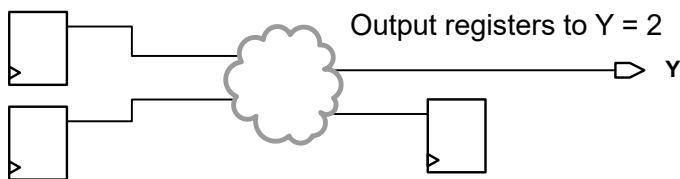
- If the sum of the input fanout registers and the internal registers feeding them exceeds the reuse threshold, then a dedicated wrapper cell is added to the input port.
- When the input register cells exceed the reuse threshold for a bidirectional port, dedicated wrapper cells are added to the data-out, enable, and data-in paths.
- If an input port is associated with a CTL-modeled cell, then a dedicated wrapper cell is added to the port. See [Wrapping Ports Associated With CTL-Modeled Cells on page 473](#) for details.

A warning is issued if all registers associated with an input port do not use the same clock.

Computing Reuse Thresholds for Output Ports

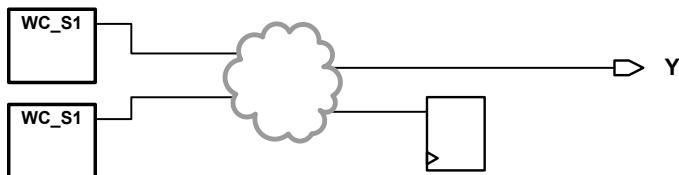
For an output port, the tool determines the number of registers in the fanin of the port. It does not include any additional surrounding registers. [Figure 208](#) shows an output port with an associated register count of two.

Figure 208 Output Port Register Count Computation Example



In this example, if the reuse threshold is set to a value of two or higher, the tool replaces the registers with shared wrapper cells, as shown in [Figure 209](#).

Figure 209 Output Port Registers After Wrapper Cell Replacement



The tool places the shared wrapper cells for the fanin registers in the output wrapper chain.

In the maximized reuse flow, dedicated wrapper cells are added to output ports according to the following rules:

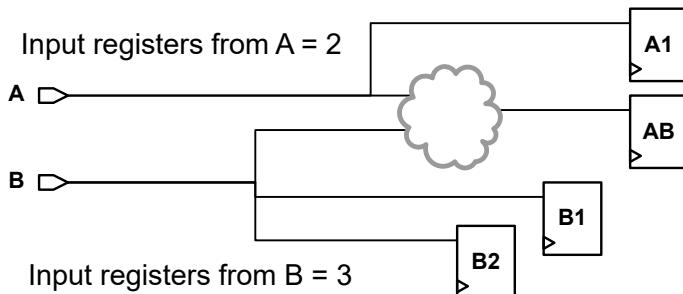
- If the sum of the output fanin registers exceeds the reuse threshold, then a dedicated wrapper cell is added to the output port.
- When the number of the output register cells exceeds the reuse threshold value for a three-state output port, dedicated wrapper cells are added to both data-out and enable paths.
- If an output port is associated with a CTL-modeled cell, then a dedicated wrapper cell is added to the port. See [Wrapping Ports Associated With CTL-Modeled Cells on page 473](#) for details.
- For three-state and bidirectional ports, only a functional register that directly controls the pad is considered as a shared wrapper cell for the enable path of the pad. There can be inverting or noninverting buffers between the register and the pad enable pin. If no such register is found, a dedicated wrapper cell is added at the driver cell enable pin to control the port.

A warning is issued if all registers associated with an output port do not use the same clock.

Registers Associated With Multiple Ports

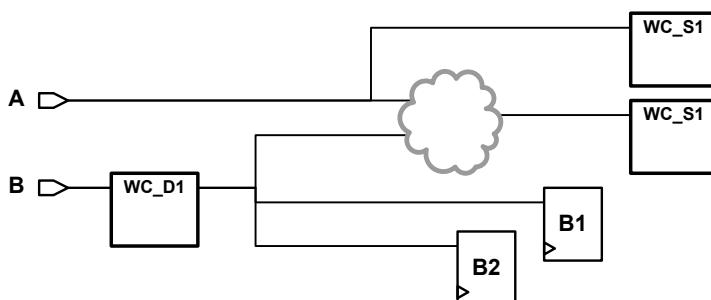
If the same register is associated with multiple ports, the register is replaced with a shared wrapper cell if *any* port meets the threshold. In [Figure 210](#), register AB is in the fanout of ports A and B. Input port A has two associated registers, and input port B has three associated registers.

Figure 210 Register in Fanout of Multiple Ports



In this example, if the reuse threshold is set to a value of two, the tool replaces all registers for input port A with shared wrapper cells, and it inserts a dedicated wrapper cell at input port B, as shown in [Figure 211](#). Register AB is replaced with a shared wrapper cell to completely wrap port A, even though it is also in the fanout of the dedicated wrapper cell for input port B.

Figure 211 Register in Fanout of Multiple Ports After Wrapper Cell Replacement



Registers that become shared wrapper cells for one port do not affect the associated register count for other ports. In [Figure 211](#), the associated register count for input port B is three even though register AB is wrapped for input port A.

If a register is classified as both an input shared register and an output shared register, the register becomes an input shared register and is removed from the output shared register list. Unlike the simple core-wrapping flow, no dedicated wrapper cell is placed at the output port.

Applying a Combinational Depth Threshold

For a core-wrapped design, any logic that exists between the wrapper chain and the I/O ports becomes logic that is external to the block for testing purposes. The `-reuse_threshold` option of the `set_wrapper_configuration` command applies a maximum fanout-breadth or fanin-breadth threshold limit for input and output shared wrapper cells, respectively. However, this reuse threshold only indirectly limits the logic depth that can exist outside the wrapper chain.

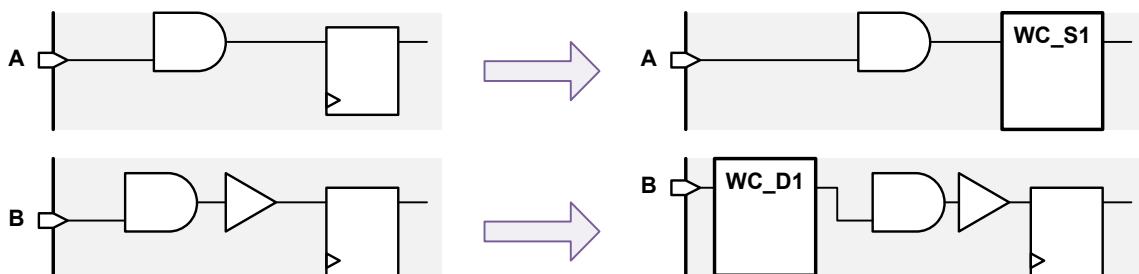
You can use the `-depth_threshold` option of the `set_wrapper_configuration` command to directly specify the maximum number of combinational cells, including buffers and inverters, that can exist between a port and its associated registers. For example,

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -depth_threshold 2
```

If this depth is exceeded, a dedicated wrapper cell is used for that port. This can be used to prevent too much logic from being placed on the external side of the wrapper chain.

[Figure 212](#) shows the wrapper cell insertion behavior when the `-depth_threshold` value is set to 1.

Figure 212 Core Wrapping With a Combinational Depth Threshold Value of 1



Specifying Port-Specific Maximized Reuse Behaviors

The reuse threshold can be set on a port-by-port basis with the `set_boundary_cell` command:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -reuse_threshold N -ports {port_list}
```

This command sets the reuse threshold value to N (where $N \geq 0$) for all ports in the `port_list` specification.

To ignore the reuse threshold and force the use of a dedicated wrapper cell for one or more ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -ports port_list -type WC_D1
```

If a port exceeds the reuse threshold and you want to manually specify some port-associated registers as shared wrapper cells, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -include {IO_register_cell_list} -ports {port_name}
```

This command causes the cells listed in `IO_register_cell_list` associated with the port `port_name` to be used as shared wrapper cells.

To manually specify some registers as shared wrapper cells for all associated ports that exceed the reuse threshold, omit the `-ports` option:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -include {IO_register_cell_list}
```

When using the `-include` option with the `-ports` option, any specified ports exceeding the reuse threshold do not get a dedicated wrapper cell. When using the `-include` option without the `-ports` option, all ports associated with the specified registers exceeding the reuse threshold do not get a dedicated wrapper cell. You can exceed the reuse threshold when manually specifying shared wrapper cell registers with this option.

If a port meets the reuse threshold and you want to manually exclude some port-associated registers as shared wrapper cells, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper
    -exclude {IO_register_cell_names} -ports {port_name}
```

This command causes the registers listed in `IO_register_cell_names` to be excluded for the port `port_name`. If a register is excluded for the specified port but qualifies to be wrapped for another port, the register is still wrapped.

To manually exclude some registers as shared wrapper cells for all associated ports that meet the reuse threshold, omit the `-ports` option:

```
dc_shell> set_boundary_cell -class core_wrapper
    -exclude {IO_register_cell_names}
```

Note:

The `-include` and `-exclude` options of the `set_boundary_cell` command can cause a port to become partially wrapped.

Special Cases for Register Reuse

In some cases, ports are associated with logic constructs that require special-case handling for register reuse.

Wrapping Ports Associated With CTL-Modeled Cells

If your design contains CTL-modeled synchronizer registers, they can be reused as shared wrapper cells. See [Wrapping Cores With Synchronizer Registers on page 499](#).

Other CTL-modeled cells, such as memories or DFT-inserted cores, cannot be used as shared wrapper cells. When a port is associated with a CTL-modeled cell, as shown in the examples in [Figure 213](#) and [Figure 214](#), a dedicated wrapper cell is added to the port.

Figure 213 Core Wrapping of Input Ports Associated With CTL-Modeled Cells

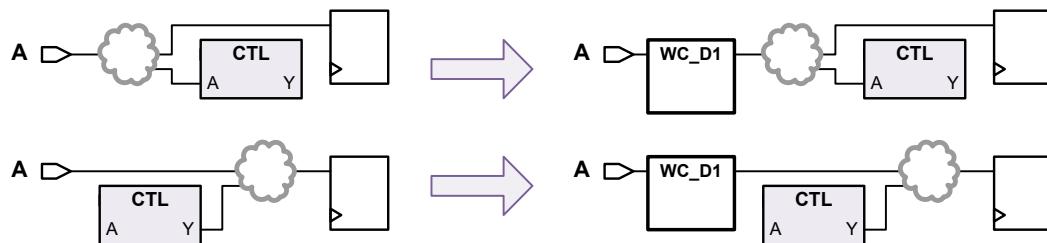
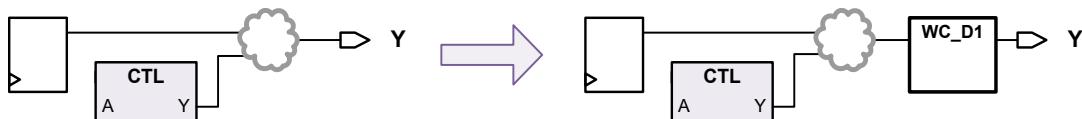


Figure 214 Core Wrapping of Output Ports Associated With CTL-Modeled Cells



In addition, a warning message is issued:

Warning: Port 'DACK' is connected to cell 'IP_CORE', which is represented by a CTL model; a dedicated wrapper cell is added to the port.
(TEST-1184)

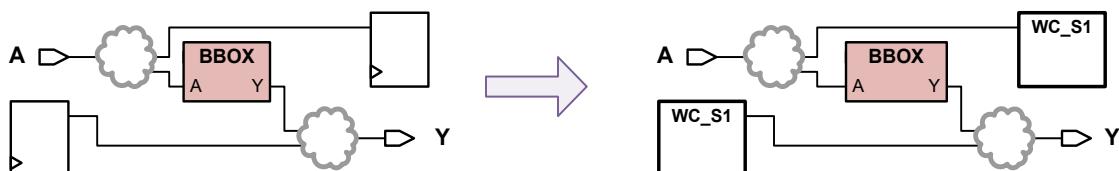
The scan clocks of the CTL-modeled cell are considered when determining the dominant system clock to use for the dedicated wrapper cell.

If the CTL-modeled cell has a netlist and there is no path from the port to a register inside the CTL-modeled cell, this behavior does not apply. In other words, connections to clock, reset, or DFT-related pins of a CTL-modeled cell do not force a dedicated wrapper cell.

Port Wrapping and Other Black-Box Cells

Black-box cells without CTL models do not affect core wrapping. Wrapper cells are not added unless needed by other design logic. See [Figure 215](#).

Figure 215 Black-Box Cell With No CTL Model



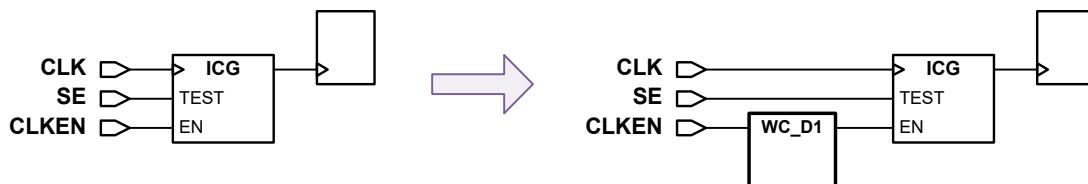
Note:

If LogicBIST self-test is enabled, dedicated wrapper cells are added as described in [Configuring Wrapper Chain Isolation Logic on page 1037](#).

Wrapping Ports Associated With Clock-Gating Cells

A dedicated wrapper cell is added to an input port that drives the enable signal of a clock-gating cell. In [Figure 216](#), an integrated clock-gating cell has both a functional enable pin and a test-mode pin. A dedicated wrapper cell is added for the functional enable signal driven by input port CLKEN. No wrapper cells are inserted for the clock signal or the global test-mode signal.

Figure 216 Integrated Clock Gating Cell Enable Signals



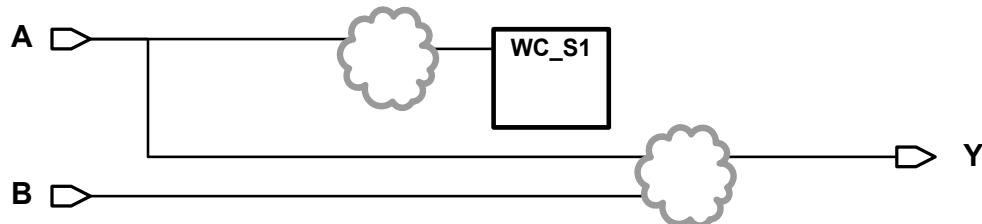
A warning message is issued if such a clock-gating enable signal is detected:

Warning: Port 'CLKEN' is connected to a clock gating cell 'UICG';
a dedicated wrapper cell is added to the port. (TEST-1183)

Port Wrapping and Feedthrough Paths

Feedthrough paths do not affect core wrapping. Wrapper cells are not added unless needed by other design logic. See [Figure 217](#).

Figure 217 Combinational Feedthrough Paths



An information message is issued if a feedthrough port is detected and no dedicated wrapper cell is manually specified for the port:

Information: No I/O registers are found for port 'B';
not adding any dedicated wrapper cells to the port. (TEST-1180)

Information: No I/O registers are found for port 'Y';
not adding any dedicated wrapper cells to the port. (TEST-1180)

Note:

If LogicBIST self-test is enabled, dedicated wrapper cells are added as described in [Configuring Wrapper Chain Isolation Logic on page 1037](#).

Using Dedicated Wrapper Cells

Dedicated wrapper cells are used for any ports that exceed the reuse or combinational depth thresholds. You can also manually force the use of a dedicated wrapper cell with the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
    -ports port_list -type WC_D1
```

In the simple core-wrapping flow, dedicated wrapper cells use the dedicated wrapper clock. However, in the maximized reuse flow, the tool attempts to identify the dominant clock domain associated with the port and uses that clock for the dedicated wrapper cell.

The following rules guide the automatic selection of a clock for dedicated wrapper cells in the maximized reuse flow:

- A port's dedicated wrapper cell uses the same clock as the flip-flops associated with that port.
- If the port is associated with flip-flops of multiple clock domains, the dominant clock is used.
- If a dominant clock is not found, any user-specified wrapper clock, defined using the `set_dft_signal -view spec -type wrp_clock` command, is used.
- If no user-specified wrapper clock has been defined, a dedicated wrapper clock is created and used.

Configuring Dedicated Wrapper Cell Clocks

There are two ways to override automatic clock selection for dedicated wrapper cells:

- To specify a particular wrapper clock to be used for the dedicated wrapper cells of specific ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper -type WC_D1 \
    -shift_clk clock_name -ports port_name
```

- To specify that a dedicated wrapper clock is to be used for all dedicated wrapper cells, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -use_system_clock_for_dedicated_wrp_cells disable
```

See Also

- [Defining Dedicated Wrapper Clock Signals on page 460](#) for more information about defining dedicated wrapper clock signals

Defining Input/Output Clock-Domain-Based Wrapper Shift Signals

Two signal types, `input_wrp_shift` and `output_wrp_shift`, allow you to define clock-domain-based input and output wrapper shift signals. For example,

```
# define per-clock-domain input wrapper shift signals
set_dft_signal -view spec -type input_wrp_shift \
    -port WRP_ISHIFT1 -connect_to CLK1
set_dft_signal -view spec -type input_wrp_shift \
    -port WRP_ISHIFT2 -connect_to CLK2

# define per-clock-domain output wrapper shift signals
set_dft_signal -view spec -type output_wrp_shift \
    -port WRP_OSHIFT1 -connect_to CLK1
set_dft_signal -view spec -type output_wrp_shift \
    -port WRP_OSHIFT2 -connect_to CLK2
```

Note:

The `input_wrp_shift` and `output_wrp_shift` signal types can be used only with the `-connect_to` option as part of a clock-domain-based signal specification, and they can be used only in the maximized-reuse flow.

See Also

- [SolvNet article 2138931, “Why Are There Two Ways to Specify Input and Output Wrapper Shift Signals?”](#) for more information about input and output wrapper shift signals

Including Additional Scan Cells in Input and Output Wrapper Chains

When wrapping a core, you can include additional scan cells in the input and output wrapper chains by using the `-input_wrapper_cells` and `-output_wrapper_cells` options, respectively. For example,

```
dc_shell> set_wrapper_configuration \
    -maximize_reuse enable \
    -input_wrapper_cells \
        [get_object_name [get_cells IN_CFG*reg]] \
    -output_wrapper_cells \
        [get_object_name [get_cells OUT_CFG*reg]]
```

The specified cells are reclassified from internal scan cells to shared wrapper cells in all test modes. This feature requires that the maximized-reuse feature also be enabled.

Note the following limitations of these options:

- The input is accepted as a simple list. Wildcards and collections are not supported, although you can use the `get_object_name` command to convert a collection to a list.
- No checking is performed for invalid object specifications.

Using the Pipelined Scan-Enable Feature

The pipelined scan-enable feature can be used in the maximized reuse flow. By default, two signals are used to enable shifts for designs using wrapper cells:

- `test_se` – Pipelined, used for internal flip-flops
- `wrp_shift` – Pipelined, used for wrapper flip-flops (shared or dedicated)

When the pipelined scan-enable feature is enabled, pipelined scan-enable structures are added to both of these signals. A pipelined scan-enable cell is used for each clock domain used by the input and output wrapper cells, with pipeline registers clocked by that clock. Pipelined scan-enable cells are not shared between input and output wrapper cells.

For example, if the design has two clock domains driving input wrapper cells and three clock domains driving output wrapper cells and if the number of wrapper cells within each of these domains meets the required pipeline fanout limit, a total of five pipelined scan-enable cells are created. Two of the pipelined scan-enable cells drive input wrapper cells; the remaining three pipelined scan-enable cells drive output wrapper cells. To create this pipelined structure, use the following commands:

```
dc_shell> set_dft_configuration -wrapper enable
dc_shell> set_wrapper_configuration -class core_wrapper \
           -maximize_reuse enable
dc_shell> set_scan_configuration \
           -pipeline_scan_enable true -pipeline_fanout_limit P
```

To use the same scan-enable for all registers, define the signal as both a scan-enable signal and a wrapper shift signal using the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type ScanEnable -port SE
dc_shell> set_dft_signal -view spec -type wrp_shift -port SE
dc_shell> set_dft_configuration -wrapper enable
dc_shell> set_wrapper_configuration -class core_wrapper \
           -maximize_reuse enable
dc_shell> set_scan_configuration \
           -pipeline_scan_enable true -pipeline_fanout_limit P
```

In this case, pipelined scan-enable cells are not shared between internal cells, input wrapper cells, or output wrapper cells.

To use separate scan-enables for input and output wrapper chains, use the following commands:

```
dc_shell> set_dft_signal -view spec -type ScanEnable -port SE
dc_shell> set_dft_signal -view spec -type wrp_shift -port WSE_I
dc_shell> set_dft_signal -view spec -type wrp_shift -port WSE_O
dc_shell> set_dft_configuration -wrapper enable
dc_shell> set_wrapper_configuration -class core_wrapper \
    -maximize_reuse enable \
    -input_shift_enable WSE_I -output_shift_enable WSE_O
dc_shell> set_scan_configuration \
    -pipeline_scan_enable true -pipeline_fanout_limit P
```

Note:

A single global_pipe_se signal is used for both internal and wrapper chains for all of these scenarios.

See Also

- [The Pipelined Scan-Enable Architecture on page 346](#) for more information about the pipelined scan-enable feature

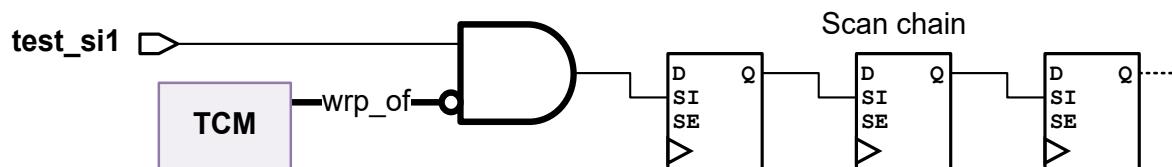
Low-Power Maximized Reuse Features

The following topics describe low-power features available in the maximized reuse flow.

Loading Constant Core Scan Data in EXTEST Mode

In the maximized reuse flow, to minimize power consumption in wrp_of (EXTEST) mode, all core scan chain cells are loaded with logic 0. This reduces toggle activity inside the block during outward-facing (EXTEST) modes. To accomplish this, the logic shown in [Figure 218](#) drives the first scan cell inputs of the internal scan chains.

Figure 218 Scan Data Gating Logic for Low-Power EXTEST



This feature is part of the wrapped core scan architecture; there is no option that controls it.

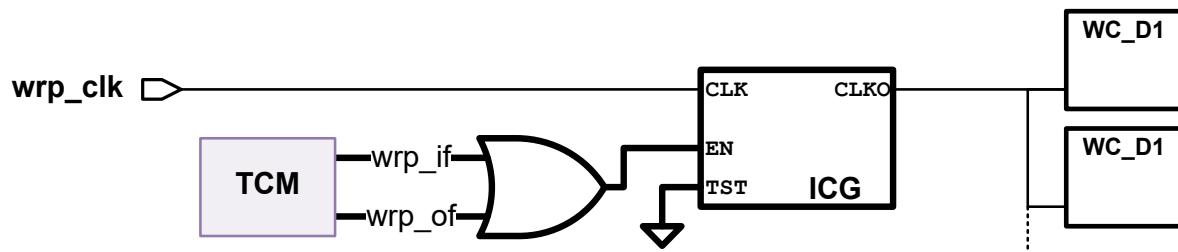
Gating Dedicated Wrapper Cell Clocks in Non-Wrapper Modes

The tool can insert clock-gating cells to disable the clock used for dedicated wrapper cells when a wrapper mode is not active. To do this, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -gate_dedicated_wrapper_cell_clk enable
```

The tool adds a clock-gating cell to each dedicated wrapper cell clock source, including any scan clocks used for dedicated wrapper cells, as shown in [Figure 219](#). Shared wrapper cells and scan cells are not affected, even if they use the same clock as a dedicated wrapper cell.

Figure 219 Clock Gating Logic for Low-Power Dedicated Clock Operation



By default, the clock-gating logic uses discrete latch cells. To use an integrated clock-gating cell instead, set the desired library cell reference (without the library name) by using the `test_icg_p_ref_for_dft` variable.

Gating Scan and Wrapper Cell Clocks in Wrapper Modes

The tool can use integrated clock-gating cells to disable the clock to unused scan and wrapper cells in the `wrp_of` (EXTEST) and `wrp_safe` (SAFE) wrapper modes, as shown in [Table 45](#).

Table 45 Gated-Clock Behaviors in Wrapper and Non-Wrapper Test Modes

Test mode	Clock disabled for input wrapper cells?	Clock disabled for scan cells?	Clock disabled for output wrapper cells?
INTEST (<code>wrp_if</code>)	No	No	No
EXTEST (<code>wrp_of</code>)	No	Yes	No
SAFE (<code>wrp_safe</code>)	Yes	Yes	Yes
All other test modes	No	No	No

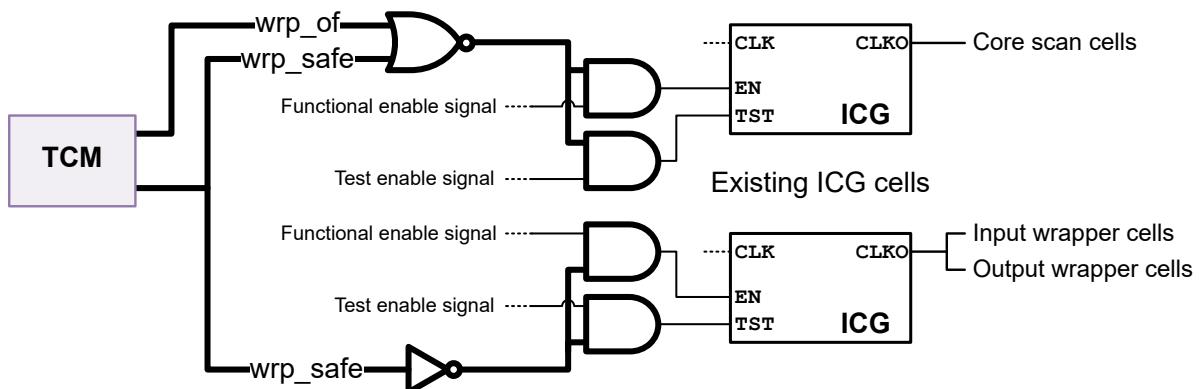
Test mode	Clock disabled for input wrapper cells?	Clock disabled for scan cells?	Clock disabled for output wrapper cells?
Mission mode	No	No	No

To modify only existing integrated clock-gating cells in the design, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
-gate_cells existing_cg
```

[Figure 220](#) highlights the logic added to existing clock-gating cells by DFT insertion. The functional enable and test enable signals are both deasserted to disable the clock. The `existing_cg` mode leaves the clock tree unchanged.

Figure 220 Clock-Disabling Logic for Existing Clock-Gating Cells



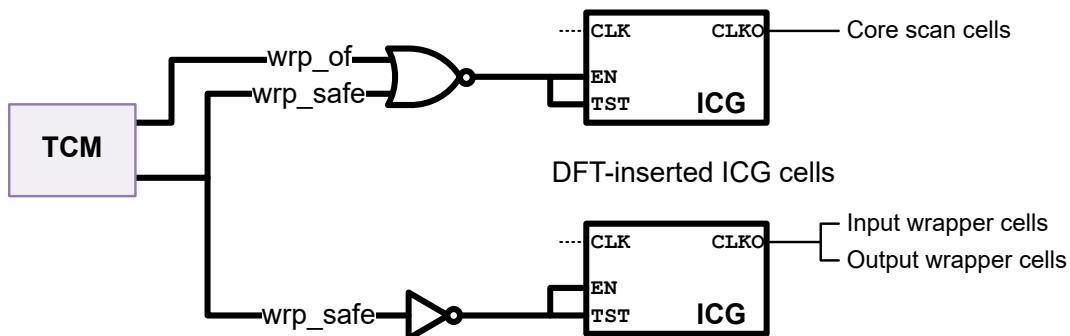
If an existing clock-gating cell gates multiple cell types, that gated clock signal is disabled only when all downstream clocked cells can be gated.

To apply clock-disabling logic to all wrapper and scan cells in the design for more aggressive power reduction, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
-gate_cells all
```

Existing integrated clock-gating cells are modified as previously described. In addition, new integrated clock-gating cells are added to ungated wrapper and scan cells. [Figure 221](#) highlights the clock-gating cells added by DFT insertion. The tool inserts a clock-gating cell for each group of same-edge cells clocked by a common hierarchical net. It does not trace backward through buffers and inverters to find logically identical nets.

Figure 221 Clock-Disabling Logic Added for Ungated Cells



You must specify the desired integrated clock-gating library cell references (without the library name) using the `test_icg_p_ref_for_dft` variable for rising-edge cells and the `test_icg_n_ref_for_dft` variable for falling-edge cells (if present). Otherwise, DFT insertion does not add clock-gating logic.

Note the following limitations of low-power wrapper and scan cell clock gating:

- Multiple levels of existing clock-gating cells are not supported.
- Falling-edge scan cells use the `test_icg_p_ref_for_dft` variable instead of the `test_icg_n_ref_for_dft` variable.

Hierarchical Core Wrapping

The hierarchical core wrapping feature can be used to build a core that contains only wrapper and core chains, but does not contain any test control module (TCM) or wrapper mode logic. This core can then be used at a higher level of integration with other wrapper chains or compression architectures. This hierarchical wrapping feature is only available in the maximized reuse flow.

Note:

This feature should not be confused with simply integrating a wrapped core in a hierarchical DFT flow, which is covered in [Integrating Wrapped Cores in Hierarchical Flows on page 505](#).

To enable hierarchical wrapping, use the following commands at the core level:

```
dc_shell> set_dft_configuration -wrapper enable
dc_shell> set_wrapper_configuration -class core_wrapper \
           -maximize_reuse enable -hier_wrapping enable
```

By default, hierarchical wrapping is disabled.

When hierarchical wrapping is enabled, a single mode, Internal_scan, is created. This mode creates both wrapper chains and core internal chains. In addition, the following core-level input ports and signals are created:

- Three separate shift signals are created to control input wrapper chains, output wrapper chains, and core internal chains.
- If safe states are specified, separate input and output safe control signals are created for input and output wrapper cells.
- In the maximized reuse flow, shared wrapper cells do not use a capture signal. However, if there are dedicated wrapper cells in the input and output wrapper chains, separate input and output capture-enable signals are created for input and output wrapper cells.

By default, the tool creates these signals using default port and signal names. To use existing placeholder ports for these signals instead, define them with the `set_dft_signal` command as follows:

```
dc_shell> set_dft_signal -view spec -type wrp_shift -port port_name
dc_shell> set_dft_signal -view spec -type wrp_ded_capture_in \
           -port port_name
dc_shell> set_dft_signal -view spec -type wrp_ded_capture_out \
           -port port_name
dc_shell> set_dft_signal -view spec -type wrp_safe_in -port port_name
dc_shell> set_dft_signal -view spec -type wrp_safe_out -port port_name
```

The test model generated after hierarchical wrapping includes attributes that identify input and output wrapper chains, input and output wrapper scan-enable signals, input- and output-dedicated safe control signals, and input- and output-dedicated wrapper capture-enable ports.

Note:

Multiple test modes are not allowed at the core level when using hierarchical wrapping to create the core.

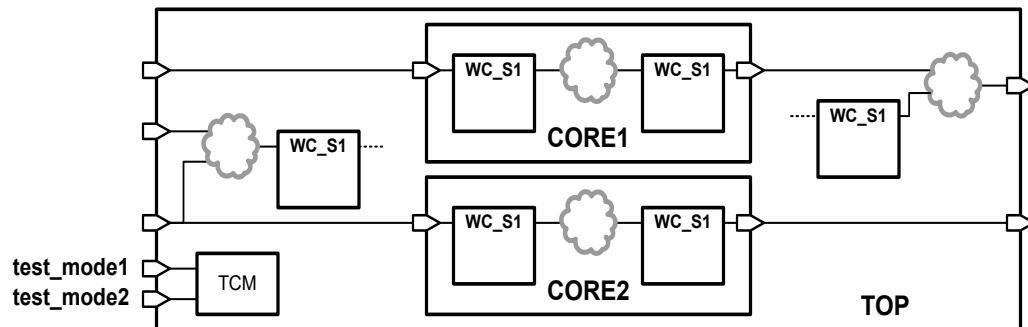
After the core is created, to enable integration of hierarchically wrapped cores at a higher level of the hierarchy, use the following commands:

```
dc_shell> set_dft_configuration -wrapper enable
dc_shell> set_wrapper_configuration -class core_wrapper \
           -maximize_reuse enable
```

During integration, the tool detects any hierarchical wrapped cores using the previously stored test attributes. Wrapper cells are incrementally added as needed to augment core-level wrapper capabilities, as shown in [Figure 222](#). The tool creates the normal set of wrapper test modes, selected by a test control module, according to user specifications.

As with typical wrapping flows, multiple test modes, including scan compression, are allowed at this level.

Figure 222 Integrating Wrapper-Only Cores at a Higher Level



Core-level input wrapper chains are mixed with integration-level input wrapper cells, and core-level output wrapper chains are mixed with integration-level output wrapper cells. Core-level wrapper shift, capture, and safe control signals are connected to appropriate integration-level wrapper logic.

Core-level wrapper chains can be integrated only one time. There is no support for recursive wrapping flows that propagate wrapper capabilities up through multiple hierarchy levels.

Limitations of the Maximized Reuse Flow

Note the following limitations related to the behavior of the scan-enable signal in capture mode:

- Preexisting scan-enable control of clock-gating cells

If the design has I/O registers controlled by clock-gating cells that are controlled by the scan-enable port, these I/O registers might not shift during the capture mode operation of the INTEST and EXTEST modes because scan-enable might not be active in capture mode.
- Preexisting scan-enable control of set and reset

If the set or reset pins of the I/O registers are controlled by the scan-enable port, the I/O registers might not shift during the capture mode of operation of the INTEST and EXTEST modes because scan-enable might not be active in capture mode.

Determining Power Domains for Dedicated Wrapper Cells

By default, dedicated wrapper cells are added to the top-level power domain. To have dedicated wrapper cells added to power domains other than the top-level power domain, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -add_wrapper_cells_to_power_domains enable
```

When this option is enabled, no new wrapper chain hierarchical block is created to enclose the dedicated wrapper cells. Wrapper cells are added according to the following rules:

- If the port is connected to an I/O register, the dedicated wrapper cell is added to the top-level power-domain hierarchy of the I/O register.
- If the port is connected to a CTL model, the top-level power-domain hierarchy of the instantiated CTL model is used as the location for the dedicated wrapper cell.
- If the port is connected to a gate, the top-level power-domain hierarchy of the gate is used as the location for the dedicated wrapper cell.
- If none of these conditions apply, the cell is added to the top-level hierarchy.

Using the `set_scan_path` Command With Wrapper Chains

You can use the `set_scan_path` command to control detailed aspects of wrapper chain construction. The following options are supported for wrapper chains:

```
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]
    [-scan_data_in si_port]
    [-scan_data_out ordered_port_list]
```

You must always specify the `-class wrapper` option when using `set_scan_path` to configure wrapper chains.

You can use the `-ordered_elements` option to control the ordering of wrapper cells in the wrapper chain. Provide an ordered list of ports, and DFT Compiler uses it to order the corresponding wrapper cells according to the specification:

```
dc_shell> set_scan_path -class wrapper WC1 \
    -ordered_elements {C B A Z Y}
```

By default, DFT Compiler can add wrapper cells to the beginning of the specified chain. To prevent this, specify the `-complete true` option. You cannot use multiple ordered list specifications to add more wrapper cells to an already specified wrapper chain because the last specification for a chain 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 specification takes precedence.

As described in [Wrapping Three-State and Bidirectional Ports on page 456](#), 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 Wchain0 -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 and output wrapper cell belong to different chains for a given port.

You can use the `-scan_data_in` and `-scan_data_out` options to specify the wrapper scan-in or wrapper scan-out signals for a chain:

```
dc_shell> set_dft_signal -type ScanInData -port wsi
dc_shell> set_dft_signal -type ScanDataOut -port wso
dc_shell> set_scan_path -class wrapper WC2 \
           -ordered_elements [list A B C] \
           -scan_data_in wsi -scan_data_out wso
```

The following commands implement separate input and output wrapper chains with separate wrapper shift signals using the `set_scan_path` command:

```
dc_shell> set_dft_signal -view spec \
           -type wrp_shift -port {wrp_ishift wrp_oshift}

dc_shell> set_scan_path -class wrapper WC_inputs \
           -input_wrapper_cells_only enable \
           -scan_enable wrp_ishift

dc_shell> set_scan_path -class wrapper WC_outputs \
           -output_wrapper_cells_only enable \
           -scan_enable wrp_oshift
```

A `set_scan_path` specification applied with the `-class wrapper` option and without the `-test_mode` option applies to all wrapper modes. You can also explicitly specify the

`-test_mode all` option. To apply a `set_scan_path` specification to a specific wrapper mode, you must predefine the wrapper mode before referencing it. For more information, see [Creating User-Defined Core Wrapping Test Modes on page 491](#).

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

Previewing the Wrapper Cells

After you have configured your wrapper chain and wrapper cells, use the `preview_dft` command with the `-test_wrappers all` option to preview the scan chain and wrapper chain characteristics. [Example 60](#) shows a wrapper chain preview report for a design.

Example 60 Preview Report of Shared I/O Registers

```
dc_shell> preview_dft -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
MY_core

Number of Wrapper Interface ports : 5

port type    port name
-----  -----
WRP_CLOCK    wrp_clock
WRP_CLOCK    ck1
WRP_CLOCK    ck2
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: 7

      Wrapper   Wrapper   Cntrl     Safe   Wrapper
Index  Port  Function  Cell Type  Cell  Impl  Value  Clock   Cell Name
-----  ----  -----  -----  -----  ---  ---  -----  -----
9      *       control  WC_S1_S   -     INP   0     ck1    BI1_2
8      bidi[1] inout    WC_S1    -     INP   -     ck1    BI1_3
7      bidi[1] tristate WC_S1    9     INP   -     ck1    BI1_1
6      A1      input    WC_S1    -     SWP   -     ck1    I1_reg
5      A2      input    WC_D1    -     SWP   -     wrp_clk A2_wrp0_8
4      A3      input    WC_D1    -     SWP   -     wrp_clk A3_wrp0_7
3      A4      input    WC_D1    -     SWP   -     wrp_clk A4_wrp0_6
```

```

2      Q1      output    WC_S1      -      SWP      -      ck1      Q1_reg
1      Q2      output    WC_S1      -      SWP      -      ck2      iQ2_reg
0      Q3      output    WC_S1      -      SWP      -      ck2      iQ3_reg

```

```

Number of ports not wrapped      : 3
MAINT_PORT
ck1
ck2

```

Input and output wrapper chains as well as core scan chains are shown in the preview_dft report. To see this information, run the following command:

```
dc_shell> preview_dft -show all -test_wrappers all
```

For all ports that have wrapper cells, the following information is reported:

- Index number of wrapper cell (used to indicate duplicate input/output wrapper cells and to reference control cells)
- Port name
- Wrapper cell type: dedicated or shared
- Function of wrapper cell: input, output, three-state, or control
- Control cell index number associated with a bidirectional or three-state wrapper cell
- Wrapper cell implementation: swapped-in (SWP) or in-place (INP)
- Safe value
- Wrapper cell clock
- Wrapper cell instance name

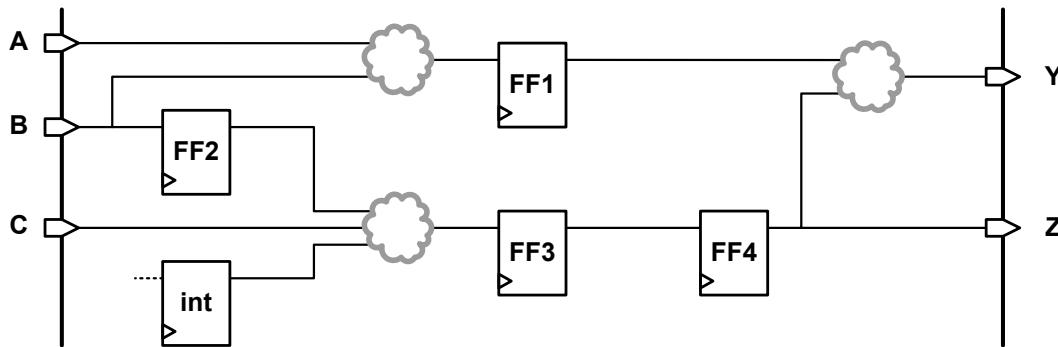
After DFT is inserted with the `insert_dft` command, you can report both normal scan chains and wrapper chains with the `report_scan_path` command. To report wrapper chains, use the `-test_mode` option of the `report_scan_path` command to provide the wrapper test-mode name. To list the available test modes, use the `list_test_modes` command.

Previewing Maximized Reuse Wrapper Cells

In the maximized reuse flow, the `preview_dft` command shows additional information about how register reuse is applied.

The preview report uses annotations to indicate when a shared wrapper cell is associated with multiple ports. [Figure 223](#) shows a design to be core-wrapped in the maximized reuse flow, and [Example 61](#) shows the corresponding wrapper chain report from the `preview_dft` command.

Figure 223 Design Example With Common Shared Wrapper Cells



Example 61 Wrapper Chain Preview Report with Common Shared Wrapper Cells

```
dc_shell> preview_dft -test_wrappers all
...

```

Index	Port	Wrapper Function	Wrapper Cell Type	Control Cell Impl	Safe Value	Wrapper Clock	Cell Name	
4	A	input	WC_S1	-	INP	-	CLK	FF1_reg
4	B	input	WC_S1		INP	-	CLK	FF1_reg(d)
4	Y	output	WC_S1		INP	-	CLK	FF1_reg(*)
3	B	input	WC_S1	-	INP	-	CLK	FF2_reg
3	C	output	WC_S1		INP	-	CLK	FF2_reg(*) (i)
2	C	input	WC_S1	-	INP	-	CLK	FF3_reg
1	C	output	WC_S1	-	INP	-	CLK	int_reg
0	Y	output	WC_S1	-	INP	-	CLK	FF4_reg
0	Z	output	WC_S1		INP	-	CLK	FF4_reg(d)

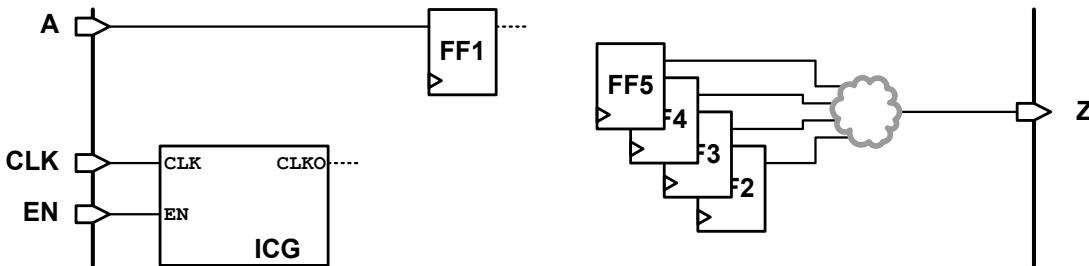
Note the following reporting conventions:

- Shared registers that are common to multiple input ports and multiple output ports are shown with the letter d (d) for all subsequent entries after the first register entry; these marked entries do not count against the total wrapped register count.
- Internal registers in the fanin to an input shared register are classified as output registers (because they drive values that are captured during outward-facing test).
- Shared registers that are common to both an input port and an output port but are used only as input wrapper cells are classified as input registers and shown with an asterisk (*) for the output port.
- Shared input registers that are in the fanin to another shared input register are classified as input registers and shown with the annotation (*) (i) for the fanin to the shared input register.

The preview report also indicates the reason why a dedicated wrapper cell is used at a port instead of a shared wrapper cell. Figure 224 shows a design to be core-wrapped in

the maximized reuse flow, and [Example 62](#) shows the corresponding wrapper chain report from the `preview_dft` command.

Figure 224 Design Example With Dedicated Wrapper Cells



Example 62 Wrapper Chain Preview Report With Dedicated Wrapper Cells

```
dc_shell> preview_dft -test_wrappers all
...
Note: Dedicated wrapper cells are grouped into a hierarchical instance named:
"top_Wrapper_inst" (Module name: "top_Wrapper_inst_design")

Dedicated Wrapper
Cell Reason          Description
-----
TEST-1183           Port drives enable pin of clock-gating cell
TEST-1185           Number of I/O registers exceeds reuse threshold

Wrapper Length:      3

          Wrapper      Wrapper
Index    Port   Function  Cell Type ...  Cell Name
-----  -----  -----
  2      A       input     WC_S1      FF1_reg
  1      EN      input     WC_D1      top_Wrapper_inst/top_EN_wrp0_1 (TEST-1
183)   0       output    WC_D1      top_Wrapper_inst/top_Z_wrp0_0 (TEST-11
85)
```

In the wrapper chain report, each port with a dedicated wrapper cell has a reason message code indicating why the dedicated wrapper cell is used. A short description for each reason is shown in a legend table before the report. For more information about a dedicated wrapper cell reason, see the man page. For a complete list of dedicated wrapper cell reasons, see [SolvNet article 038531, “What Are the Reasons for Dedicated Wrapper Cells to Be Inserted?”](#)

Post-DFT DRC Rule Checks

You can perform post-DFT DRC for the wrapper modes. To verify that the input wrapper chains shift in wrp_if (INTEST) mode, use the following commands:

```
dc_shell> current_test_mode wrp_if
```

```
dc_shell> dft_drc
```

To verify that the output wrapper chains shift in wrp_of (EXTEST) mode, run the following commands:

```
dc_shell> current_test_mode wrp_of
```

```
dc_shell> dft_drc
```

Creating User-Defined Core Wrapping Test Modes

Normally, when core wrapping is enabled, DFT Compiler creates a default set of core wrapping test modes as described in [Wrapper Test Modes on page 445](#). You can also create user-defined test modes for wrapped cores using the `define_test_mode` command.

Table 46 shows the test mode usage types you can specify with the `-usage` option of the `define_test_mode` command. You can define one or more test modes for each usage. When you define a test mode for a given usage, the default test mode is not created for that usage.

Table 46 Test Mode Usage Types for Wrapped Cores

Test mode usage	Test mode description	Default test mode name
wrp_if	Inward-facing uncompressed scan mode	wrp_if
scan_compression	Inward-facing compressed scan mode	ScanCompression_mode
wrp_of	Outward-facing uncompressed scan mode	wrp_of
wrp_safe	Safe mode	wrp_safe
scan	Unwrapped standard scan mode	Internal_scan ⁹

You can configure individual user-defined test modes using the `-test_mode` option of DFT configuration commands. The `-test_mode` option cannot reference a default test mode unless that mode name was explicitly defined with the `define_test_mode` command.

9. This default unwrapped test mode is not created when core wrapping is enabled; it can only be created for a wrapped core by defining a user-defined test mode.

Note that only the following options of the `set_wrapper_configuration` command can be used with the `-test_mode` option:

- `-max_length`
- `-chain_count`
- `-mix_cells`
- `-input_shift_enable`
- `-output_shift_enable`
- `-shift_enable`
- `-no_dedicated_wrapper_cells`

[Example 63](#) defines a set of user-defined core wrapping test modes. The inward-facing compressed scan mode, defined with `-usage scan_compression`, specifies the base mode as the inward-facing uncompressed scan mode, defined with `-usage wrp_if`.

Example 63 User-Defined Core Wrapping Test Modes

```
# define test modes
define_test_mode MY_INTEST      -usage wrp_if
define_test_mode MY_INTEST_COMP -usage scan_compression
define_test_mode MY_EXTEST      -usage wrp_of

# configure scan modes
set_wrapper_configuration -test_mode MY_INTEST \
    -class core_wrapper -chain_count 1 -mix_cells true
set_scan_configuration     -test_mode MY_INTEST \
    -chain_count 2 -clock_mixing mix_clocks \

set_wrapper_configuration -test_mode MY_EXTEST \
    -class core_wrapper -chain_count 2 -mix_cells false
set_scan_configuration     -test_mode MY_EXTEST \
    -chain_count 2 -clock_mixing mix_clocks

set_scan_compression_configuration \
    -test_mode MY_INTEST_COMP -base_mode MY_INTEST \
    -chain_count 12
```

Creating Compressed EXTEST Core Wrapping Test Modes

When you use DFTMAX Ultra streaming compression, you can compress both your inward-facing (INTEST) and outward-facing (EXTEST) test modes. This helps reduce scan I/O requirements when you have many wrapped cores instantiated at a higher level.

Figure 225 Uncompressed and Compressed Outward-Facing (EXTEST) Wrapper Modes

To do this, define a streaming compression mode that references an uncompressed outward-facing mode as its base mode. For example,

```
# enable DFT clients
set_dft_configuration -wrapper enable -streaming_compression enable

# define test modes
define_test_mode wrp_if -usage wrp_if
define_test_mode wrp_if_comp -usage streaming_compression
define_test_mode wrp_of -usage wrp_of
define_test_mode wrp_of_comp -usage streaming_compression

# configure inward-facing standard and compressed modes
set_scan_configuration \
    -test_mode wrp_if \
    -chain_count 2
set_streaming_compression_configuration \
    -test_mode wrp_if_comp -base_mode wrp_if \
    -chain_count 4

# configure outward-facing standard and compressed modes
set_wrapper_configuration -class core_wrapper \
    -test_mode wrp_of \
    -chain_count 2
set_streaming_compression_configuration \
    -test_mode wrp_of_comp -base_mode wrp_of \
    -chain_count 4 -inputs 1 -outputs 1
```

Wrapper chain count and length is configured as follows:

- For the compressed EXTEST mode, wrapper chains are always length-balanced using the configuration specified by the `set_streaming_compression_configuration` command. The `set_wrapper_configuration` command should not be applied to this mode.
- For the uncompressed EXTEST mode and all other wrapper modes, wrapper chains follow the rules described in [Controlling Wrapper Chain Count and Length on page 462](#).

Keep in mind that unlike uncompressed EXTEST chains, compressed EXTEST chains cannot be concatenated with other chains for rebalancing at higher levels. Just as with other compression modes, compressed EXTEST modes require dedicated codec scan I/O connections during core integration.

Alternatively, you can create many uncompressed wrapper chains at the core level, then concatenate or compress them along with other chains at a higher level in standard or compressed scan modes, respectively.

Note:

Because this feature is intended for few scan I/O pins, it requires DFTMAX Ultra streaming compression. Other compression technologies are not supported.

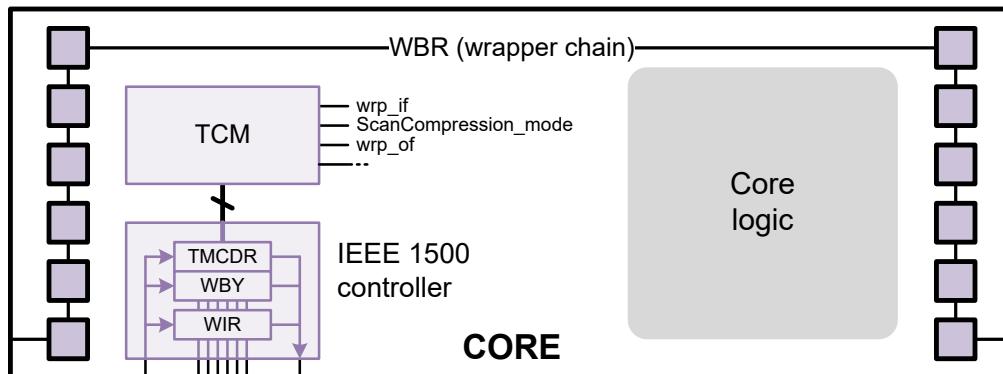
Creating an IEEE 1500 Wrapped Core

To create an IEEE 1500 wrapped core, simply enable both of the following features:

- Core wrapping (described in this chapter)
- Core-level IEEE 1500 test-mode control

[Figure 226](#) shows a core with these features enabled. In an IEEE 1500 core, the wrapper chain is also called the wrapper boundary register (WBR).

Figure 226 Wrapped Core With IEEE 1500 Controller



The WBR operation is controlled by the IEEE 1500 logic, but it uses regular scan-in and scan-out signals instead of the IEEE 1500 WSI and WSO scan signals. This enables the following features:

- The WBR can be implemented using separate input and output wrapper chains, which is required in the maximized reuse flow.
- The WBR can be compressed by scan compression codecs.

See Also

- [Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces on page 377](#) for information on implementing IEEE 1500 test-mode control

Wrapping Cores With OCC Controllers

When you create a core with a DFT-inserted or user-defined OCC controller, the clock chain provides control of the OCC-controlled clock.

When you also wrap such a core, the tool uses the following rules to determine how clock chains are incorporated into the core wrapping test modes that are created:

- Clock chains are always included in inward-facing (INTEST) modes. This allows the internal logic clocked by the OCC controller to be tested.
- Clock chains are included in outward-facing (EXTEST) modes only when the corresponding OCC-controlled clock clocks any shared or dedicated wrapper cell. This allows those wrapper cells to be controlled.

Note:

If you define a clock chain with the `set_scan_path -test_mode all` command, the clock chain is forced to be included in all test modes. To avoid this, define the clock chain only with the `set_scan_group` command, or apply the `set_scan_path` specification to only the desired test modes.

Use the `preview_dft -test_wrappers all` command to report the clock associated with each wrapper cell.

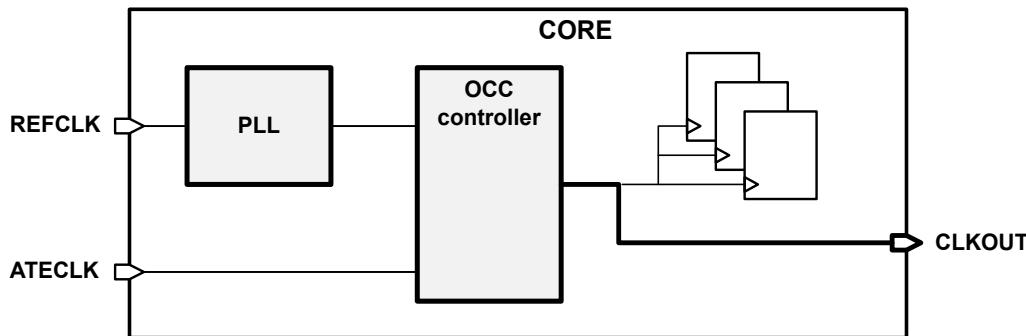
See Also

- [Chapter 12, On-Chip Clocking Support](#) for more information on OCC controllers and clock chains.

Wrapping Cores With OCC Clock Outputs

In some cases, you might have a core in which an OCC-controlled clock drives a core output port, as shown in [Figure 227](#).

Figure 227 OCC-Controlled Clock Driving a Core Output Port



In this case, do the following tasks during core creation:

- Enable advanced clock feedthrough analysis to help DRC include the clock output information into the core's CTL model:

```
dc_shell> set_app_var test_fast_feedthrough_analysis true
```

- Ensure that the clock chains are included in outward-facing (EXTEST) mode so that the top-level logic clocked by the OCC clock can be tested.

The presence of the OCC clock output is not a sufficient condition to do this. You must either clock at least one wrapper cell with the OCC clock, or you must manually define the clock chain with the `set_scan_path -test_mode all` command.

You do not need to disable wrapping on the clock output port. The tool recognizes the output port as a feedthrough port, as shown by the following message:

```
Information: No I/O registers are found for port 'CLKOUT'; not adding any dedicated wrapper cells to the port. (TEST-1180)
```

See Also

- [Using Advanced Clock Feedthrough Analysis on page 132](#) for more information on advanced clock feedthrough analysis.

Wrapping Cores With DFT Partitions

If you are using DFT partitions, the tool creates wrapper chains within each partition and assigns wrapper cells to partitions as follows:

- Shared wrapper cells are functional scan cells. They inherently belong to a DFT partition, as specified by the DFT partition configuration.
- Dedicated wrapper cells are associated with a port. The tool finds the first test cell (flip-flop or clock-gating cell) in the fanout of the port, then assigns the dedicated wrapper cell to the DFT partition for that test cell.

The following example shows per-partition wrapper chain configuration:

```
# enable core wrapping
set_dft_configuration -wrapper enable
set_wrapper_configuration -class core_wrapper -maximize_reuse enable

# define DFT partitions
define_dft_partition P1 -include ...
define_dft_partition P2 -include ...

# configure DFT partition P1
current_dft_partition P1
set_scan_configuration -chain_count 4
set_wrapper_configuration -class core_wrapper -chain_count 2

# configure DFT partition P2
current_dft_partition P2
set_scan_configuration -chain_count 4
set_wrapper_configuration -class core_wrapper -chain_count 2
```

Note the following limitation:

- When you use DFT partitions, separate wrapper chains are created within each partition. Wrapper chains cannot span across partitions.

Wrapping Cores With Multibit Registers

You can core-wrap designs that use multibit registers.

Multibit registers have multiple state elements (bits) that are individually addressable but share clock and scan signals. This reduces power consumption and routing congestion.

Both the RTL bus inference flow and placement-based register banking flows are supported. (However, if you use the `-input_map_file` or `-register_group_file` option of the `identify_register_banks` command in the placement-based flow, then the algorithms in this section are disabled.)

The Maximized Reuse Core Wrapper Flow

In the maximized reuse flow, multibit registers associated with ports can be shared wrapper cells. For best results, enable and configure maximized reuse core wrapping *before* performing multibit banking with

- The initial `compile_ultra` command (RTL bus inference flow)
- The `identify_register_banks` command (placement-based banking flow)

This informs the multibit banking algorithms that you will perform maximized reuse core wrapping during DFT insertion. The tool builds each multibit register from single-bit registers of the same type—input registers, output registers, or core registers—so they can be stitched into the corresponding wrapper or core scan chains.

To confirm this, the tool issues the following message during multibit banking:

Information: DFT core wrapping client enabled; banking anticipates core wrapping. (TEST-1291)

[Table 47](#) lists the wrapper configuration commands used by multibit banking.

Table 47 Wrapper Configuration Commands Used by Multibit Banking

Wrapper configuration command	Required?
<code>set_dft_configuration -wrapper enable</code>	Required
<code>set_wrapper_configuration -class core_wrapper \ -maximize_reuse enable</code>	Required
<code>set_wrapper_configuration -class core_wrapper \ -reuse_threshold value</code>	Optional
<code>set_wrapper_configuration -class core_wrapper \ -depth_threshold value</code>	Optional

Specifying the threshold values allows multibit banking to determine any ports that get dedicated wrapper cells by exceeding the thresholds. This removes the multibit banking register-type restrictions for registers associated with those ports.

Other core wrapper settings, such as port-specific wrapper cell specifications, are not considered by multibit banking. You do not need to define DFT signals, create a test protocol, or perform pre-DFT DRC.

If you forget to enable and configure core wrapping *before* performing multibit banking, the tool issues the following message during multibit banking:

Information: DFT core wrapping client disabled; banking anticipates no core wrapping. (TEST-1290)

To prevent multibit registers from being used as shared wrapper cells, which forces dedicated wrapper cells on any ports associated with them (as indicated by TEST-1067 warnings), set the following variable:

```
dc_shell> set_app_var test_soc_core_wrap_allow_multibit_ioregs false
```

The Simple Core Wrapper Flow

In the simple core wrapper flow, multibit registers cannot be wrapper cells. Ports associated with multibit registers always get a dedicated wrapper cell.

See Also

- [“Multibit Register Synthesis and Physical Implementation Application Note”](#) for detailed information on multibit cells and flows across multiple tools

Wrapping Cores With Synchronizer Registers

You can core-wrap designs that use synchronizer registers.

Synchronizer registers synchronize data communicated between asynchronous clock domains. A synchronizer register consists of two or more serially connected registers within the synchronizer cell.

The Simple Core Wrapper Flow

In the simple core wrapper flow, synchronizer registers cannot be shared wrapper cells. Ports associated with synchronizer registers always get a dedicated wrapper cell.

The Maximized Reuse Core Wrapper Flow

In the maximized reuse flow, you can reuse CTL-modeled synchronizer registers as shared wrapper cells by setting the following synchronizer length limit variable:

```
dc_shell> set_app_var test_core_wrap_sync_ctl_segment_length 2
```

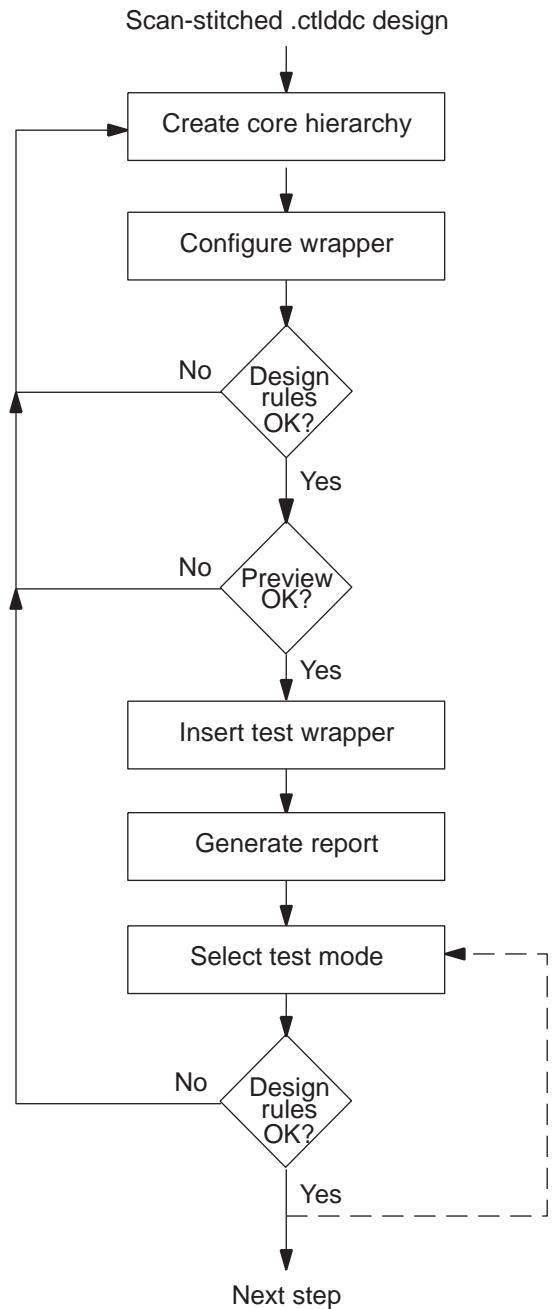
Ports associated with synchronizer registers longer than this value get a dedicated wrapper cell. The default is 0, which does not allow synchronizer registers to be reused as shared wrapper cells. For more information, see the man page.

For this feature, the synchronizer register must be a library cell that has a CTL model with a single test mode of type “InternalTestMode” with a single scan chain from scan data input to scan data output. Synchronizer registers modeled using Liberty library constructs are not supported.

Wrapping Cores With Existing Scan Chains

To wrap a core that is already scan-stitched, you should use the scan-stitched core flow. [Figure 228](#) illustrates this flow.

Figure 228 Scan-Stitched Core Flow Diagram



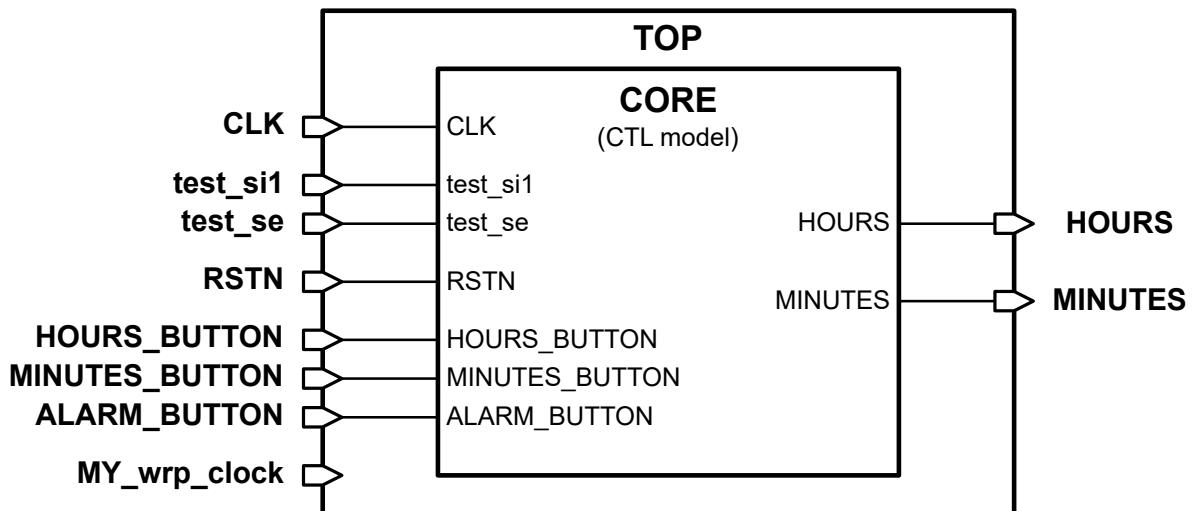
Start the flow with a CTL test model of the core. If you use a test model, DFT Compiler cannot touch the scan logic in the model during wrapping.

1. Create an enclosing top-level hierarchy that instantiates the core.

[Figure 229](#) 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. You can also include unconnected placeholder ports for wrapper signals in the top-level design.

You can use a text editor or an automated script to create a netlist file that accomplishes this step.

Figure 229 Creating Core Hierarchy



2. Configure the core wrapper.

Use the `set_dft_configuration -wrapper enable` command to enable core wrapping. If you do not issue this command, the other core wrapping commands will not have any effect. To set the 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 -maximize_reuse true
  
```

3. Define the wrapper signals and configure the wrapper chains.

By default, control signals are added to the design to control the wrapper configuration. To specify existing placeholder ports for these signals, use the `set_dft_signal`

command to specify the names and connections of these signals. These wrapper signals must exist in the top-level design you previously created.

```
dc_shell> set_dft_signal -view spec \
    -type wrp_clock -port MY_wrp_clock
```

4. Configure the wrapper chain characteristics.

You can use the `set_wrapper_configuration` and `set_boundary_cell` commands to specify any wrapper chain characteristics. You can use the `set_scan_path` command to specify the order of the wrapper cells.

```
dc_shell> set_wrapper_configuration -class core_wrapper \
    -chain_count 2

dc_shell> set_boundary_cell -class core_wrapper \
    -ports {CLKOUT} -type none

dc_shell> set_scan_path \
    -class wrapper WC0 \
    -ordered_elements [list ordered_port_list]
```

5. Check test design rules.

Use the `dft_drc` command to check test design rules.

```
dc_shell> create_test_protocol
dc_shell> dft_drc
```

6. 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_wrappers all
```

7. 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
```

8. (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
```

9. 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.

Creating an EXTEST-Only Core Netlist

During core creation, you can create and write out an additional version of the core netlist that contains only the logic needed for operation in outward-facing (EXTEST) test modes. This EXTEST-only core netlist significantly reduces the memory requirements for pattern generation of top-level test modes in TestMAX ATPG and for pattern simulation.

After inserting DFT and writing the full netlist files, protocol files, and other design files, execute the following command:

```
dc_shell> create_dft_netlist -type extest
```

This command removes all logic in the design except for the following:

- Wrapper chains
- Interface logic between wrapper chains and I/O ports
- Wrapper chain control logic
- Test-mode decode logic
- IEEE 1500 controller logic
- Any other logic required for EXTEST mode operation

Note:

A DFTMAX or TestMAX DFT license is required to create an EXTEST-only core netlist.

After removing the unnecessary logic with the `create_dft_netlist` command, write out an EXTEST-only Verilog netlist file using the `write -format verilog` command. Because the `create_dft_netlist` command removes logic from the design in memory, these should be the last steps in your core creation script.

You can use this EXTEST-only core netlist in any TestMAX ATPG pattern generation run where the core operates in its outward-facing test mode. All of the necessary logic is retained so that the core operates properly when exercised by the test protocols.

Note the following limitations:

- The `create_dft_netlist` command can only be run after DFT is inserted, or after a DFT-inserted design is read from a .ddc file. Existing-scan inference flows are not supported.
- The `create_dft_netlist` command uses attributes set by the `insert_dft` command. Any structural design modifications made after DFT insertion are not considered during netlist processing.
- Cores with multiple EXTEST test modes are not supported.

See Also

- [SolvNet article 2686021, “How To Preserve Special Logic When Creating an EXTEST Netlist”](#) for details on using the `dont_touch` attribute to preserve special logic

Integrating Wrapped Cores in Hierarchical Flows

The following topics describe how wrapped cores can be integrated:

- [Scheduling Wrapped Cores](#)
- [Integrating Wrapped Cores in a Compressed Scan Flow](#)
- [Nested Integration of Wrapped Cores](#)
- [Mixing Wrapped and Unwrapped Cores](#)
- [Top-Down Flat Testing With Transparent Wrapped Cores](#)

Scheduling Wrapped Cores

In hierarchical flows, wrapped cores have inward-facing and outward-facing test modes that must be incorporated into top-level test modes during core integration. To specify this test-mode mapping for wrapped cores, use the `-target` option of the `define_test_mode` command at the top level.

The `-target` option specifies a list of core and test-mode pairs to use for the top-level test mode being defined; each pair consists of a core instance name and a core test-mode name separated by a colon (:). In compressed scan flows, the list can also contain the name of the current design to specify that the top-level logic should be active and tested.

When you use the `-target` option in a flow that integrates wrapped cores, the following rules apply:

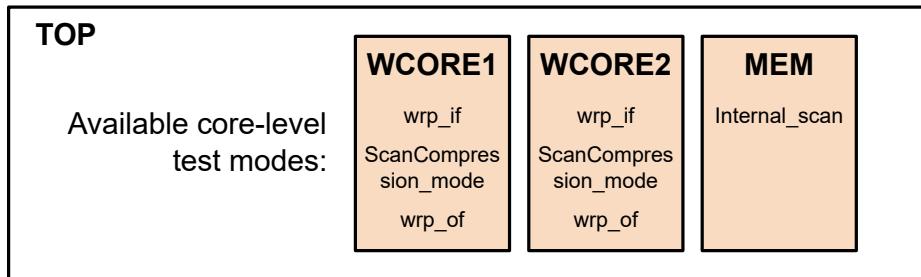
- All test modes must be defined with the `define_test_mode` command; no test modes are automatically created.
- All test mode definitions must use the `-target` option.
- Targeted cores (included in the target list) are placed in their targeted mode.
- If a core is targeted in some core-testing modes but not others, it is not tested in the test modes where it is not targeted. This is known as *sparse targeting*. (To completely exclude a core from all top-level test modes, use the `-exclude_elements` option of the `set_scan_configuration` command.)
 - Untested wrapped cores are placed in SAFE mode, if available. Otherwise, they are placed in mission mode.
- Untargeted cores (not included in any target list) are tested in top-level modes where the top-level logic is tested:
 - In top-level standard scan modes, they are placed in standard scan mode.
 - In top-level compressed scan modes, they are placed in compressed scan mode (for compressed scan cores) or standard scan mode (for standard scan cores).
 - They are placed in the first available such mode defined inside the core's test model.
- The top-level logic, which is all scannable logic outside DFT cores, is only active and tested when targeted.
- Untested wrapped cores with a default-named `wrp_of` test mode do not need to be explicitly scheduled; they are automatically placed in that mode when the top-level logic is scheduled. Outward-facing test modes with nondefault names must be explicitly scheduled.

Note:

The `-target` option has some limitations when used in compressed scan core integration modes. See [HASS and Hybrid Flow Limitations on page 733](#).

[Figure 230](#) shows an example with three DFT cores instantiated in a top-level design. The example includes two wrapped cores with inward-facing and outward-facing test modes and a scannable memory with a single `Internal_scan` test mode.

Figure 230 Three Cores With Different Test Modes Instantiated in a Top-Level Design



The combination of the core-wrapping feature and the `-target` option provide a great deal of flexibility in testing the design—one core at a time, all cores together, in groups, top-level logic only with no cores active, and so on. [Example 64](#) shows commands that define a schedule for the wrapped core example.

Example 64 Specifying User-Defined Test Mode Scheduling

```
# test cores one at a time in standard scan mode
define_test_mode STD_ONLY1 -usage scan -target {Wcore1:wrp_if}
define_test_mode STD_ONLY2 -usage scan -target {Wcore2:wrp_if}

# test both cores together in compressed scan mode
define_test_mode COMP_1AND2 -usage scan_compression \
    -target {Wcore1:ScanCompression_mode Wcore2:ScanCompression_mode}

# test top-level logic in standard and compressed scan modes
define_test_mode STD_ONLYTOP -usage scan -target {top}
define_test_mode COMP_ONLYTOP -usage scan_compression -target {top}
```

For the previous example, [Figure 231](#) shows the top-level test modes created by the tool during core integration. Each column represents a core, each row represents a top-level test mode, and the intersections of the columns and rows show the core-level test mode used for that top-level test mode. Blue columns indicate logic scheduled by the `-target` option.

Figure 231 Top-Level Test Modes With User-Defined Test-Mode Scheduling

Available core-level test modes: Top-level test mode definitions:	WCORE1	WCORE2	top	MEM
<pre>define_test_mode STD_ONLY1 -usage scan -target {WCORE1:wrp_if}</pre>	wrp_if ScanCompression_mode wrp_of	wrp_if		Internal_scan
<pre>define_test_mode STD_ONLY2 -usage scan -target {WCORE2:wrp_if}</pre>		wrp_if		Internal_scan
<pre>define_test_mode COMP_1AND2 -usage scan_compression -target {WCORE1:ScanCompression_mode WCORE2:ScanCompression_mode}</pre>	ScanCompression_mode	ScanCompression_mode		Internal_scan
<pre>define_test_mode STD_ONLYTOP -usage scan -target {top}</pre>	wrp_of	wrp_of	(Top-level logic tested)	Internal_scan
<pre>define_test_mode COMP_ONLYTOP -usage scan_compression -target {top}</pre>	wrp_of	wrp_of	(Top-level logic tested)	Internal_scan

Wrapped cores are not usually scheduled in inward-facing modes in test modes that test top-level logic because the active wrapper chains prevent the cores from capturing values from the top-level logic. If this configuration is detected, the tool issues a warning:

Warning: Inward-facing cores are tested along with logic outside those cores in test mode 'COMP_1AND2_WITH_TOP'. (TEST-2077)

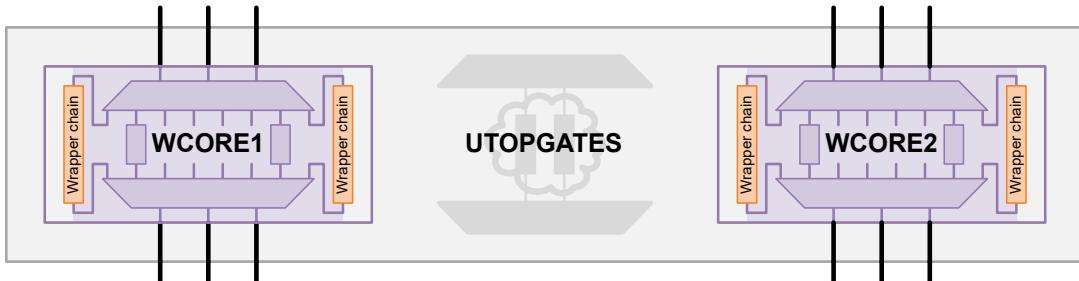
However, to test wrapped cores along with top-level logic, they can be placed in transparent modes, as described in [Top-Down Flat Testing With Transparent Wrapped Cores on page 511](#).

You do not need to set the `-wrapper` option of the `set_dft_configuration` command to `enable` when integrating wrapped cores; this option is only needed when creating wrapped cores.

Integrating Wrapped Cores in a Compressed Scan Flow

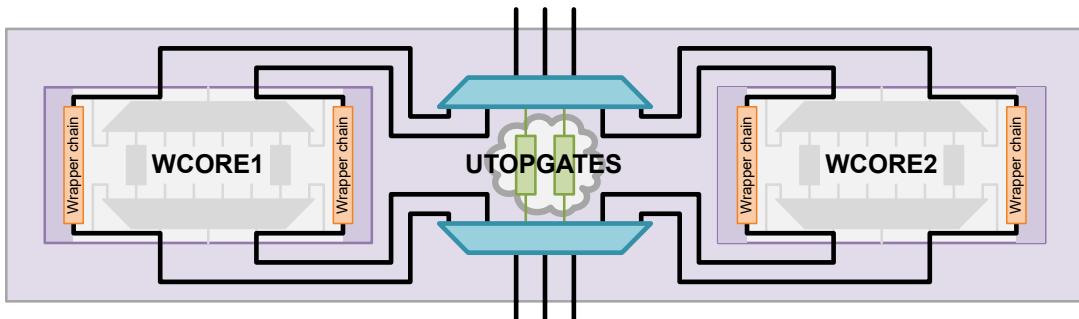
In test modes that test wrapped cores, wrapped cores are placed in the inward-facing (INTEST) modes specified by the test mode schedule definition. For wrapped cores with scan compression, the codecs inside the core are used, as shown in [Figure 232](#). In this case, compressed scan cores act as cores to be integrated.

Figure 232 Testing Two Wrapped Inward-Facing Compressed Scan Cores



In test modes that test the top-level logic, wrapped cores are placed in their outward-facing (EXTEST) modes, which allows the tests to control and observe signals at the core boundaries. In this case, the outward-facing wrapper chains become scan segments at the top level, which can be concatenated and compressed as with any other scan segment. You can use the Hybrid integration flow to compress these wrapper chains with a top-level codec, as shown in [Figure 233](#).

Figure 233 Testing the Top-Level Logic With Outward-Facing Wrapped Cores



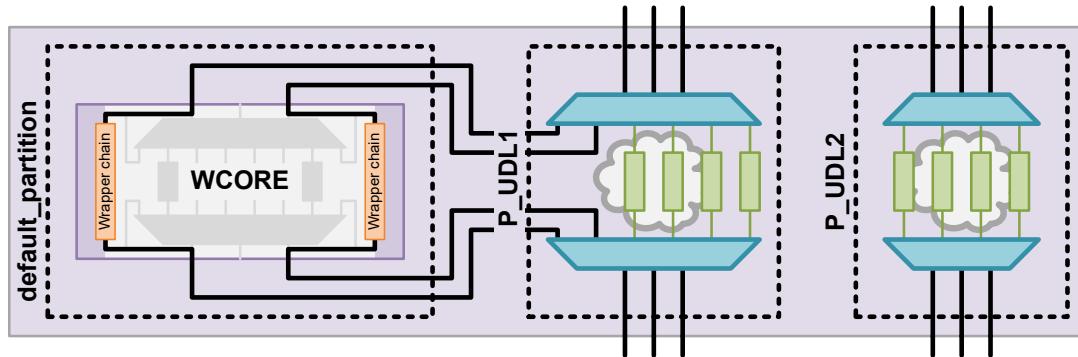
Because the outward-facing chains of wrapped cores are incorporated into the top-level scheduled modes, you should ensure that the number and lengths of the wrapper chains allow them to be effectively length-balanced in the top-level scan chains.

In flows with DFT partitions, you can reassign the outward-facing wrapper chains of cores to another partition by specifying those cores with the `-extest_cells` option of the `define_dft_partition` command. This allows you to compress core wrapper chains with specific top-level codecs. For example,

```
# reassign the outward-facing wrapper chains of the wrapped core WCORE
# to partition P_UDL1
define_dft_partition P_UDL1 -include {UDL1} -extest_cells {WCORE}
define_dft_partition P_UDL2 -include {UDL2}
```

[Figure 234](#) shows the scan compression logic created by these commands.

Figure 234 Reassigning Outward-Facing Chains to a Different DFT Partition

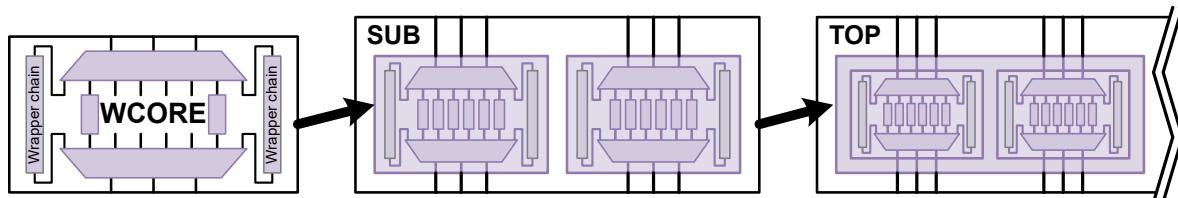


The `-extest_cells` option affects all test modes that use the specified cores in outward-facing modes, including uncompressed scan modes. You can reassign multiple cores to a single partition or you can distribute the cores across the partitions. Test modes that use the cores in inward-facing modes are unaffected.

Nested Integration of Wrapped Cores

You can perform multiple levels of integration for wrapped cores, as shown in [Figure 235](#).

Figure 235 Nested Integration of Wrapped Cores



Core wrapping information is passed upward through each subsequent level of integration, so that each higher integration level effectively also becomes a wrapped core. Additional scan logic, such as unwrapped cores or glue logic, can exist at any level.

Nested integration of wrapped cores is supported. However, nested core wrapping, which is enabling core wrapping when a wrapped core already exists in the design hierarchy, is not supported.

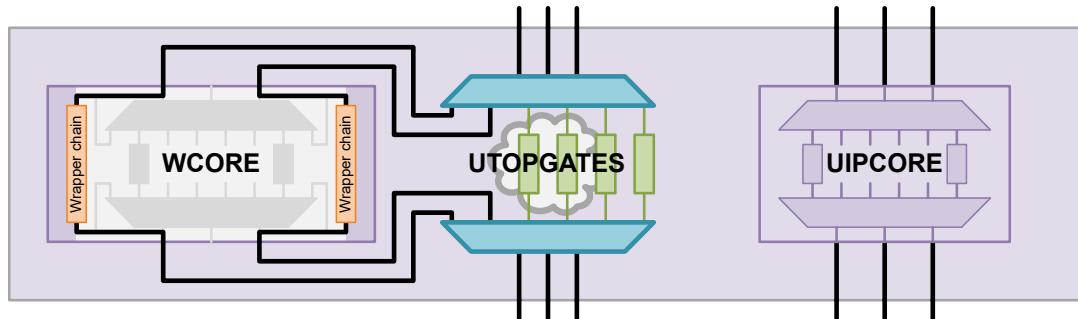
Mixing Wrapped and Unwrapped Cores

If you mix wrapped and unwrapped cores, the unwrapped cores are tested along with the top-level logic. Unwrapped cores are integrated according to the usual rules:

- For unwrapped compressed scan cores, the core-level connections are promoted to top-level connections.
- For unwrapped standard scan cores, the core-level scan segments are incorporated into top-level scan chains. In the Hybrid mode, they are compressed by the top-level codec.

[Figure 236](#) shows an unwrapped, compressed scan core that is tested along with the top-level logic in a compressed scan flow.

Figure 236 Testing an Unwrapped Compressed Scan Core Along With the Top-Level Logic



Only DFT-inserted cores, which have CTL model information, can be included in top-level test-mode schedule definitions. If you have hierarchical blocks at the top level that are not DFT-inserted, do not include them in the top-level test-mode schedule definition; they have no core-level test modes to schedule.

Top-Down Flat Testing With Transparent Wrapped Cores

In some cases (such as for IDDQ testing), you might want to make core wrapper chains transparent to perform top-down flat testing of the full chip. This can be done by implementing and using a *transparent mode* in your wrapped cores.

Transparent modes are described in the following topics:

- [Introduction to Transparent Test Modes](#)
- [Defining Core-Level Transparent Test Modes](#)
- [Defining Top-Level Flat Test Modes](#)
- [Limitations](#)

Introduction to Transparent Test Modes

Wrapped cores allow a design to be tested hierarchically. Inward-facing test modes test the logic inside a core, and outward-facing test modes test the logic surrounding a core. In both types of modes, the wrapper chain is actively used to logically separate the core logic from the surrounding logic.

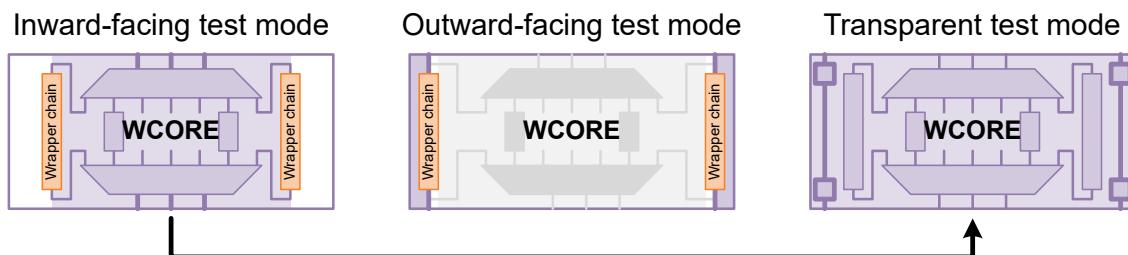
To perform top-down flat testing of a design with wrapped cores, wrapped cores must provide an additional test mode where the wrapper chain is logically transparent. This core-level test mode is called a *transparent* test mode. At the top level, wrapped cores can be placed into their transparent modes to perform top-down flat testing of the entire design. This top-level test mode is called a *flat* test mode.

At the core level, a transparent test mode is defined as an extension of an inward-facing standard scan or compressed scan test mode. A transparent mode is identical to its referenced inward-facing mode, except that

- Wrapper chains are treated as regular scan chains.
- Dedicated wrapper cells are logically transparent, although their flip-flops remain in the wrapper chains and act as observe test points on I/O paths.
- Any scan compression codecs from the referenced inward-facing mode are reused.
- Feedthrough chains drive the outward-facing wrapper scan I/Os in transparent mode.

[Figure 237](#) shows how the transparent mode of a wrapped core relates to the inward-facing and outward-facing test modes. Standard scan modes are not shown.

Figure 237 The Three Test Mode Types of a Wrapped Core



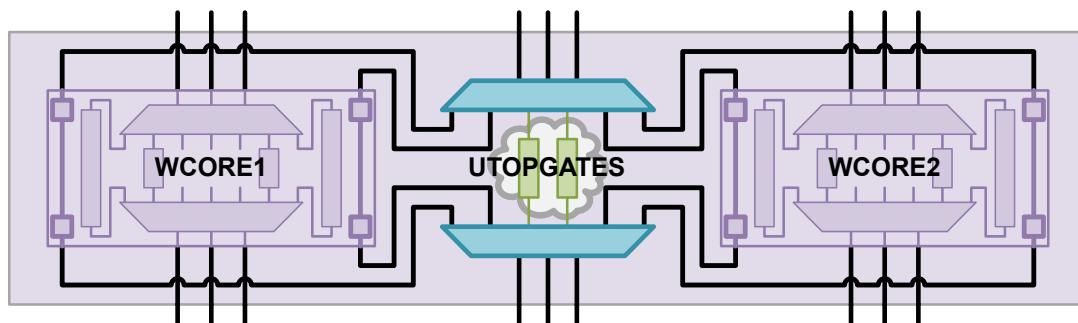
Transparent modes require that dedicated wrapper scan I/Os be used in the core's outward-facing modes. The tool creates and stitches placeholder chains, called *feedthrough* chains, between these outward-facing I/Os in the transparent modes. Each feedthrough chain consists of two scan cells that are clocked by the same head and tail clock and edge as the outward-facing wrapper chains they represent.

At the top level, a flat test mode is defined as an extension of a top-level-only mode, which tests only top-level logic with wrapped cores in outward-facing mode. A flat mode is identical to its underlying top-level-only mode, except that:

- You schedule wrapped cores in transparent mode instead of outward-facing mode.
- Any top-level scan compression codecs from the underlying top-level-only mode are reused.

[Figure 238](#) shows how wrapped cores, placed in their transparent modes, are tested along with the top-level logic in a top-level flat test mode.

Figure 238 Performing Top-Down Flat Testing Using Wrapped Cores in Transparent Mode



The feedthrough chains allow the codec from the top-level-only mode to be reused in the flat test mode. Feedthrough chains are used only in flat test modes where they are compressed by a top-level codec reused from the underlying top-level-only mode.

Feedthrough chains are unused in

- Top-level flat standard scan modes
- Top-level flat compressed scan modes without top-level codecs that compress the core wrapper connections, such as the DFTMAX HASS integration flow
- Flat modes where the top-level codec participates in codec I/O sharing

See [Limitations on page 515](#).

Defining Core-Level Transparent Test Modes

To define a transparent test mode at the core level, use the `-transparent_mode_of` option of the `define_test_mode` command to specify a previously defined inward-facing standard scan or compressed scan test mode as the parent mode. The transparent mode is derived from this parent mode. For example,

```
# define inward-facing modes
define_test_mode INWARD_STD -usage wrp_if
define_test_mode INWARD_COMP -usage scan_compression
```

```
# define outward-facing mode
define_test_mode OUTWARD_STD -usage wrp_of

# define transparent modes
define_test_mode TRANS_STD -usage wrp_if \
    -transparent_mode_of INWARD_STD
define_test_mode TRANS_COMP -usage scan_compression \
    -transparent_mode_of INWARD_COMP
```

Specify the usage of a transparent mode to be the same as its parent mode: `wrp_if` for a standard scan mode and `scan_compression` for a compressed scan mode.

To define dedicated wrapper scan I/Os for the core's outward-facing modes, use the `-test_mode` option of the `set_dft_signal` command. For example,

```
# define dedicated scan I/Os for outward-facing mode
set_dft_signal -view spec -type ScanDataIn -port SI[*] \
    -test_mode {INWARD_STD TRANS_STD INWARD_COMP TRANS_COMP}
set_dft_signal -view spec -type ScanDataOut -port SO[*] \
    -test_mode {INWARD_STD TRANS_STD INWARD_COMP TRANS_COMP}

set_dft_signal -view spec -type ScanDataIn -port WSI[*] \
    -test_mode {OUTWARD_STD}
set_dft_signal -view spec -type ScanDataOut -port WSO[*] \
    -test_mode {OUTWARD_STD}
```

A transparent mode inherits all DFT configuration information from its parent mode. Do not apply any other DFT configuration commands specifically to the transparent test modes; they are ignored. Do not specify a base mode for transparent compressed scan modes.

As a reminder, the `define_test_mode -transparent_mode_of`, `preview_dft`, and `insert_dft` commands print the following message:

Information: Transparent modes inherit their DFT configuration from their parent modes; any other DFT specifications applied to transparent modes are ignored. (TEST-2082)

For more information, see the man page for this message.

Defining Top-Level Flat Test Modes

To define a flat test mode at the top level, define a test mode that schedules the cores in their transparent mode along with the top-level logic of the current design. For example,

```
# define modes that test only cores
define_test_mode CORES_INWARD_STD -usage scan \
    -target {core1:INWARD_STD core2:INWARD_STD}
define_test_mode CORES_INWARD_COMP -usage scan_compression \
    -target {core1:INWARD_COMP core2:INWARD_COMP}

# define modes that test only top-level logic
define_test_mode TOPONLY_STD -usage scan \
    -target {top}
```

```
define_test_mode TOPONLY_COMP -usage scan_compression \
    -target {top}

# define flat test modes that test the entire design
define_test_mode TOPFLAT_STD -usage scan \
    -target {core1:TRANS_STD core2:TRANS_STD top}
define_test_mode TOPFLAT_COMP -usage scan_compression \
    -target {core1:TRANS_COMP core2:TRANS_COMP top}
```

The tool automatically identifies the standard scan or compressed scan top-level-only mode associated with each flat mode.

Do not apply any DFT configuration commands specifically to the flat test modes; they are ignored.

Limitations

Note the following limitations of top-down flat testing using transparent core-level test modes:

- DFTMAX Ultra compression is not supported.
- You must define core-level transparent and top-level flat test modes with the `define_test_mode` command; they are not created by default.
- At the core level, to create transparent modes, you must use dedicated wrapper scan I/Os in the outward-facing modes.
- At the top level, you cannot mix transparent and nontransparent core test modes in the same test mode.
- At the top level, you cannot define multiple flat top-level scan compression modes.
- At the top level, codecs with shared I/O connections cannot be reused by a top-level flat mode. In this case, the tool inserts a new codec for the flat mode. This new codec does not compress any feedthrough chains.

SCANDEF Generation for Wrapper Chains

SCANDEF generation is supported for wrapper chains. The SCANDEF information is generated as follows:

- Wrapper cells can be reordered within wrapper chains.
- Wrapper chain cells cannot be repartitioned with regular scan chain cells.
- Input wrapper cells can be repartitioned between input wrapper chains, and output wrapper cells can be repartitioned between output wrapper chains.

- If input and output wrapper cell mixing is enabled, input wrapper chain cells can be repartitioned with output wrapper chain cells.

Input and output wrapper chain mixing is enabled by default in the simple wrapper flow and disabled by default in the maximized reuse flow. You can change this setting with the `-mix_cells` option of the `set_wrapper_configuration` command.

- Each wrapper cell is represented as an ORDERED construct that ensures all logic gates for that wrapper cell are kept together. However, a shared wrapper cell in the maximized reuse flow is simply a scan cell, so it is represented as an individual scan cell instead of an ORDERED construct.

See Also

- [Using The SCANDEF-Based Reordering Flow on page 635](#) for more information about generating SCANDEF information

Core Wrapping Scripts

The following script examples illustrate core wrapping.

Core Wrapping With Dedicated Wrapper Cells

[Example 65](#) uses the simple wrapping flow to wrap a core with dedicated wrapper cells at all ports.

Example 65 Script Example 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

# 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 -verbose

# Report the configuration of the wrapper utility, optional
report_wrapper_configuration

# Preview all test structures to be inserted
preview_dft -show all -test_wrappers all
report_dft_configuration

# Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode wrp_of
report_scan_path -view existing_dft -cell all \
    > reports/wrap_dedicated_wrp_of.rpt

current_test_mode wrp_if
report_scan_path -view existing_dft -cell all \
    > reports/wrap_dedicated_wrp_if.rpt

report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchy -output ddc/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_protocol -test_mode wrp_if -output stil/wrp_if.spf
write_test_protocol -test_mode wrp_of -output stil/wrp_of.spf

```

Core Wrapping With Maximized Reuse

Example 66 wraps a core with maximized reuse enabled.

Example 66 Script Example for Maximized Reuse 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

# Enable and configure wrapper client
set_dft_configuration -wrapper enable

# Configure for maximized reuse wrappers, using existing cells
set_wrapper_configuration -class core_wrapper \

```

```

-maximize_reuse enable \
-reuse_threshold 4 \
-style shared \
-register_io_implementation in_place \
-mix_cells false \
-use_system_clock_for_dedicated_wrp_cells enable \
-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 -verbose

# Report the configuration of the wrapper utility, optional
report_wrapper_configuration

# Preview all test structures to be inserted
preview_dft -show all -test_wrappers all
report_dft_configuration

# Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode wrp_of
report_scan_path -view existing_dft -cell all \
> reports/wrap_dedicated_wrp_of.rpt

current_test_mode wrp_if
report_scan_path -view existing_dft -cell all \
> reports/wrap_dedicated_wrp_if.rpt

report_dft_signal -view existing_dft -port *

report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchy -output ddc/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_protocol -test_mode wrp_if -output stil/wrp_if.spf
write_test_protocol -test_mode wrp_of -output stil/wrp_of.spf

```

12

On-Chip Clocking Support

On-Chip Clocking (OCC) support is common to all scan ATPG (Basic-Scan and Fast-Sequential) and compressed 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 DFT-inserted OCC clock controller. 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 clock chain control bits.

This chapter includes the following topics:

- [Background](#)
- [Supported DFT Flows](#)
- [Clock Type Definitions](#)
- [Capabilities](#)
- [OCC Controller Structure and Operation](#)
- [Enabling On-Chip Clocking Support](#)
- [Specifying OCC Controllers](#)
- [Reporting Clock Controller Information](#)
- [DRC Support](#)
- [DFT-Inserted OCC Controller Configurations](#)
- [Waveform and Capture Cycle Example](#)
- [Limitations](#)

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.

When using this approach, additional on-chip controller circuitry is inserted 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. The DFT Compiler and TestMAX ATPG tools support a comprehensive set of features to ensure that

- The test-mode control logic for the OCC controller operates correctly and has been connected properly
- Test-mode clocks from the OCC circuitry can be efficiently used by TestMAX ATPG for at-speed test generation
- OCC circuitry can operate asynchronously to other shift clocks from the tester
- TestMAX ATPG patterns do not require additional modifications to use the OCC and to run properly on the tester

Supported DFT Flows

On-chip clocking (OCC) is supported in the following DFT flows:

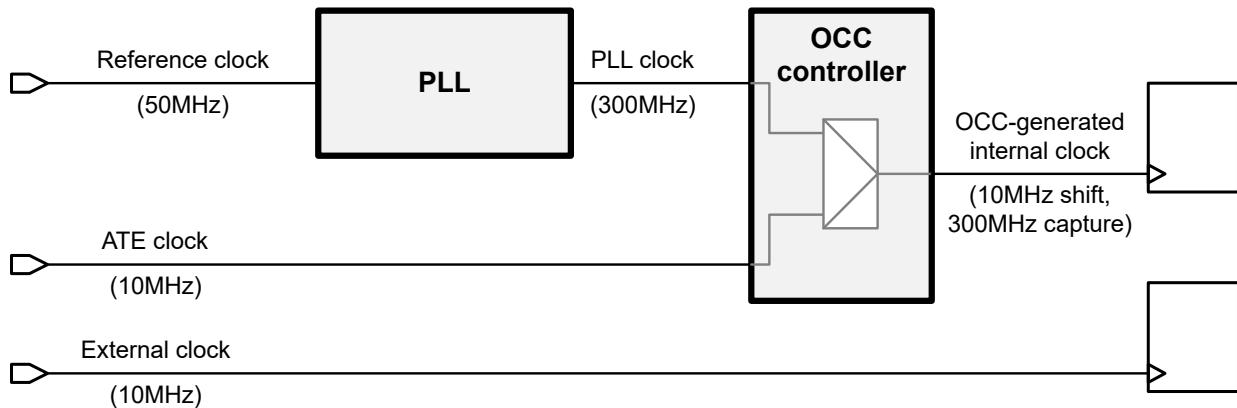
- Basic scan flow
 - Top-down, nonhierarchical compressed scan insertion flow
 - Bottom-up basic scan flow with OCC controller stitching at the top level
 - Bottom-up hierarchical adaptive scan synthesis flow, which represents subblocks with test models that are OCC controller stitched at the top level
 - Hierarchical adaptive scan synthesis (HASS) and Hybrid flows, which stitch the cores at the top level with integration at the top level
-

Clock Type Definitions

Note the following clock definitions as they apply to OCC controller clocks in this chapter. [Figure 239](#) shows an example of each clock type.

- *Reference clock* – The frequency reference to the phase-locked loop (PLL). It must be maintained as a constantly pulsing and free-running oscillator, or the circuitry will lose synchronization.
- *PLL clock* – The output of the PLL. It is also a free-running source that runs at a constant frequency that might or might not be the same as the reference clock.
- *ATE clock* – 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. The period for this clock is determined by the `test_default_period` variable. Usually the ATE clock is not used as a reference clock, but it must be treated as a free-running oscillator so that it does not capture predictable data while the OCC controller generates at-speed clock pulses. The ATE clock is called a dual clock signal when the same port drives both the ATE clock and the reference clock.
- *Internal clock* – The OCC controller is responsible for gating and selecting between the PLL and ATE clocks, thus creating the internal clock signal to satisfy ATPG requirements.
- *External clock* – A primary clock input of a design that directly clocks flip-flops through the combinational logic, without the use of a PLL clock. The period for this clock is determined by the `test_default_period` variable.

Figure 239 Clock Types Used in the OCC Controller Flow



Capabilities

The following OCC features are available:

- Synthesis of individual or multiple clock controllers and clock chains, using the DFT-inserted OCC controller
- 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 the TestMAX ATPG tool
- Support of post-DFT DRC, scan chain shifting, and scan compression
- Support of user-defined clock controller logic and clock chains that are already instantiated in the design

OCC Controller Structure and Operation

OCC controller types and operation is covered in the following topics:

- [DFT-Inserted and User-Defined OCC Controllers](#)
 - [Synchronous and Asynchronous OCC Controllers](#)
 - [OCC Controller Signal Operation](#)
 - [Clock Chain Operation](#)
 - [Logic Representation of an OCC Controller and Clock Chain](#)
 - [Scan-Enable Signal Requirements for OCC Controller Operation](#)
-

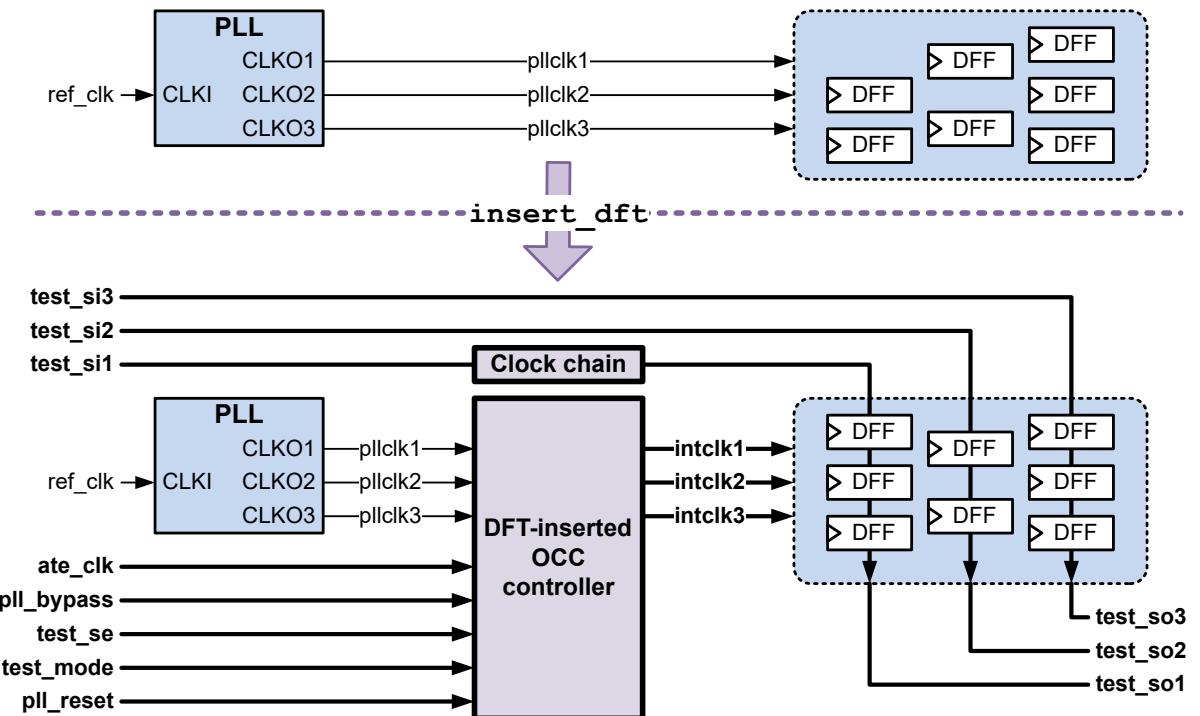
DFT-Inserted and User-Defined OCC Controllers

You can use DFT-inserted or user-defined OCC controllers, as described in the following flows:

- [Specifying DFT-Inserted OCC Controllers](#)

The `insert_dft` command performs insertion and synthesis of a DFT-inserted OCC controller and clock chain, making control signal connections and modifying the clock signal connections as needed. The OCC controller design is validated and incorporated into the resulting test protocol. This flow is shown in [Figure 240](#).

Figure 240 OCC and Clock Chain Synthesis Insertion Flow

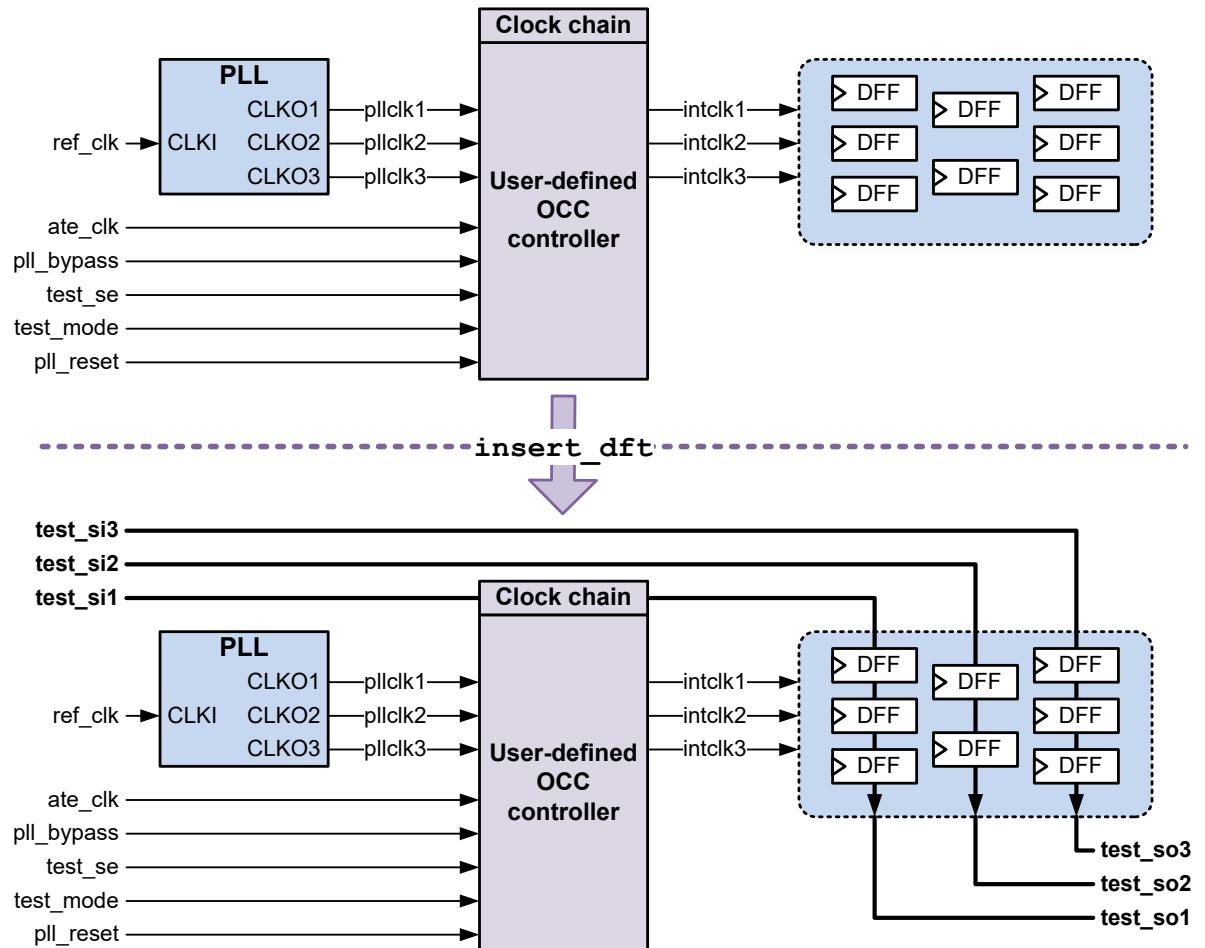


- Specifying Existing User-Defined OCC Controllers

The RTL contains the OCC controller and clock chain logic, all control signal and clock signal connections, and the connections from the clock chain to the OCC controller.

Before DFT insertion, this existing user-defined OCC controller and clock chain logic is described to the tool using the `set_dft_signal` and `set_scan_group` commands. The OCC controller design is validated and incorporated into the resulting DFT logic and test protocol. This flow is shown in [Figure 241](#).

Figure 241 User-Defined Clock Controller and Clock Chain Flow



Note that in this flow, the tool does not make any signal connections to the OCC controller or clock chain logic, except for scan data connections to the clock chain segments. You must ensure that all clock and control signal connections exist prior to DFT insertion.

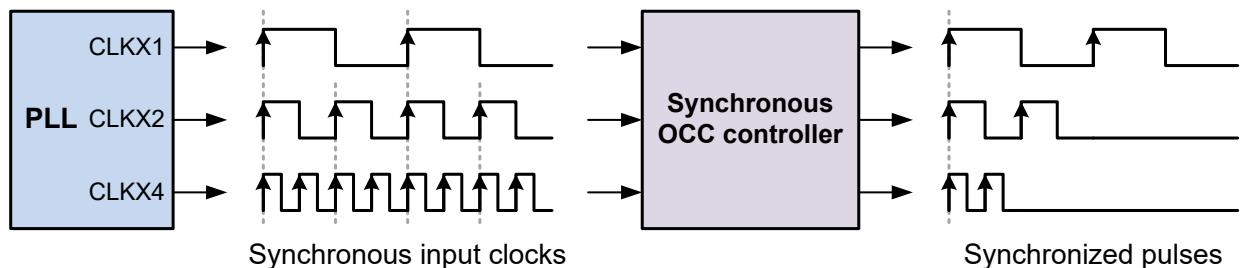
Synchronous and Asynchronous OCC Controllers

You can use synchronous or asynchronous OCC controllers, as described in this section:

- Synchronous OCC controller

The clocks controlled by the OCC controller are synchronous. The initial rising edges of the controlled clocks are synchronized to the lowest-frequency output clock, as shown in [Figure 242](#).

Figure 242 Synchronous OCC Controller Generating Two Clock Pulses



Synchronous OCC controllers have the following requirements:

- The rising edge of each clock being controlled must be at time zero of its waveform definition.
- The clocks being controlled **must** have a synchronous 1x, 2x, or 4x frequency multiplier relationship with respect to the lowest-frequency controlled clock.
- Capture paths between the controlled clock domains must capture data without hold violations.

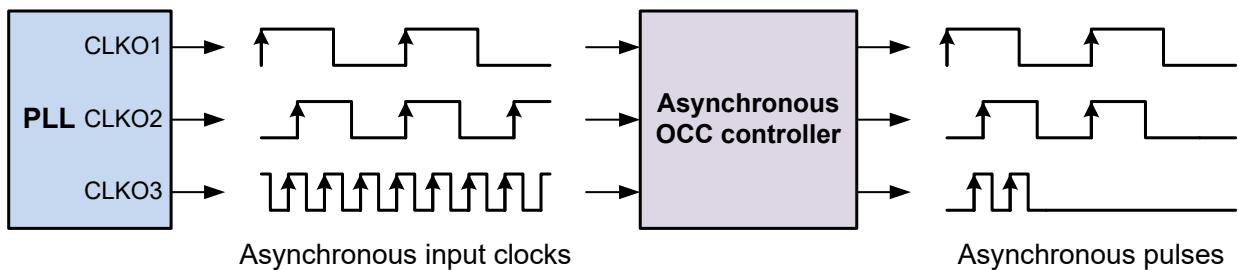
TestMAX ATPG understands that the controlled clock domains are synchronous; faults between them can be tested. For more information, see “Using Synchronized Multi-Frequency Internal Clocks” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Synchronous clock information is described in the `ClockTiming` block of a STIL protocol file (SPF). For more information, see “Specifying Synchronized Multi-Frequency Internal Clocks for an OCC Controller” in TestMAX ATPG and TestMAX Diagnosis Online Help.

- Asynchronous OCC controller

Each controlled clock uses dedicated clock control logic that generates clock pulses with no regard to alignment with other controlled clocks, as shown in [Figure 243](#). The input clocks being controlled can be synchronous or asynchronous with each other.

Figure 243 Asynchronous OCC Controller Generating Two Clock Pulses



ATPG treats the controlled clock domains as asynchronous; faults between them are not tested.

A synchronous OCC controller preserves the synchronous relationship of its input clocks, if they meet the requirements. You cannot use a synchronous OCC controller to control asynchronous clocks; it will not make them synchronous.

An asynchronous OCC controller uses separate, independent pulse generation logic for each controlled clock. You can use an asynchronous OCC controller to control synchronous clocks, but their synchronous relationships are lost; there is no guarantee that their pulse sequences will initiate on the same rising edge.

When a single OCC controller controls a mix of synchronous and asynchronous clocks, you must use an asynchronous controller. If needed, you can use a mix of synchronous and asynchronous OCC controllers.

OCC Controller Signal Operation

For [Figure 240 on page 524](#) and [Figure 241 on page 525](#), note the following:

- The reference clock (refclk) 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 (ate_clk) shifts the scan chain per tester specifications. Each PLL might have its own ATE clock.

See [Waveform and Capture Cycle Example on page 561](#) for a waveform diagram that demonstrates the relationship between the various clocks.

- The OCC controller 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 (pll_bypass) allows the ATE clock signal to connect directly to the internal clock signals, thus bypassing the PLL clocks.
- The ScanEnable signal (test_se) enables switching between the ATE shift clock and output PLL clock signals. ScanEnable must be inactive during every capture procedure, as described in [Scan-Enable Signal Requirements for OCC Controller Operation on page 530](#). You can use individual ScanEnable signals for each PLL clock signal.
- The TestMode signal (test_mode) must be active in order for the circuit to work.
- The OCC reset signal (pll_reset) is asserted during test setup to reset the OCC controller flip-flops to their initial states.

Clock Chain Operation

The clock chain provides a per-pattern clock selection mechanism for ATPG. It is implemented as a scan chain segment of one or more scan cells. Clock selection values are loaded into the clock chain as part of the regular scan load process.

A clock chain operates as follows:

- During scan shift, the clock chain shifts in new values when clocked by a scan clock.
The clock chain can be clocked by either the rising or falling clock edge, depending on what best fits into the overall DFT architecture.
- During scan capture, the clock chain holds its value.
The value scanned into the clock chain must be scanned out, undisturbed, after capture. The clock controller inserted by DFT Compiler meets this requirement. If you provide your own clock controller, ensure that it meets this requirement.

Note the following scan architecture aspects of clock chains:

- For standard scan designs, the clock chain can be a dedicated scan chain or a segment within a scan chain.
- For DFTMAX compressed scan designs, the clock chain can be an uncompressed (external) scan chain or a special segment within a compressed scan chain. For more information, see [Scan Compression and OCC Controllers on page 687](#).

- For DFTMAX Ultra compressed scan designs, the clock chain must be an uncompressed (external) scan chain. For more information, see [Using OCC Controllers With DFTMAX Ultra Compression on page 944](#).
- Clock chains of the same type (compressed or external) can be concatenated together. Compressed clock chains are concatenated into a single chain and placed inside the compressor where a regular single chain would be placed.

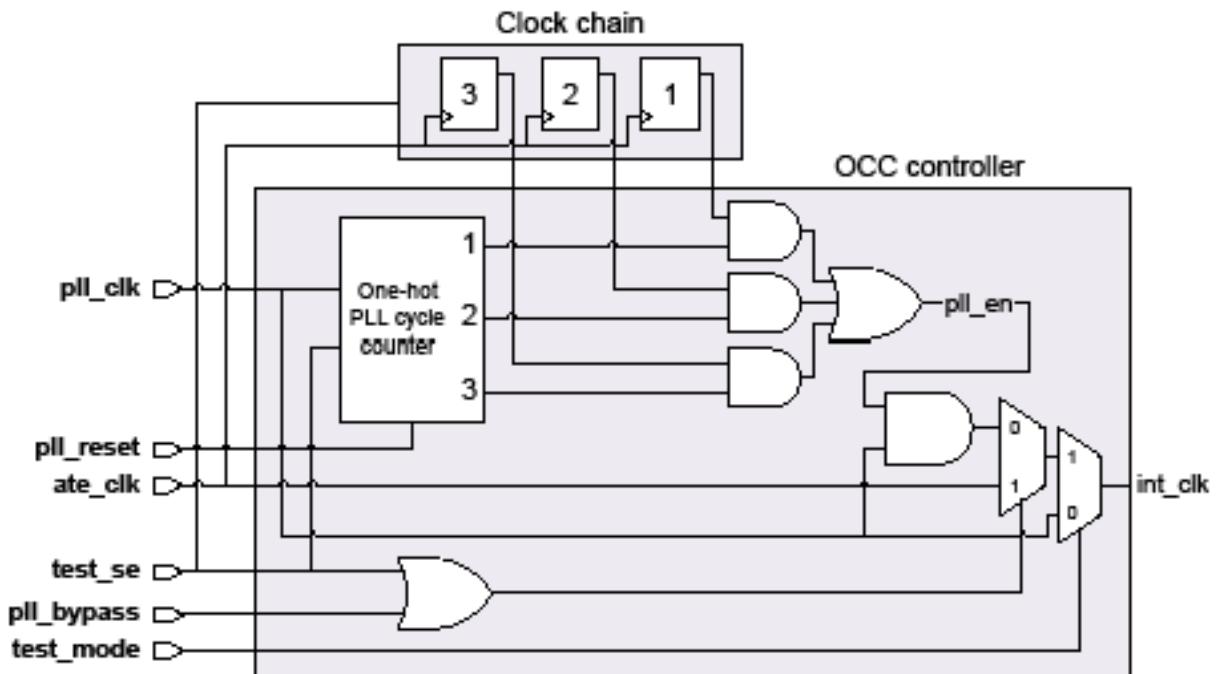
In the DFT-inserted OCC controller flow, the tool inserts a clock chain block that is separate from the OCC controller block.

In the user-defined OCC controller flow, the clock chain can be a part of the OCC controller design or it can be a separate design.

Logic Representation of an OCC Controller and Clock Chain

[Figure 244](#) shows the logic structure of an OCC controller and clock chain.

Figure 244 Logic Representation of an OCC Controller



Note: This is a conceptual logic view, not a functional implementation.

An OCC controller is designed to deliver up to a user-specified number N of at-speed clock pulse cycles during capture. A PLL cycle counter generates N successive one-hot

enable signals, each of which is gated by the output of a clock chain scan cell. This logic structure provides ATPG with the flexibility to control which cycles deliver an at-speed clock pulse. For an OCC controller that handles multiple OCC generators, the clock chain contains a set of N scan cells for each clock.

Note that the figure shows the conceptual operation of a single-clock OCC controller. Implementation details, such as cleanly switching between the PLL clock and ATE clock and providing synchronous or asynchronous control of multiple clocks, are not shown.

See Also

- [SolvNet article 034274, “DFT-inserted OCC Controller Data Sheet”](#) for more information about the logic structure and operation of the DFT-inserted OCC controller

Scan-Enable Signal Requirements for OCC Controller Operation

The scan-enable signal switches the OCC controller between the ATE shift clock and output PLL clock signals. Therefore, for proper operation, *the scan-enable signal must be held in the inactive state in all capture procedures*.

If you use the STIL protocol file created by the tool, the protocol already meets this requirement. The tool constrains all scan-enable signals to the inactive state in the capture procedures, excluding any scan-enable signals defined with the `-usage {clock_gating}` option of the `set_dft_signal` command. (Signals with multiple usages that include `clock_gating` are still constrained.)

If you use a custom STIL protocol file, make sure that all scan-enable signals used by OCC controllers are constrained to the inactive state in all capture procedures.

Enabling On-Chip Clocking Support

To enable OCC support for a design that contains or will contain OCC controllers, use the `-clock_controller` option of the `set_dft_configuration` command:

```
dc_shell> set_dft_configuration -clock_controller enable
```

In hierarchical flows, you also must enable OCC support at all hierarchical levels above those that contain OCC controllers. For more information, see [Using OCC Controllers in Hierarchical DFT Flows on page 551](#).

Specifying OCC Controllers

This topic covers the different methods of specifying OCC controllers in DFT Compiler:

- [Specifying DFT-Inserted OCC Controllers](#)
 - [Specifying Existing User-Defined OCC Controllers](#)
 - [Specifying OCC Controllers for External Clock Sources](#)
 - [Using OCC Controllers in Hierarchical DFT Flows](#)
-

Specifying DFT-Inserted OCC Controllers

If you have a design that contains an OCC generator, such as a PLL, but not an OCC controller and clock chain, DFT Compiler can insert both the OCC controller and clock chain. Note that this clock controller design supports only one ATE clock per OCC controller. This topic 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 topic covers the following:

- [Defining Clocks](#)
- [Defining Global Signals](#)
- [Configuring the OCC Controller](#)
- [Configuring the Clock Selection Logic](#)
- [Configuring the Clock-Chain Clock Connection](#)
- [Specifying Scan Configuration](#)
- [Performing Timing Analysis](#)
- [Script Example](#)

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 topic covers the following:

- [Reference Clocks](#)
- [PLL-Generated Clocks](#)
- [ATE Clocks](#)

Reference Clocks

Reference clock signals are always defined in the existing DFT view. The `insert_dft` command does not connect them because they are considered to be functional signals rather than test signals. The only effect of defining them is that they are defined in the test protocol for use by DRC in the TestMAX ATPG tool. For some special cases, a reference clock signal might not be needed.

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_dft \
    -type MasterClock -port refclk1 \
    -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
    -type refclock -port refclk1 \
    -period 100 -timing [list 45 55]
```

Note that you only need to define the reference clock with signal type `MasterClock` when the reference clock has the same period as the `test_default_period` variable. Otherwise, this signal definition is not needed and not accepted.

To define a reference clock that has a period other than the `test_default_period`, use the following command:

```
dc_shell> set_dft_signal -view existing_dft \
    -type refclock -port refclk1 \
    -period 10 -timing [list 3 8]
```

Note:

When the reference clock period differs from the `test_default_period`, do not define the signal as any signal type other than `refclock`.

Also note the following caveats associated with the `test_default_period` when defining a reference clock:

- 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. The TestMAX ATPG tool cannot work with finer timing resolutions.

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_dft \
    -type Oscillator \
    -hookup_pin PLL/pllclk1

dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -hookup_pin PLL/pllclk2

dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -hookup_pin PLL/pllclk3
```

For a synchronous OCC controller, you can optionally specify the clock period of the slowest-frequency clock with the `test_sync_occ_1x_period` variable. This variable affects the clock period values in the `ClockTiming` block of the STIL protocol file. Although the value does not affect pattern generation in TestMAX ATPG, you can specify it for informational purposes. For more information, see the man page.

ATE Clocks

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_dft \
    -type ScanClock \
    -port ATEclk \
    -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -port ATEclk
```

The ATE clock must be defined as both `-type ScanClock` and `-type Oscillator`. The `ScanClock` signal definition uses the `-view existing_dft` option because the `-timing` option can be specified only in that view. The `Oscillator` signal definition uses the `-view existing_dft` option so that the ATE clock is modeled as a free-running clock in the test protocol.

By default, DFT Compiler makes the ATE clock connection at the source port specified in the `-view existing_dft` signal definition. To specify a hookup pin to be used for the clock connection, use the `-hookup_pin` option in a subsequent `-view spec` scan clock signal definition. For example,

```
dc_shell> set_dft_signal -view spec \
    -type ScanClock \
    -port ATEclk \
    -hookup_pin PAD_ateclk/z
```

Define this additional specification only for the `ScanClock` signal definition; do not define it for the `Oscillator` signal definition.

Note:

You can use the same clock port as both the ATE clock and PLL reference clock. However, caveats apply. For more information, see [SolvNet article 037838, "How Can I Use the Same Clock Port for the ATE and PLL Reference Clocks?"](#)

See Also

- [Specifying a Hookup Pin for DFT-Inserted Clock Connections on page 246](#) for more information about clock hookup pins

Defining Global Signals

You must identify the top-level interface signals that control the OCC controller. This includes the OCC bypass, OCC reset, and ScanEnable signals. You must also define a dedicated TestMode signal that activates the OCC controller logic. In the OCC controller insertion flow, these signals are defined with the `-view spec` option because they will be implemented and connected by the `insert_dft` command.

The following examples show how to define a set of OCC controller interface signals for the design example:

```
dc_shell> set_dft_signal -view spec \
    -type pll_reset \
    -port OCC_reset

dc_shell> set_dft_signal -view spec \
    -type pll_bypass \
    -port PLL_bypass

dc_shell> set_dft_signal -view spec \
    -type ScanEnable \
    -port SE

dc_shell> set_dft_signal -view spec \
```

```
-type TestMode \
-port TM_OCC
```

The TestMode signal must be a dedicated signal for the OCC controller. It must be active in all test modes and inactive in mission mode. It cannot be shared with TestMode signals used for other purposes, such as AutoFix or multiple test-mode selection.

Note:

In the internal pins flow, you can specify internal hookup pins for these OCC control signals by using the `-hookup_pin` option of the `set_dft_signal` command. However, you cannot specify internal hookup pins for ATE clocks or reference clocks.

Configuring the OCC Controller

To specify where to insert a DFT-inserted OCC controller, use the `set_dft_clock_controller` command. Note the following syntax and descriptions:

```
set_dft_clock_controller
[-cell_name cell_name]
[-design_name design_name]
[-pllclocks ordered_list]
[-1x_clocks ordered_list]
[-2x_clocks ordered_list]
[-4x_clocks ordered_list]
[-ateclocks clock_name]
[-chain_count integer]
[-cycles_per_clock integer]
[-test_mode_port port_name]
```

Option	Description
<code>-cell_name cell_name</code>	Specifies the hierarchical name of the clock controller cell.
<code>-design_name design_name</code>	Specifies the OCC controller design name. You must specify <code>smps_clk_mux</code> .
<code>-pllclocks ordered_list</code>	For asynchronous OCC controllers, specifies the ordered list of PLL output clock pins to control.
<code>-1x_clocks ordered_list</code>	For synchronous OCC controllers, specifies the ordered list of PLL output clock pins to control that run at the slowest frequency.
<code>-2x_clocks ordered_list</code>	For synchronous OCC controllers, specifies the ordered list of PLL output clock pins to control that run at two times the slowest frequency.

Option	Description
<code>-4x_clocks ordered_list</code>	For synchronous OCC controllers, specifies the ordered list of PLL output clock pins to control that run at four times the slowest frequency.
<code>-ateclocks clock_name</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 integer</code>	Specifies the number of clock chains. The default number of clock chains is one.
<code>-cycles_per_clock integer</code>	Specifies the maximum number of capture cycles per clock. You should specify a value of two or greater. 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.
<code>-test_mode_port port_name</code>	Specifies the test-mode port used to enable the clock controller. Use this option if you have multiple test-mode ports and you want to use a specific port to enable the clock controller. The specified port must be defined as a TestMode signal using the <code>set_dft_signal</code> command.

For asynchronous OCC controllers, use the `-pllclocks` option to specify the list of hierarchical clock source pins to be controlled.

For synchronous OCC controllers, use the `-1x_clocks`, `-2x_clocks`, and `-4x_clocks` options to specify the list of synchronous hierarchical clock source pins to be controlled. The `-1x_clocks` option is required; the first clock in the list is used as the master synchronization clock. The `-2x_clocks` and `-4x_clocks` options are optional.

The following example inserts an asynchronous OCC controller that controls three clocks:

```
dc_shell> set_dft_clock_controller \
    -cell_name occ_snps \
    -design_name snps_clk_mux \
    -pllclocks { p11/p11clk1 p11/p11clk2 p11/p11clk3 } \
    -ateclocks { ATEclk } \
    -cycles_per_clock 2 -chain_count 1
```

To insert multiple OCC controllers, use multiple `set_dft_clock_controller` commands. You can insert a mix of synchronous and asynchronous OCC controllers.

Configuring the Clock Selection Logic

By default, the OCC controller uses purely combinational clock-gating logic for glitch-free selection between the fast and slow clocks. However, this is a legacy clock selection logic structure that can introduce a number of logic gates along the fast and slow clock paths, and its implementation is not configurable.

To avoid these issues, you can use latch-based clock-gating logic. For the fast and slow clocks, the combinational gating-enable signals are combined and latched for each clock; the latched enable signal is then used to gate the clock.

When latch-based clock-gating logic is used, the gate structure used to combine the clocks is also configurable. The clock selection logic configuration is global and applies to all OCC controllers inserted by DFT insertion in the current design. Cores that contain previously inserted OCC controllers can use a different clock selection logic configuration.

Note:

DFT insertion applies the `dont_touch` attribute to any library cells referenced by the design attributes (but not variables) in this section. This implicitly applies the `dont_touch` attribute to any other instances of these library cells in your design, which excludes them from optimization after DFT insertion. To prevent this, set the `dont_touch` attribute on the OCC controller designs and remove it from the attribute-referenced library cells after DFT insertion.

See Also

- [SolvNet article 034274, “DFT-inserted OCC Controller Data Sheet”](#) for more information about the combinational and latch-based clock selection logic structures

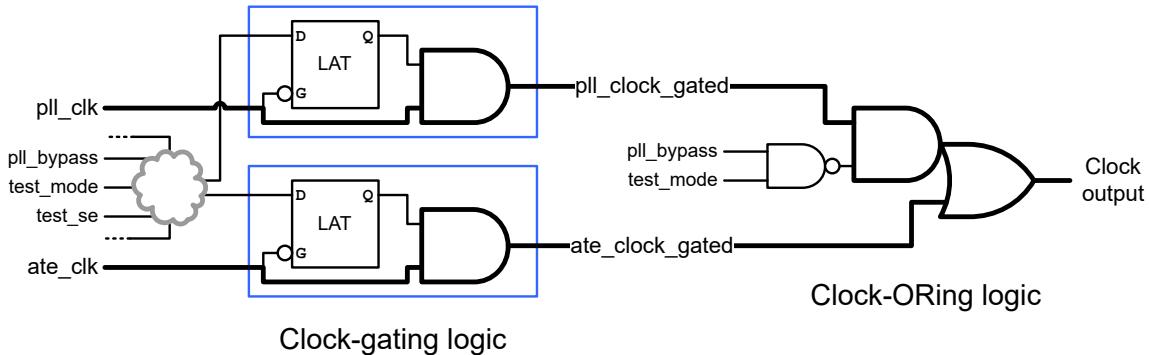
Using Latch-Based Clock-Gating Logic

To use latch-based clock-gating logic, set the following variable to `true`:

```
dc_shell> set_app_var test_occ_insert_clock_gating_cells true
```

The default latch-based clock-gating logic structure is shown in [Figure 245](#). Clock paths are shown in bold, and rectangles indicate hierarchy created inside the OCC controller block by DFT insertion.

Figure 245 Default Latch-Based Clock-Gating and Clock Selection Logic Structure



The tool always uses latch-based clock gating for synchronous OCC controllers. If you have not set the `test_occ_insert_clock_gating_cells` variable to `true`, the tool issues a warning indicating that latch-based clock gating will be used for them.

When you use latch-based clock gating in a serialized compressed scan flow, the `test_elpc_unique_fsm` variable must be set to its default of `true`. For details on this variable, see [Serializer in Conjunction With On-Chip Clocking Controllers on page 883](#).

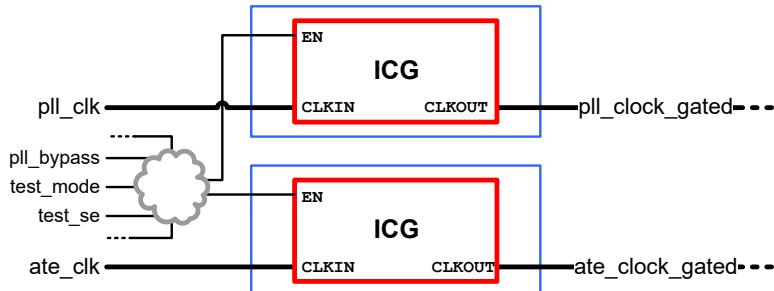
Specifying Library Cells for the Clock-Gating Logic

You can configure the library cells used for the clock-gating logic. This allows you to use higher-drive cells or specially designed clock-driver cells along the clock path.

By default, the latch-based clock-gating logic uses discrete latch cells. To use an integrated clock-gating cell instead, as shown in [Figure 246](#), set the desired library cell reference (without the library name) using the `test_icg_p_ref_for_dft` variable. For example,

```
dc_shell> set_app_var test_icg_p_ref_for_dft ICGPTX8
```

Figure 246 Specifying an Integrated Clock-Gating Cell for the Clock-Gating Logic



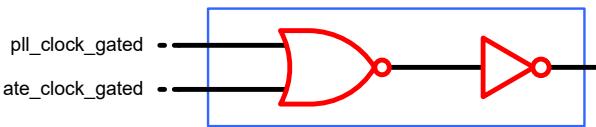
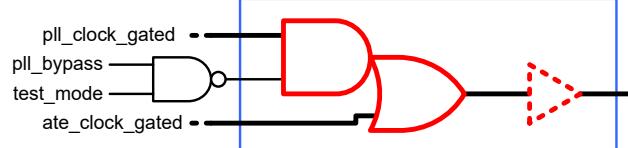
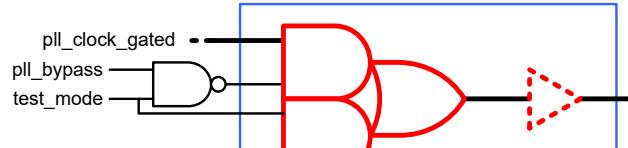
For more information about this variable, see the man page. For information on clock gating styles, see the *Power Compiler User Guide*.

Specifying Library Cells for the Clock-ORing Logic

You can configure the library cells used for the clock-ORing logic. This allows you to use higher-drive cells or specially designed clock-driver cells along the clock path.

By default, gate-level synthesis maps the default clock-ORing logic structure to any suitable gate configuration. You can specify the library cells used for the clock-ORing logic by setting design attributes on the current design to the desired library cell references (without the library name). [Table 48](#) shows the allowed attribute combinations. Buffer specifications are optional.

Table 48 Specifying Library Cells for the Clock-ORing Logic

Logic structure	Structure type and design attributes
	NOR2 clock ORing ¹⁰ occ_lib_cell_nor2 occ_lib_cell_clkinv
	ANDOR21 clock ORing occ_lib_cell_andor21 occ_lib_cell_clkbuf (optional)
	ANDOR22 clock ORing occ_lib_cell_andor22 occ_lib_cell_clkbuf (optional)

For example,

```
dc_shell> set_attribute [current_design] occ_lib_cell_andor21 CKAO21X8
```

Note:

The NOR2 logic structure requires that the design provide an active PLL clock during OCC bypass mode because this structure does not include a logic term that blocks unknown initial values from an unclocked PLL-clocked gating latch. Post-DFT DRC is not supported for PLL bypass mode. In TestMAX ATPG, run the following commands before running DRC for PLL bypass mode:

```
DRC-T> add_clocks 0 pll_pin_pathname -pllclock
DRC-T> set_drc -pll_simulate_test_setup
```

10. Requires manual intervention to pass post-DFT DRC in PLL bypass mode - see note.

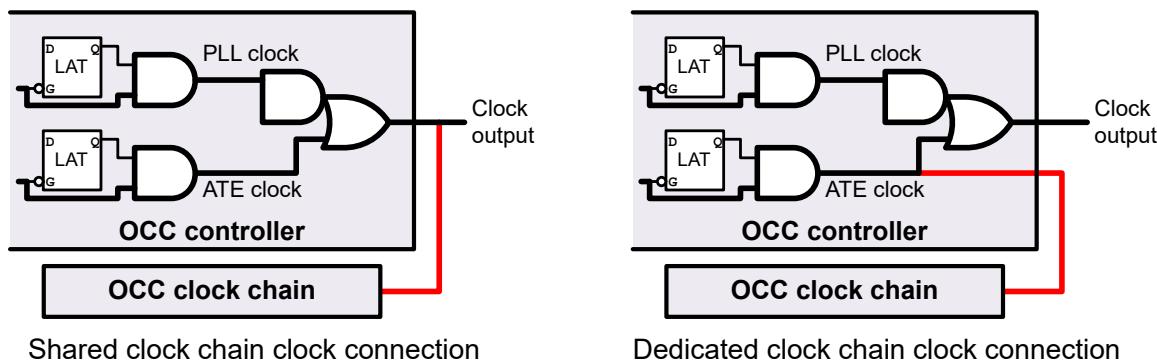
Configuring the Clock-Chain Clock Connection

By default, the clock-chain clock connection shares the first functional clock output of the OCC controller. This places the clock chain in both the PLL and ATE clock paths.

To use a dedicated clock-chain clock connection from the OCC controller design, set the `test_dedicated_clock_chain_clock` variable to `true`. This creates a dedicated OCC controller clock output for the clock chain that places it in only the ATE clock path, which prevents it from affecting the high-speed PLL clock path.

[Figure 247](#) shows both types of clock-chain clock connections.

Figure 247 Shared and Dedicated Clock Chain Clock Connections



In both cases, you should consider how the clock-chain clock connection interacts with clock tree synthesis (CTS). For more information, see [man page](#). Also, note that this variable can be set to `true` only when the `test_occ_insert_clock_gating_cells` variable is also set to `true`.

Specifying Scan Configuration

Use 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>...
```

If the current design is to be used as a test model later in a hierarchical flow, it is important to avoid clock mixing. Such mixing can cause the clock chain of the OCC controller to mix with flip-flops of opposite polarity on a single scan chain. As a result, this scan chain cannot be combined with scan chains of other test models and the minimum scan chain count at the top level is increased. This problem is worsened when multiple OCC controllers are added to the design or when multiple subdesigns of the top-level design will have OCC controllers.

Performing Timing Analysis

After DFT insertion completes, you must ensure that the OCC controller logic is properly constrained for timing analysis.

If you are using the combinational clock-gating method and synthesis maps the clock selection logic to a MUX cell, you must use the `set_clock_gating_check` command to manually specify a clock-gating check at the MUX inputs. 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). Combinational clock gating is used when the `test_occ_insert_clock_gating_cells` variable is set to its default of `false`.

See Also

- [SolvNet article 034274, "DFT-Inserted OCC Controller Data Sheet"](#)
The two sections titled “Special Considerations” for information about synthesizing a DFT-inserted OCC controller
- [SolvNet article 022490, “Static Timing Analysis Constraints for On-Chip Clocking Support](#) for more information about performing timing analysis in a DFT-inserted OCC controller flow

Script Example

[Example 67](#) shows DFT Compiler performing pre-DFT DRC, scan chain stitching, and post-DFT DRC. The STIL protocol file generated at the end of the DFT insertion process contains PLL clock details suitable for the TestMAX ATPG tool.

Example 67 Flow Example for DFT-Inserted OCC Controller and Clock Chain

```
read_verilog mydesign.v
current_design mydesign
link

# Define the PLL reference clock
# top level free running clock
set_dft_signal -view existing_dft -type refclock \
    -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -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_dft -type ScanClock \
    -port ATEcclk -timing [list 45 55]

set_dft_signal -view existing_dft -type Oscillator \
    -port ATEcclk
```

```

# 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_dft -type Oscillator \
    -hookup_pin pll/pllclk1

set_dft_signal -view existing_dft -type Oscillator \
    -hookup_pin pll/pllclk2

set_dft_signal -view existing_dft -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_name snps_clk_mux -pll_clocks { pll/pllclk1 \
    pll/pllclk2 pll/pllclk3 } -ateclocks { ATEclk } \
    -cycles_per_clock 2 -chain_count 1

set_scan_configuration -chain_count 30 -clock_mixing no_mix
create_test_protocol
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 -hierarchy
write -format verilog -hierarchy -output top.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -output scan_pll.stil

```

Specifying Existing User-Defined OCC Controllers

If you have a design that contains an OCC generator and it already instantiates an existing user-defined OCC controller and clock chain, DFT Compiler can analyze the OCC controller, validate the functionality, and incorporate it into the test protocol.

This topic covers the following:

- [Defining Clocks](#)
- [Defining Global Signals](#)
- [Specifying Clock Chains](#)
- [Scan Configuration for User-Defined OCC Controllers](#)
- [Script Example](#)

Defining Clocks

Use the `set_dft_signal` command to define the following clock signals for [Figure 241](#):

- [Reference Clocks](#)
- [PLL-Generated Clocks](#)
- [ATE Clocks](#)
- [Clock Chain Configuration and Control-Per-Pattern Information](#)

Reference Clocks

A reference clock definition is used primarily as an informational device. The only effect of defining them is that they are defined in the test protocol for use by TestMAX ATPG. For some special cases, a reference clock signal might not be needed.

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_dft \
    -type MasterClock -port refclk1 \
    -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
    -type refclock -port refclk1 \
    -period 100 -timing [list 45 55]
```

Note that you only need to define the reference clock with signal type `MasterClock` when the reference clock has the same period as the `test_default_period` variable. Otherwise, this signal definition is not needed and not accepted.

To define a reference clock that has a period other than the `test_default_period`, use the following command:

```
dc_shell> set_dft_signal -view existing_dft \
    -type refclock -port refclk1 \
    -period 10 -timing [list 3 8]
```

Note:

When the reference clock period differs from the `test_default_period`, do not define the signal as any signal type other than `refclock`.

Also note the following caveats associated with the `test_default_period` when defining a reference clock:

- 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. The TestMAX ATPG tool cannot work with finer timing resolutions.

PLL-Generated 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 the TestMAX ATPG tool through the protocol file to allow the verification of the clock controller logic.

The following commands show how to define the PLL clocks for the design example:

```
dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -hookup_pin PLL/pllclk1

dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -hookup_pin PLL/pllclk2

dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -hookup_pin PLL/pllclk3
```

For synchronous OCC controllers, you must also update the STIL protocol file with an appropriate `ClockTiming` block before using it in TestMAX ATPG. See the “Specifying Synchronized Multi-Frequency Internal Clocks for an OCC Controller” section in TestMAX ATPG and TestMAX Diagnosis Online Help.

ATE Clocks

An ATE clock signal can be pulsed several times before and after scan shift (scan-enable signal 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 design example:

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

dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -port ATEclk
```

ATEclk must be defined as `-type ScanClock` and `-type Oscillator`.

Note:

You can use the same clock port as both the ATE clock and PLL reference clock. However, caveats apply. For more information, see [SolvNet article 037838, “How Can I Use the Same Clock Port for the ATE and PLL Reference Clocks?”](#)

A user-defined OCC controller can use ATE clock synchronization logic which differs from the DFT-inserted OCC controller. In these cases, you might need to set the `test_ate_sync_cycles` variable to a nondefault value. For more information, see [SolvNet article 035708, “What Does the test_ate_sync_cycles Variable Do?”](#)

Clock Chain Configuration 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 provided by the user-defined clock chain. This information indicates how the clock signal is enabled by the clock chain control bits in each clock cycle. The correspondence between the controlled internal clock signals and clock chain control bits is identified in the protocol file for TestMAX ATPG to generate patterns.

You must specify

- All user-defined clock controller outputs referencing the internal clocks
- A corresponding set of clock chain control bits, ATE clock, and clock generator output (PLL) for each clock controller output

The following example specifies a user-defined OCC controller with a three-bit clock chain:

```
dc_shell> set_dft_signal -view existing_dft \
    -type Oscillator \
    -hookup_pin occ_ctrl1/clkout \
    -ate_clock ATEclk \
    -pll_clock PLL/pllclk1 \
    -ctrl_bits [list 0 occ_chn/FF_cyc1/Q 1 \
                1 occ_chn/FF_cyc2/Q 1 \
                2 occ_chn/FF_cyc3/Q 1]
```

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 the cycle where the clock signal will be propagated. The second element is the pin name (a valid design hierarchical pin name) of the clock chain control bit. The third element is the active state (0 or 1) of the control bit signal. For more information about this option, see the `set_dft_signal` man page.

Note:

The `-view existing_dft` option is used because connections already exist between the referenced port and the clock controller.

Defining Global Signals

You must identify the top-level interface signals that control the OCC controller. This includes the OCC bypass, OCC reset, and ScanEnable signals. You must also define a dedicated TestMode signal that activates the OCC controller logic. In the user-defined OCC controller flow, these signals are defined with the `-view existing_dft` option because they already exist and must be described to DFT Compiler.

The following examples show how to define a set of OCC controller interface signals for the design example:

```
dc_shell> set_dft_signal -view existing_dft \
    -type pll_reset \
    -port OCC_reset

dc_shell> set_dft_signal -view existing_dft \
    -type pll_bypass \
    -port PLL_bypass

dc_shell> set_dft_signal -view existing_dft \
    -type ScanEnable \
    -port SE

dc_shell> set_dft_signal -view existing_dft \
    -type TestMode \
    -port TM_OCC
```

The TestMode signal must be a dedicated signal for the OCC controller. It must be active in all test modes and inactive in mission mode. It cannot be shared with TestMode signals used for other purposes, such as AutoFix or multiple test-mode selection.

Note:

You can specify an internal hookup pin for any of these OCC controller interface signals by using the `-hookup_pin` option of the `set_dft_signal` command.

You cannot specify internal hookup pins for ATE clocks or reference clocks.

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 \
    -class OCC \
    -include_elements \
    [list occ_chn/FF_1 occ_chn/FF_2] \
    -access [list ScanDataIn occ_chn/si \
        ScanDataOut occ_chn/so \
        ScanEnable occ_chn/se] \
    -serial_routed true
```

In this flow, you should insert the clock chain to be clocked by the falling edge of the internal clock in such a way that it can be placed at the head of a scan chain. When defining the clock chain, use the `set_scan_group -class OCC` command to specify the special treatment of the clock chain, and avoid using the `set_scan_path` command so that the scan architect has maximum flexibility in putting the clock chain in the best location on the scan chains. The `-class OCC` option allows the clock chain to be recognized if the module is incorporated as a test model in the integration flow.

In case a separate scan path is required for the clock chain, the `set_scan_path -class OCC` command is also available. If you are using scan compression, the clock chain does not need to be a separate scan path, but if you want to define it as such, two `set_scan_path` commands are required, one with the `-test_mode Internal_scan` option and one with the `-test_mode ScanCompression_mode` option. The process of combining scan compression with clock controllers results in a multiple test-mode architecture; therefore, both modes must be specified.

If the current design is to be used later as a test model in a hierarchical flow and your scan configuration allows clock mixing, you should make sure that the clock chains are kept separate from other scan chains. Otherwise, the top-level scan architecture might require too many scan chains because the submodule scan chains are incompatible with each other. To force the clock chains to be separate, use the `set_scan_path` command with the `-class occ` and `-complete true` options. For more information, see [SolvNet article 018046, "How Can I Control Scan Stitching of OCC Controller Clock Chains?"](#)

Scan Configuration for User-Defined OCC Controllers

For proper clock control during test, your user-defined OCC controller and clock chain must be excluded from scan replacement. To do this, use the `set_scan_element` command:

```
dc_shell> set_scan_element false {occ_ctrl occ_chn}
```

For convenience, you can apply this directive using RTL pragmas in your user-defined OCC RTL. For details, see [SolvNet article 2898128, “Excluding User-Defined OCC Controllers from DFT Insertion.”](#)

If the current design is to be used later as a test model in a hierarchical flow, it is important to avoid clock mixing. If you must mix clocks, use the method described in [Specifying Clock Chains on page 547](#) to avoid problems integrating the clock chains at the next higher level of hierarchy.

If your OCC controller uses integrated clock-gating cells, verify that pre-DFT DRC does not issue any TEST-130 identification messages for them, or incorrect test-pin connections will result. For details, see [SolvNet article 2819507, “Simulation Fails Due to Bad ICG Test-Pin Connection in User-Defined OCC Controller.”](#)

Script Example

When you run a design that contains a user-defined OCC controller and clock chains, a STIL protocol file is generated, as shown in [Example 68](#).

Example 68 Flow Example for Existing OCC Controller and Clock Chain

```
read_verilog mydesign.v
current_design mydesign
link

# Define the PLL reference clock
# (top-level free running clock)
set_dft_signal -view existing_dft -type refclock \
    -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -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_dft -type ScanClock \
    -port ATEcclk -timing [list 45 55]

set_dft_signal -view existing_dft -type Oscillator \
    -port ATEcclk

# Define the PLL generated clocks
set_dft_signal -view existing_dft -type Oscillator \
```

Chapter 12: On-Chip Clocking Support

Specifying OCC Controllers

```

-h hookup_pin pll/pllclk1

set_dft_signal -view existing_dft -type Oscillator \
    -hookup_pin pll/pllclk2

set_dft_signal -view existing_dft -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_ctrl/clkout \
    -ate_clock ATEclk -pll_clock PLL/pllclk1 \
    -ctrl_bits [list 0 occ_chn/FF_1/Q 1 \
                1 occ_chn/FF_2/Q 1] \
    -view existing_dft

# Define the existing clock chain segments
set_scan_group clk_chain -class occ \
    -include_elements [list occ_chn/FF_1 \
                        occ_chn/FF_2] \
    -access [ list ScanDataIn occ_chn/si \
              ScanDataOut occ_chn/so \
              ScanEnable occ_chn/se] \
    -serial_routed true

# Specify global controller signals
set_dft_signal -type pll_reset -port OCC_reset -view existing_dft
set_dft_signal -type pll_bypass -port PLL_bypass -view existing_dft
set_dft_signal -type ScanEnable -port SE -view existing_dft

# Define the TestMode signals
set_dft_signal -type TestMode -port TM_OCC -view existing_dft

# Registers inside the OCC controller and clock chain should not
# be scan-replaced. (The OCC controller requires nonscan cells, and the
# clock chain is already a stitched scan segment.)
set_scan_element false {occ_ctrl occ_chn}

# If you are using automatic clock-gating cell identification (the
# identify_clock_gating command or the power_cg_auto_identify variable),
# prevent identification inside the OCC controller.
set_dft_clock_gating_configuration -exclude_elements {occ_ctrl}

set_scan_configuration -chain_count 30 -clock_mixing no_mix
create_test_protocol
dft_drc
report_dft_clock_controller -view existing_dft
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

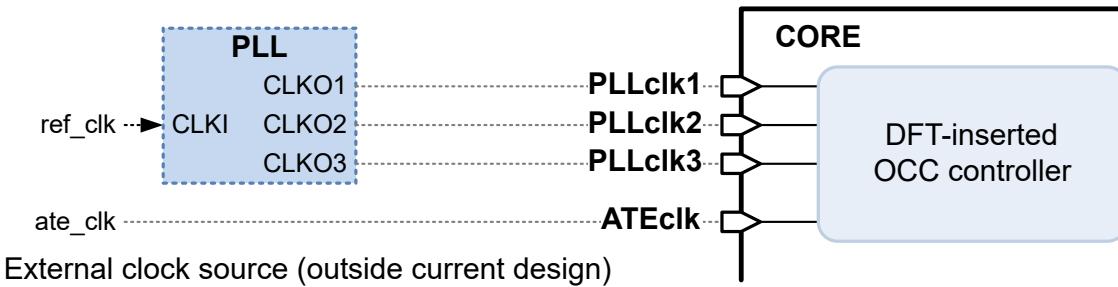
change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output top.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -output scan_pll.stil
```

Specifying OCC Controllers for External Clock Sources

In some cases, the on-chip clocking source might be external to the current design, as shown in [Figure 248](#), so that the PLL clocks enter the design through input ports.

Figure 248 On-Chip Clock Source External to Current Design



Define such external clock sources as follows:

- Define the port-driven PLL-generated clocks at the ports, and define them as `ScanClock` instead of `Oscillator`:

```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock \
    -port {PLLclk1 PLLclk2 PLLclk3} \
    -timing {45 55}
```

Specify typical scan clock timing, even though the clock is a PLL-generated clock instead of a tester-driven clock. This timing discrepancy does not affect OCC architecture or operation.

- If the reference clock does not enter the design, you do not need to define it.
- Include the PLL-generated input ports in the PLL clock sources list specified with the `-pllclocks` option:

```
dc_shell> set_dft_clock_controller \
    -cell_name occ_snps \
```

```
-design_name snps_clk_mux \
-pllclocks {PLLclk1 PLLclk2 PLLclk3} \
-ateclocks {ATEclk} \
-cycles_per_clock 2 -chain_count 1
```

The resulting SPF does not pulse the clock ports during capture, which is incorrect. As a workaround, define the clock as a reference clock in the TestMAX ATPG tool. For example,

```
DRC-T> add_clock 0 {PLLclk1 PLLclk2 PLLclk3} -refclock
```

External clock sources are not supported for existing user-defined OCC controllers.

Using OCC Controllers in Hierarchical DFT Flows

You can integrate DFT-inserted cores that contain OCC controllers, as described in the following topics:

- [Integrating Cores That Contain OCC Controllers](#)
- [Defining Signals for Cores Without Preconnected OCC Signals](#)
- [Defining Signals for Cores With Preconnected OCC Signals](#)
- [Handling Cores With OCC Clock Output Pins](#)

Integrating Cores That Contain OCC Controllers

You can integrate DFT-inserted cores that contain OCC controllers. It does not matter whether a core was created with the DFT-inserted or user-defined OCC controller flow; once created, it is simply a core that contains OCC controller information in its CTL model, and the hierarchical OCC integration flow treats the core the same in either case.

Enable the OCC controller feature at all hierarchical levels above those that contain OCC controllers:

```
dc_shell> set_dft_configuration -clock_controller enable
```

Top-level connections for the `pll_reset`, `pll_bypass`, ATE clock, and OCC TestMode signals of all core-level OCC controllers should be either all preconnected or all left unconnected so that the `insert_dft` command can make the connections. The DFT architect cannot recognize partially connected conditions reliably and might make mistakes if some, but not all, of these signals are already connected.

If you define a top-level DFT-inserted OCC controller, and the current design contains cores that have their own OCC controllers, all core-level OCC signals must be unconnected so that the `insert_dft` command can make the top-level connections. Preconnected core-level OCC signals are not supported when a top-level DFT-inserted OCC controller is used.

Defining Signals for Cores Without Preconnected OCC Signals

Top-level integration uses signal types to determine the correct connections to make. Most DFT signals used by OCC controllers, including the `pll_reset` and `pll_bypass` types, can be connected by the `insert_dft` command.

However, the ATE clock signal has no special signal type in the core test model, so user guidance is required. To give this guidance, use the `-connect_to` option of the `set_dft_signal` command.

Example 69 defines the OCC signals in an integration flow where the `insert_dft` command must make the OCC signal connections to two cores containing OCC controllers.

Example 69 OCC Signal Definitions for Cores Without Preconnected OCC Signals

```
# Define the PLL reference clock (top-level free-running clock)
#
# this is a functional signal that must always be preconnected
set_dft_signal -view existing_dft -type refclock \
    -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -type MasterClock \
    -port refclk1 -timing [list 45 55]

# Define the ATE clock
#
# this is the ATE-provided clock for shift of scan elements
set_dft_signal -view existing_dft -type ScanClock \
    -port ATEclk -timing [list 45 55] \
    -connect_to [list CORE1/ATEclk CORE2/ATEclk]

set_dft_signal -view spec -type ScanClock \
    -port ATEclk \
    -connect_to [list CORE1/ATEclk CORE2/ATEclk]

set_dft_signal -view existing_dft -type Oscillator \
    -port ATEclk

# Specify global OCC controller signals, all in spec view
set_dft_signal -view spec -type pll_reset -port OCC_reset
set_dft_signal -view spec -type pll_bypass -port PLL_bypass
set_dft_signal -view spec -type ScanEnable -port SE

# Define the PLL TestMode signals
set_dft_signal -view spec -type TestMode -port TM_OCC
```

This example uses both the `-view existing_dft` and `-view spec` options for the ATE clock. The `-view spec` option specifies that a change to the design is needed during DFT insertion, but clock timing can only be attached to a clock specification in the existing view.

For most signal types, the `-connect_to` option is used only with `-view spec` signal definitions to define connections to be made by DFT insertion. However, for ATE clocks, the `-connect_to` option is also used for the `-view existing_dft` signal definition so that the information is passed to pre-DFT DRC.

Defining Signals for Cores With Preconnected OCC Signals

When the DFT signals used by OCC controllers are preconnected to the cores, define them at the top level with only the `-view existing_dft` option of the `set_dft_signal` command.

[Example 70](#) defines the OCC signals in an integration flow where the OCC connections to two cores already exist.

Example 70 OCC Signal Definitions for Cores With Preconnected OCC Signals

```
# Define the PLL reference clock (top-level free-running clock)
#
# this is a functional signal that must always be preconnected
set_dft_signal -view existing_dft -type refclock \
    -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -type MasterClock \
    -port refclk1 -timing [list 45 55]

# Define the ATE clock
#
# this is the ATE-provided clock for shift of scan elements
set_dft_signal -view existing_dft -type ScanClock \
    -port ATEclk -timing [list 45 55] \
    -connect_to [list CORE1/ATEclk CORE2/ATEclk]

set_dft_signal -view existing_dft -type Oscillator \
    -port ATEclk

# Specify global OCC controller signals, all in existing_dft view
set_dft_signal -view existing_dft -type pll_reset -port OCC_reset
set_dft_signal -view existing_dft -type pll_bypass -port PLL_bypass
set_dft_signal -view existing_dft -type ScanEnable -port SE

# Also define ScanEnable in spec view for top-level DFT insertion
set_dft_signal -view spec -type ScanEnable -port SE

# Define the PLL TestMode signals
set_dft_signal -view existing_dft -type TestMode -port TM_OCC
```

For most signal types, the `-connect_to` option is used only with `-view spec` signal definitions to define connections to be made by DFT insertion. However, for ATE clocks, the `-connect_to` option is also used for the `-view existing_dft` signal definition so that the information is passed to pre-DFT DRC.

Handling Cores With OCC Clock Output Pins

When a core has a clock output driven by an OCC clock generated inside the core, the core CTL model models the clock output as a clock feedthrough connection from the corresponding ATE clock. For example,

```
Internal {
    "ATECLK" {
        DataType ScanMasterClock MasterClock;
    }
    "PLL_CLK_OUT" {
        IsConnected Out {
            Signal "ATECLK";
        }
    }
}
```

At the integration level, if you have multiple such cores that use the same ATE clock, by default DFT insertion does not insert lockup latches for scan chain crossings between the core output clock domains because they are not seen as separate clock domains. To treat these clock domains as separate, specify the core clock output pins with the `-associated_internal_clocks` option for the ATE clock signal definition. For example,

```
set_dft_signal -view existing_dft -type ScanClock \
    -port ATEclk -timing [list 45 55] \
    -associated_internal_clocks {CORE1/PLL_CLK_OUT CORE2/PLL_CLK_OUT}
```

This causes DFT insertion to treat the core output clock domains as separate domains for scan architecting purposes. No special treatment is needed for any top-level OCC controllers that also share the same ATE clock signal, because the tool already treats their output clock domains as separate domains.

Reporting Clock Controller Information

Use the `report_dft_clock_controller` command to generate reports.

DFT-Inserted OCC Controller Flow

For DFT-inserted OCC controllers and clock chains, use the `report_dft_clock_controller -view spec` command to output a report. This report displays the options that you set for the `set_dft_clock_controller` command.

[Example 71](#) shows a clock controller report for the DFT-inserted OCC controller flow.

Example 71 Report Example from the report_dft_clock_controller -view spec Command

```
*****
Report : Clock controller
Design : des_chip
```

```

Version: G-2012.06
Date   : Fri Sep  7 05:42:06 2012
*****
=====
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

```

Existing User-Defined OCC Controller Flow

For existing user-defined OCC 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_dft` command to report what you have specified. In [Example 72](#), the report shows information about the clock generator signal, the ATE clock, the OCC controller output, and the clock chain control bits.

Example 72 Report Example from the report_dft_clock_controller Command

```

*****
Report : Clock controller
Design : des_chip
Version: G-2012.06
Date   : Fri Sep  7 05:18:55 2012
*****

Clock controller: ctrl_0
=====
Number of bits per clock: 4
Controlled clock output pin: duto/clk
=====
Clock generator signal: dutp/PLLCLK
ATE clock signal: i_ateclk
Control pins:
    cycle 0  duto/snps_clk_chain_0/FF_0/Q  1
    cycle 1  duto/snps_clk_chain_0/FF_1/Q  1
    cycle 2  duto/snps_clk_chain_0/FF_2/Q  1
    cycle 3  duto/snps_clk_chain_0/FF_3/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 TestMAX ATPG 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 `-pll_bypass` option of 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`.

To perform DRC of both bypass configurations, PLL active and PLL bypassed, use the following commands:

```
insert_dft

set_dft_drc_configuration -pll_bypass disable ;# already the default
dft_drc

set_dft_drc_configuration -pll_bypass enable
dft_drc

write_test_protocol my_design.spf
```

The test protocol written by the `write_test_protocol` command contains information for PLL bypass as well as for PLL enabled. In the TestMAX ATPG tool, use the `run_drc -patternexec` command to select the operating mode to use.

DFT-Inserted OCC Controller Configurations

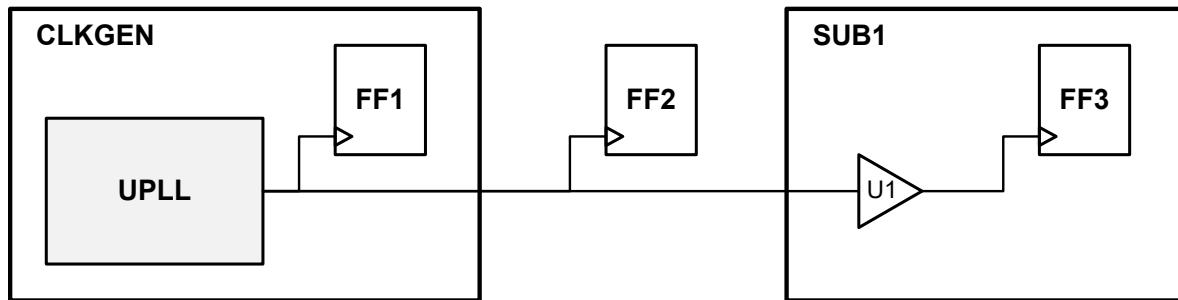
This topic shows DFT-inserted OCC controller configurations and the associated configuration commands, as described in the following topics:

- [Single OCC Controller Configurations](#)
- [Multiple DFT-Inserted OCC Controller Configurations](#)

Single OCC Controller Configurations

This topic shows the results of using various configurations of the `set_dft_clock_controller` command on the design example shown in [Figure 249](#).

Figure 249 Design Example for Single OCC Controller Insertion



The following configuration examples are applied to this design:

- [Example 1 – Controller inserted at the output of UPLL, within the CLKGEN block.](#)
- [Example 2 – Controller inserted at the output of the CLKGEN block.](#)
- [Example 3 – Controller inserted at the output of the buffer.](#)

Example 1

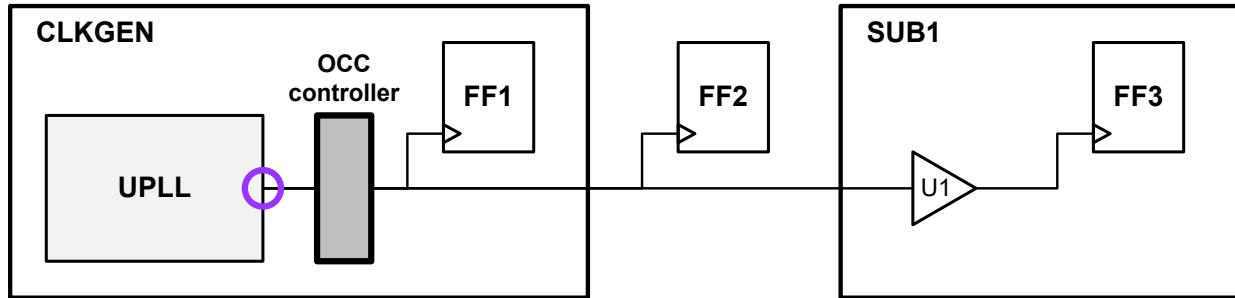
The first example, shown in [Figure 250](#), uses the following configuration:

```
dc_shell> set_dft_clock_controller \
           -pllclocks {CLKGEN/UPLL/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 250 Controller Inserted at Output of PLL Within CLKGEN Block



Example 2

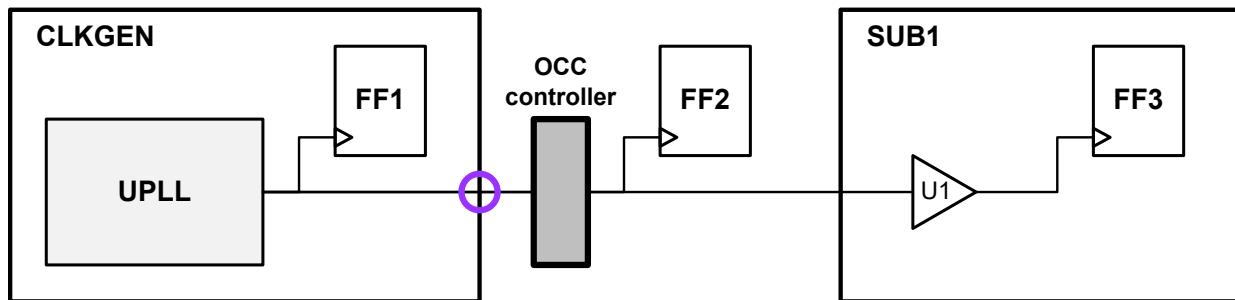
The second example, shown in [Figure 251](#), uses the following configuration:

```
dc_shell> set_dft_clock_controller \
           -pllclocks {CLKGEN/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of the CLKGEN block.
- The FF1 clock remains uncontrollable.

Figure 251 Controller Inserted at Output of CLKGEN Block



Example 3

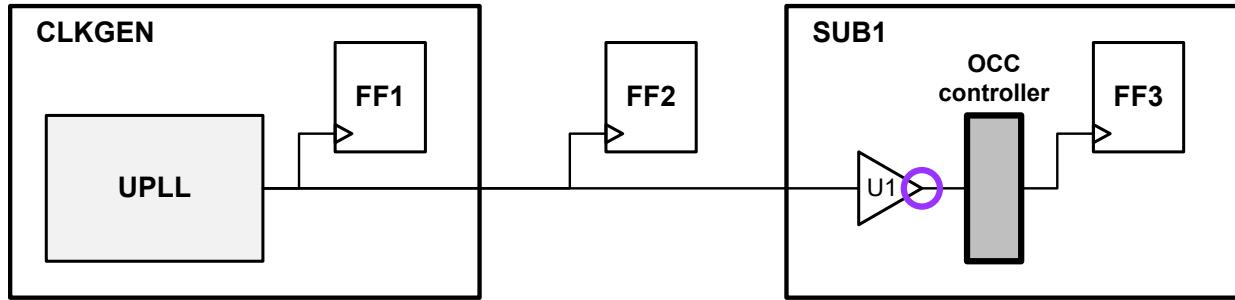
The third example, shown in [Figure 252](#), uses the following configuration:

```
dc_shell> set_dft_clock_controller \
           -pllclocks {SUB1/U1/z}
```

In this case, the following occurs:

- The controller is inserted at the output of buffer U1.
- The FF1 and FF2 clocks remain uncontrollable.

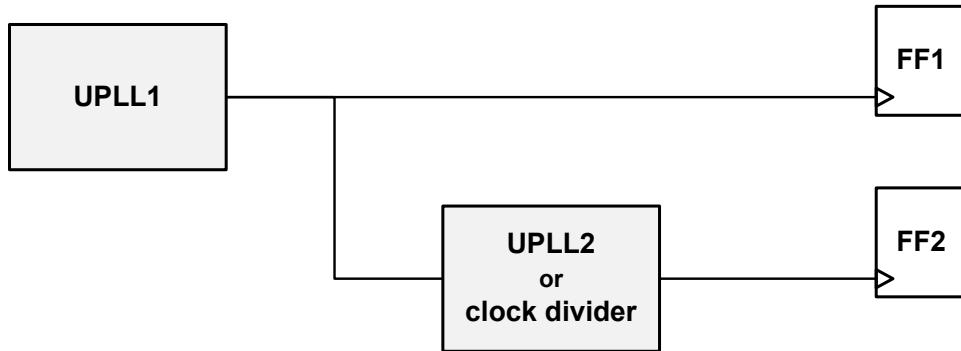
Figure 252 Controller Inserted at Output of the Buffer



Multiple DFT-Inserted OCC Controller Configurations

This topic shows the results of configuring multiple DFT-inserted OCC controllers for the design example shown in [Figure 253](#).

Figure 253 Design Example for Multiple OCC Controller Insertion



When multiple PLLs exist in a design, the reference clock input to each PLL cell must be a free-running clock in test mode. Care must be taken to insure that an OCC controller is not inserted at a location that would block a free-running clock to a downstream PLL cell.

In this design example, the primary PLL named UPLL1 receives the incoming reference clock and generates a PLL output clock. This PLL output clock then feeds either a second PLL or clock divider cell, creating a second cascaded PLL output clock.

The following configuration examples are applied to this design:

- [Example 1](#) – Controller incorrectly inserted, at the output of UPLL1.
- [Example 2](#) – Controller correctly inserted, at the output of a buffer driven by UPLL1.

Example 1

The first example, shown in [Figure 254](#), uses the following configuration:

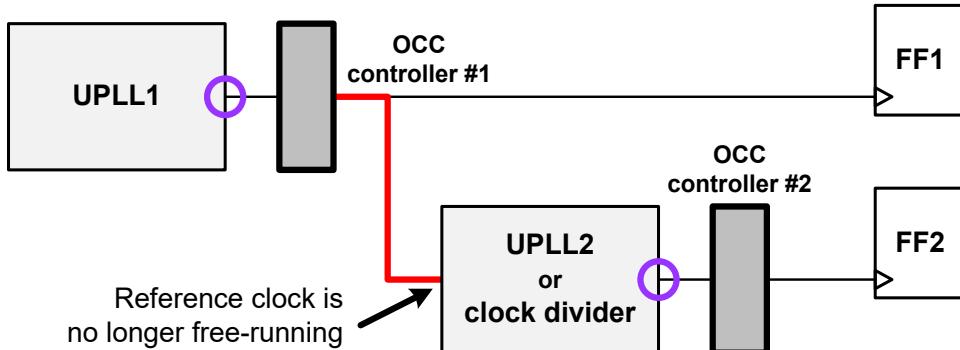
```
dc_shell> set_dft_clock_controller \
           -pllclocks {UPLL1/clkout}
dc_shell> set_dft_clock_controller \
           -pllclocks {UPLL2/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of UPLL1.
- As a result, the free-running clock to UPLL2 is blocked by the first OCC controller, causing incorrect operation of UPLL2.

The incorrect operation of UPLL2 might only be detectable during Verilog simulation of the resulting netlist.

Figure 254 Free-Running Clock Blocked to UPLL2



Example 2

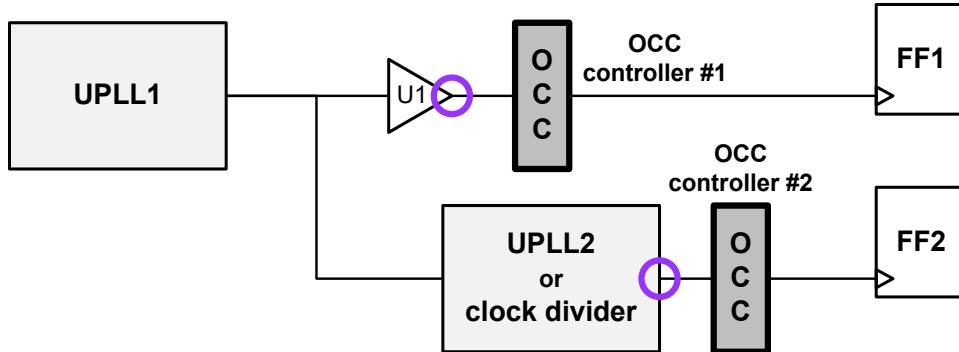
The second example, shown in [Figure 255](#), uses the following configuration:

```
dc_shell> set_dft_clock_controller \
           -pllclocks {U1/z}
dc_shell> set_dft_clock_controller \
           -pllclocks {UPLL2/clkout}
```

This example uses a buffer to isolate the downstream fanout that the first OCC controller should drive. In this case, the following occurs:

- The controller is inserted at the output of buffer U1 driven by UPLL1.
- As a result, the free-running clock from UPLL1 propagates to UPLL2, allowing correct operation of UPLL2.

Figure 255 Free-Running Clock Propagates to UPLL2

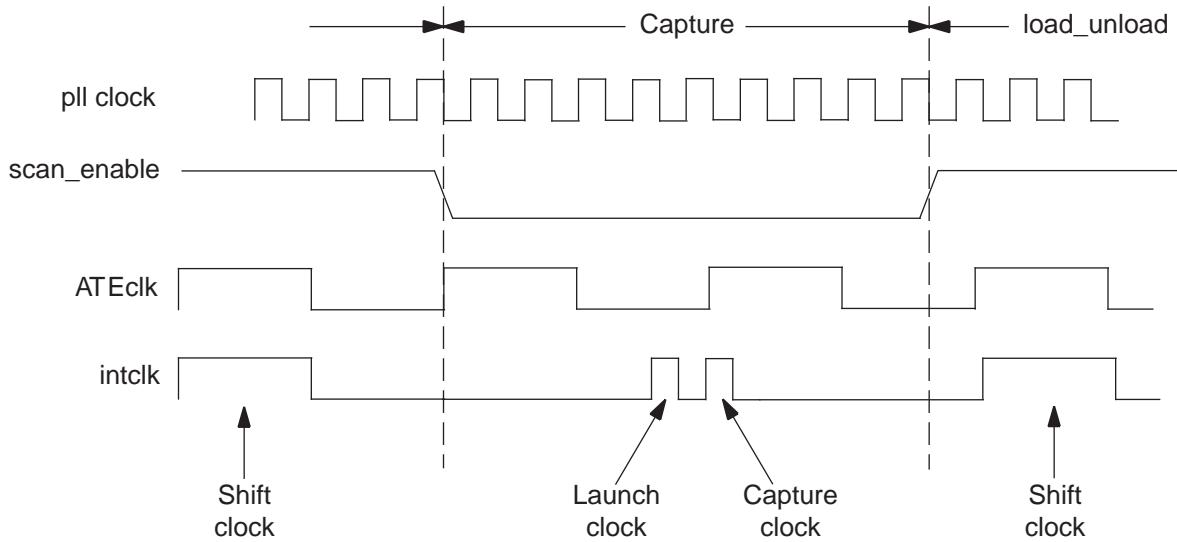


You must ensure that the isolation buffer is not optimized away by applying a `set_dont_touch` command, applying a `set_size_only` command, or using hierarchy. After the clock controller is inserted, the resulting test protocol references the specified pin as a PLL pin.

Waveform and Capture Cycle Example

[Figure 256](#) shows an example of the relationship between various clocks when the design contains an OCC generator and an OCC controller.

Figure 256 Desired Clock Launch Waveform Example



For information about `pll_clock`, `ATEclk`, and `intclk`, see [Clock Type Definitions on page 521](#).

Limitations

Note the following limitations:

- Inferencing internal PLL or any reference clocks is not supported. For pre-DFT DRC, you must explicitly define your reference clock, ATE clock, and PLL clocks.
- You can use differing `-cycles_per_clock` values across OCC controllers, but only if at least one OCC controller is synchronous. For this case, the tool generates the correct netlist and SPF, but post-DFT DRC might fail.
- You cannot mix the DFT-inserted and user-defined OCC controller types in the same DFT insertion run. However, this restriction does not apply to cores that already contain OCC controllers. In integration flows, you can mix OCC controller types across cores and between cores and the top level.
- If you run the `insert_dft` command multiple times to perform DFT operations incrementally, you must perform all OCC operations together with scan insertion in the last `insert_dft` run.

- If you are using synchronous OCC controllers,
 - You cannot use serialized scan compression.
 - Multifrequency capture must be enabled in TestMAX ATPG. To do this, run the `set_drc -fast_multifrequency_capture on` command prior to running DRC.
- The only supported scan style is multiplexed flip-flop.
- 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 with the `-ateclocks` option per controller.
- End-of-cycle measures cannot be used when an OCC controller is used to control the clock.
- If you are using the combinational clock-gating method and synthesis maps the clock selection logic to a MUX cell, you must use the `set_clock_gating_check` command to manually specify a clock-gating check at the MUX gate. For more information, see [Performing Timing Analysis on page 541](#).

For DFT-inserted asynchronous OCC controllers, combinational clock gating is used when the `test_occ_insert_clock_gating_cells` variable is set to its default of `false`. For more information, see [SolvNet article 022490, “Static Timing Analysis Constraints for On-Chip Clocking Support.”](#)

For DFT-inserted synchronous OCC controllers, latch-based clock gating is always used, regardless of the value of the `test_occ_insert_clock_gating_cells` variable.

- If you are using pipelined scan-enable signals and OCC controllers together,
 - The pipelined scan-enable registers must be clocked by the OCC-controlled clock for that clock domain.

However, when you implement a DFT-inserted OCC controller along with a pipelined scan-enable signal in the same run, the tool incorrectly drives the pipeline registers with the uncontrolled PLL clock instead. This must be manually corrected.

Existing (user-defined) OCC controllers are connected properly.

 - The OCC controller must use the unpipelined scan-enable signal, not the pipelined version.

If you are inserting or defining an OCC controller in the current design and plan to implement pipelined scan-enable signals at a higher level, you must create the core in anticipation of the pipelined scan-enable requirements at the higher level.

To do this, ensure that the OCC controller uses a different scan-enable signal than the scan cells. The domain-based scan-enable feature (`set_scan_configuration -domain_based_scan_enable true`) alone does not ensure this, and the `-usage scan` option of the `set_dft_signal` command does not make this distinction.

Then, pass each OCC-controlled clock to an output of the core, then use them for the clock connections of each pipelined scan-enable register that drives the core.

There are no tool options to automate this signal configuration; you must manually ensure these connections.

- When a pipelined scan-enable signal is used with OCC flows, the `insert_dft` command fails to make some connections properly. To use these features together, you must check and correct the connections so that the following requirements are met:
 - The scan-enable connections to the OCC controller and clock chain must use the unpipelined scan-enable signal. That is, use the input to the scan-enable pipeline register instead of its output.
 - The clock connection to the scan-enable pipeline register in OCC controller clock domains must be connected to the internal clock output of the OCC controller block.
 - In hierarchical OCC controller flows, the OCC controller can be inserted or defined at the core level, then scan-enable pipeline registers can be inserted during the top-level integration phase. In this case, the OCC controller is correctly connected at the core level, but the connections are incorrectly made during top-level integration.

To ensure correct operation, you must design the core to provide multiple scan-enable signals. Connect the OCC controller to the unpipelined scan-enable signal, and use the pipelined scan-enable signal for the remaining connections.

To satisfy the requirement that the scan-enable pipeline register must be clocked by the OCC controller's clock output, you must also pass the OCC internal clock to an output of the core, then use it for the clock connection of the top-level scan-enable pipeline register.

- In hierarchical OCC controller flows, if you insert a DFT-inserted OCC controller during integration, the OCC signals of cores containing OCC controllers must be left dangling to be completed by DFT insertion; they cannot be preconnected.
- External (port-driven) clock sources are not supported for existing user-defined OCC controllers.

13

Pre-DFT Test Design Rule Checking

This chapter describes the process for preparing for and running test design rule checking (DRC), and checking violations before DFT insertion.

This chapter includes the following topics:

- [Test DRC Basics](#)
 - [Classifying Sequential Cells](#)
 - [Checking for Modeling Violations](#)
 - [Setting Test Timing Variables](#)
 - [Creating Test Protocols](#)
 - [Masking Capture DRC Violations](#)
-

Test DRC Basics

This topic 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

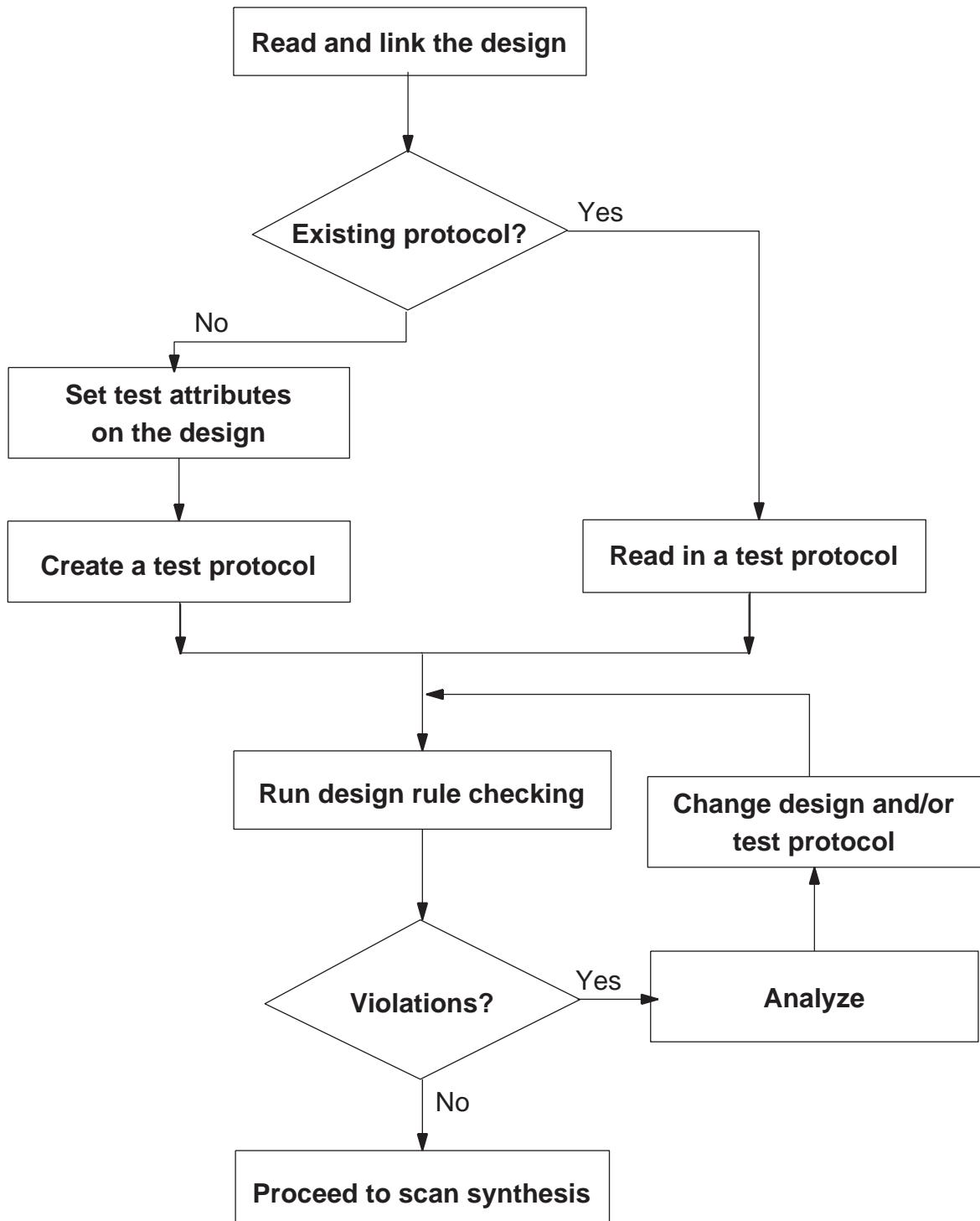
You use the `dft_drc` command to activate test design rule checking. However, before running this process, you must first create a test protocol that includes timing information (see [Creating the Test Protocol on page 568](#) for information on creating test protocols).

After running the `dft_drc` command, violation messages are reported in three categories:

- Information messages – no action is required.
- Warning messages – you should analyze the violations; however, you can still run certain DFT commands.
- Error messages – these indicate serious errors that you need to correct before you can use the DFT commands.

[Figure 257](#) illustrates a general test design rule checking flow.

Figure 257 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 567](#).
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 Design Vision. See [Chapter 7, Running the Test DRC Debugger](#).

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 logic library files referred to by your design.
3. Set the `target_library` variable to point to logic 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 logic library.

For more information about reading in your design, see “Reading Designs” in the Design Compiler User Guide.

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. Define DFT signals on these ports by using the `set_dft_signal` command.
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.

If you run the `insert_dft` command without first running the `dft_drc` command, the tool implicitly runs the `dft_drc` command before proceeding with DFT insertion.

In either case, the following message indicates that test DRC checking is performed:

```
Information: Starting test design rule checking. (TEST-222)
```

In the AutoFix flow, the first DRC analysis determines what test points are needed. The `insert_dft` command inserts the test point logic into the design database, then implicitly

runs the `dft_drc` command again to determine the final DRC results. In this case, you will see an additional TEST-222 message issued during DFT insertion.

Reporting All Violating Instances

By default, the `dft_drc` command generates a message only for the first violating instance of a given violation type. To see all violations, use the `dft_drc -verbose` command.

You cannot perform verbose violation reporting when the `dft_drc` command is implicitly run by the `preview_dft` or `insert_dft` command.

Analyzing and Debugging Violations

You can graphically analyze the cause of a violation by using Design Vision, as described in [Chapter 7, Running the Test DRC Debugger](#).

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 previously described steps to see if the violations have been fixed.

You can also use AutoFix to fix uncontrollable clocks and asynchronous sets and resets.

See Also

- [Using AutoFix on page 333](#) for more information about using AutoFix to fix design testability issues

Summary of Violations

At the completion of design rule checking, the `dft_drc` command displays a violation summary. [Example 73](#) shows the format of the violation summary.

Example 73 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 unknown cells:

```
-----  
DRC Report  
Total violations: 4  
-----
```

```
3 MODELING VIOLATIONS
1 Cell has unknown model violation (TEST-451)
```

Enhanced Reporting Capability

You can enable enhanced DRC reporting by setting the `test_disable_enhanced_dft_drc_reporting` variable to `false`. When enhanced reporting is enabled, the reporting and formatting of rule violations are changed to provide a better understanding of the respective rules.

[Example 74](#) provides a typical enhanced DRC report:

Example 74 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 the `dft_drc` command, 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.

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

When violations occur, the `dft_drc` command issues the following message:

Warning: Violations occurred during test design rule checking. (TEST-124)

For designs that are synthesized with the `compile -scan` command, the default behavior is that violations on scan-replaced cells cause the `insert_dft` command to unscan those cells. Sequential cells with violations are not included in a scan chain because they would probably prevent the scan chain from working as intended.

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 for those cells.

For certain violation types, you can configure DFT insertion to include violating sequential cells in scan chains. See [Masking Capture DRC Violations on page 600](#).

Viewing the Sequential Cell Summary

When the `dft_drc` command completes DRC, it provides a summary of the test status of the sequential cells in your design. [Example 75](#) shows an example of the summary.

Example 75 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 76](#) shows the syntax of the sequential cell summary.

Example 76 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 topics.

Sequential Cells With Violations

This topic of the sequential cell summary points to problematic sequential cells. The cells in this group have corresponding violations that can be found in the DRC output of the `dft_drc` command.

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.

After `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 covered in the following topics:

- [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 logic library. Such cells are marked as black-box by 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:

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

You do not need to correct black-box violations for memory macro cells; they are always modeled as black-box cells by the `dft_drc` command. In TestMAX ATPG, you can use memory models so that sequential ATPG can obtain fault coverage around the memories.

For more information about modeling the behavior of cells, see Library Compiler documentation.

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 logic libraries or designs in the `search_path` (unresolved reference).
- The logic 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 warning 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 logic library, you need to obtain either a functional model or a structural model of the cell.

If the cell can be functionally modeled by the Library Compiler tool, obtain an updated logic 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:

```
dc_shell> set_app_var test_simulation_library simulation_library_path
```

Note the following license-related requirements:

- 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 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 the Liberty 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 synthesis 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 User Guide*.

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 the Design Compiler or DFT Compiler tools. For more information about modeling sequential cells, see the Library Compiler documentation.

Generic Cells

Your design should be a mapped netlist. In the RTL stage, the `dft_drc` command 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 warning 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 or it has the `dont_use` attribute applied, the `dft_drc` command issues the following warning 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 information 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 nonscan cell before scan cell replacement, that cell is not modified or scan-replaced when you optimize the design. In this case, the `dft_drc` command produces the following warning:

```
Warning: Cell %s (%s) can't be made scannable because it is
dont_touched. (TEST-121)
```

If you apply the `dont_touch` attribute to scan-replaced cell, the cell can still be added to a scan chain.

Note:

Use the `dont_touch` attribute carefully, because it can increase the number of nonscan cells, and nonscan cells lower fault coverage.

Use the `set_scan_element false` command 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:

The `dft_drc` command allows synchronous elements to be on the scan chain.

Setting Test Timing Variables

This topic discusses the process for setting test timing variables for your design. The timing variables are used by the test protocol for design rule checking and for DFT preview and insertion.

This topic covers the following:

- [Protocols for Common Design Timing Requirements](#)
- [Setting Timing Variables](#)

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 variables and, if necessary, by defining test clock requirements. Timing variables are discussed in detail in [Setting Timing Variables on page 582](#).

Defining test clock requirements is discussed in detail in [Chapter 9, Architecting Your Test Design](#).”

If your design’s timing variable values are the same as the variables’ defaults, you do not need to make any changes.

Preclock Measure Protocol

To use a preclock measure protocol, use the default test timing variable values, which are as follows:

```
test_default_period :      100 ;
test_default_delay :      0 ;
test_default_bidir_delay : 0 ;
test_default_strobe :      40 ;
test_default_strobe_width : 0 ;
```

This configuration places the measure strobe before the default clock pulse. If you use a nondefault clock waveform, adjust the strobe value accordingly. Check with your semiconductor vendor for specific timing information.

End-of-Cycle Measure Protocol

To use an end-of-cycle measure protocol, set the test timing variables as follows:

```
test_default_period :      100 ;
test_default_delay :      0 ;
test_default_bidir_delay : 0 ;
test_default_strobe :      95 ;
test_default_strobe_width : 0 ;
```

This configuration places the measure strobe after the default clock pulse. Although the end-of-cycle measure protocol works with TestMAX ATPG, the default preclock measure protocol is more efficient.

The end-of-cycle measure protocol cannot be used with

- Clocks controlled by OCC controllers
- DFTMAX high X-tolerance scan compression (with or without serializer)
- DFTMAX Ultra scan compression

Setting Timing Variables

Before you run the `create_test_protocol` command, you need to define timing variables. The command uses the following test variables to determine the values in the test protocol timing variables:

```
test_default_period
test_default_delay
test_default_bidir_delay
test_default_strobe
test_default_strobe_width
```

The requirements from your semiconductor vendor, 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 Variable

The `test_default_period` variable defines the default, 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_app_var test_default_period period
```

For example,

```
dc_shell> set_app_var test_default_period 100
```

In the .synopsys_dc.setup file, the `test_default_period` variable is set to 100 ns.

test_default_delay Variable

The `test_default_delay` variable defines the default, 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 258 on page 587](#).

By default, DFT Compiler applies data to all nonclock input ports 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_app_var test_default_delay delay
```

For example,

```
dc_shell> set_app_var test_default_delay 5
```

In the .synopsys_dc.setup file, `test_default_delay` is 0 ns.

test_default_bidir_delay Variable

The `test_default_bidir_delay` variable defines the default, 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 258 on page 587](#).

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, 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_app_var test_default_bidir_delay bidir_delay
```

For example,

```
dc_shell> set_app_var test_default_bidir_delay 40
```

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, 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 258 on page 587](#)).

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_app_var test_default_strobe strobe
```

For example:

```
dc_shell> set_app_var test_default_strobe 100
```

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, 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 258 on page 587](#).

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 49](#) provides the default clock timing for each clock type.

Table 49 Default Clock Timing for Each Clock Type

Clock type	First edge	Second edge
Edge-triggered or D-latch enable	45	55

Clock type	First edge	Second edge
Master clock	30	40
Slave clock	60	70
Edge-triggered	45	60
Master clock1	50	60
Slave clock	40	70

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 logic 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_app_var test_default_strobe_width strobe_width
```

If you need a window strobe in your STIL protocol file (SPF) or STIL patterns, set the default of `test_default_strobe_width` to 1 ns, as shown in the following command:

```
dc_shell> set_app_var test_default_strobe_width 1
```

In the `.synopsys_dc.setup` file, `test_default_strobe_width` is 0 ns.

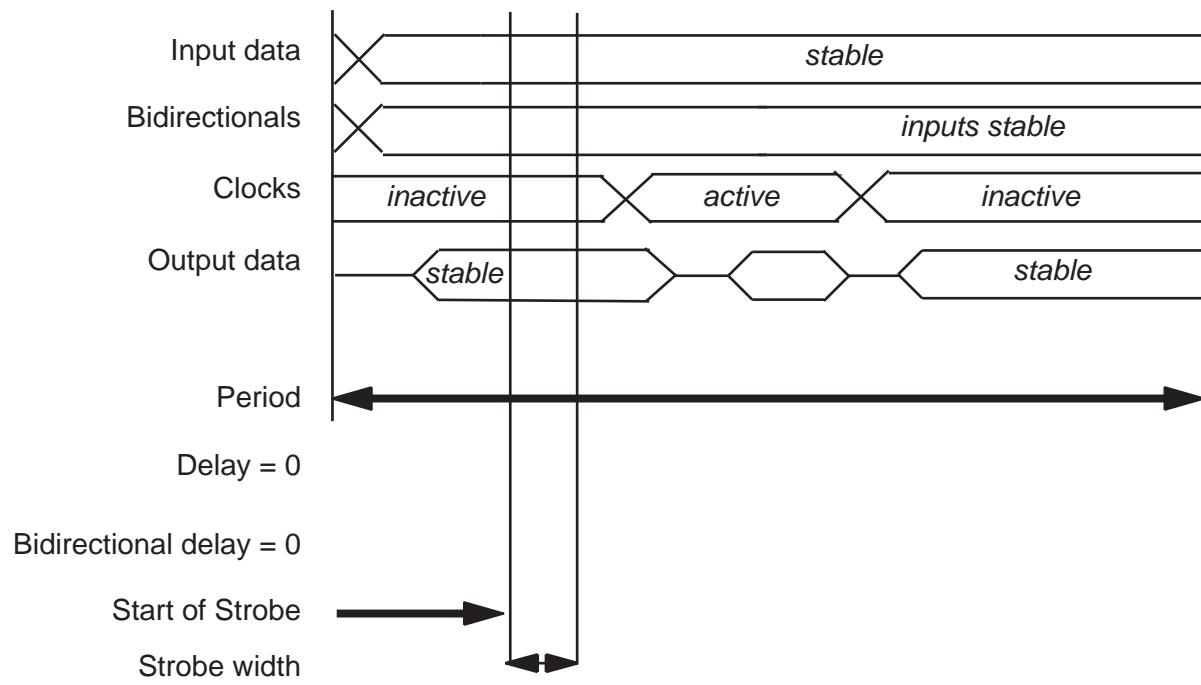
Note:

When `test_default_strobe_width` is 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 Variables on Vector Formatting

Figure 258 shows a timing diagram for a strobe-before-clock scheme.

Figure 258 Effect of Timing Variables 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 the `dft_drc` command. This topic covers the following topics related to creating test protocols:

- [Design Characteristics for Test Protocols](#)
- [STIL Test Protocol File Syntax](#)
- [Defining an Initialization Protocol](#)
- [Scan Shift and Parallel Measure Cycles](#)
- [Examining a Test Protocol File](#)

Design Characteristics for Test Protocols

A test protocol is based on certain characteristics of a design. The following topics discuss how a protocol is affected by these design characteristics:

- Scan Style
- New DFT Signals
- Existing Clock Ports
- Existing Asynchronous Control Ports
- Bidirectional Ports

Scan Style

Each scan style has a unique method of performing scan shift, which must be reflected in the test protocol.

See Also

- [Scan Shift and Parallel Measure Cycles on page 595](#) for more information about how scan style influences the scan shift process

New DFT Signals

The DFT signal attributes are set automatically for each new test port created by the `insert_dft` command. The DFT signal attributes are 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 reidentify each test port with the appropriate `set_dft_signal` command.

Existing Clock Ports

You specify existing 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` command.

Existing Asynchronous Control Ports

You specify existing asynchronous control ports by using the following command:

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

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 the `test_default_bidir_delay` variable 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 TestMAX ATPG.

Although the STIL protocol file syntax is the same as that used by TestMAX ATPG, DFT Compiler cannot read some of the STIL elements that are available in TestMAX 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 the DFT Compiler and TestMAX ATPG tools use the IEEE P1450.1 extensions to STIL. For details, see Appendix E, “STIL IEEE P1450.1 Extensions,” in TestMAX ATPG and TestMAX Diagnosis Online Help.

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 `test_setup` macro example is shown in [Example 77](#).

Example 77 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 defaults.

[Example 78](#) contains many additions to define signal timing. Line numbers have been added for reference. Note:

- Lines 6–9. Defines 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. Defines a waveform table with a period of 1,000 ns that defines the timing to be used during nonshift cycles.
- Line 37. Adds the W statement to ensure 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 78 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]"' ;

```

Chapter 13: Pre-DFT Test Design Rule Checking
 Creating Test Protocols

```

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; } }
16.             // clock
17.             CLOCK { 01Z { '0ns' D/U/Z; } }
18.             RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
19.             // async reset
20.             RESETB { 01Z { '0ns' D/U/Z; } }
21.             input_grp1 { 01Z { '0ns' D/U/Z; } }
22.             input_grp2 { 01Z { '10ns' D/U/Z; } }
23.             // outputs are to be measured at t=350
24.             out_ports { HLT { '0ns' X; '350ns' H/L/T/X; } }
25.             // bidirectional ports as inputs are forced at t=20
26.             bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
27.             // bidirectional ports as outputs are measured at
28.             // t=350
29.             bidi_ports { X { '0ns' X; } }
30.             bidi_ports { HLT { '0ns' X; '350ns' H/L/T; } }
31.         }
32.     } // end BROADSIDE_TIMING
33. }
34. ScanStructures {
35.     ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
36.     ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
37.     ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
38.     ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
39. } // end scan structures
40. Procedures {
41.     "load_unload" {
42.         W "BROADSIDE_TIMING" ;
43.         V {CLOCK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1;
44.             bidi_ports = \r16 Z;}
45.         V {}
46.         V { bidi_ports = \r4 1010 ; }
47.         Shift {
48.             V { _si=####; _so=####; CLOCK=P; }
49.         }
50.     } // end load_unload
51. }
52. MacroDef {
53.     "test_setup" {
54.         W "BROADSIDE_TIMING" ;
55.         V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
56.             BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZZ; }
57.         V {PLL_RESET = 0; }
58.         V {PLL_RESET = 1; }
59.     } // end test_setup
60. }
61. 
```

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 79](#) shows the syntax used to define scan chains.

Example 79 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 80](#) defines the shift procedure.

Example 80 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, then read the initialization protocol using the following command.

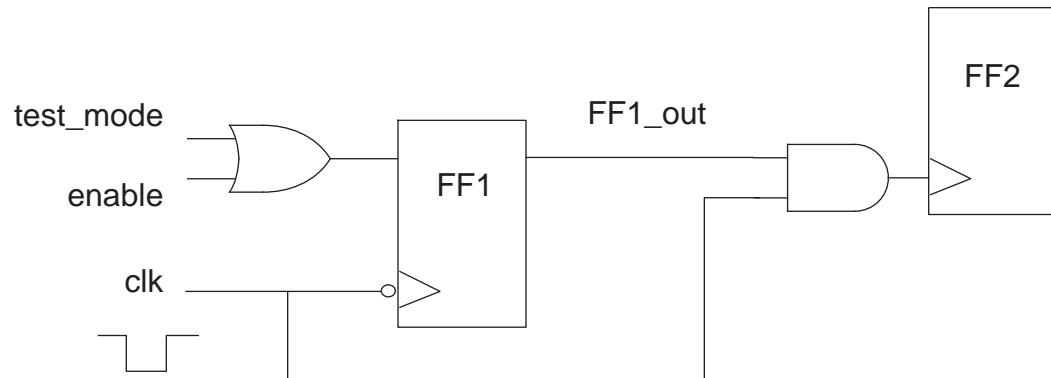
```
read_test_protocol -section test_setup
```

5. Rerun `create_test_protocol` to complete the test protocol.

Run test DRC.

See the design in [Figure 259](#) for an illustration of the use of an initialization protocol.

Figure 259 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 one time 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 as follows:

```
"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 the `insert_dft` command completes, this initialization sequence is lost. You must reapply the same initialization sequence to ensure that post-DFT test DRC reports no violations.

[Table 49](#) 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 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 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 Measure Cycles

The standard strobe-before-clock protocol shifts all scan chains simultaneously. This protocol allows scan shift output for the current pattern and scan shift input for the next pattern to overlap. If scan groups are used, not all scan chains are required to shift simultaneously. For more information about scan groups, see [Creating Scan Groups on page 432](#).

The process DFT Compiler uses to perform scan shift is determined by the scan style you selected with the `set_scan_configuration -style` command.

For all scan styles, 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.

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. Compare scan data at the scan output ports.
4. Pulse the system clocks.

System clock ports are identified with the `set_dft_signal` command as `ScanClock` signals.

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.

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. Compare scan data at the scan output ports.
3. Pulse the scan clock ports.

Scan clock ports are identified with the `set_dft_signal` command as `ScanMasterClock` signals.

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. Compare scan data at the scan output ports.
3. Pulse the test master clock, then pulse the slave clock.

Test master and slave clock ports are identified with the `set_dft_signal` command as `ScanMasterClock` and `ScanSlaveClock` signals, respectively.

Scan-Enabled LSSD Scan Style

For the scan-enabled 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. Assert the scan-enable signals.
2. Apply scan data at the scan input ports.
3. Compare scan data at the scan output ports.
4. Pulse the test master clock, then pulse the slave clock.

Test master clock ports are identified with the `set_dft_signal` command as `ScanMasterClock` signals. System clock ports, which are repurposed as slave test clock ports in test mode, are also identified as `ScanMasterClock` signals but must have slave test clock timing waveforms defined.

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 as follows:

```
write_test_protocol [-output test_protocol_file_name]
                    [-test_mode mode_name]
                    [-names verilog | verilog_single_bit]
```

Option	Description
<code>-output test_protocol_file_name</code>	Specifies the name of the ASCII output file. The default file name is <code>design_name.spf</code> , where <code>design_name</code> is the current design, and the <code>.spf</code> extension identifies the file type as a STIL format test protocol file.
<code>-test_mode mode_name</code>	Specifies the CTL model test mode from which the protocol is generated.

Option	Description
-names verilog verilog_single_bit	Specifies the form of the names used in the STIL protocol file. 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 verilogout_single_bit environment variable. In all cases, the internal representation is not changed. This option takes effect only in conjunction with -test_mode options, when HSS is used. In all other cases, the form of the names is determined by the setting of the test_stil_netlist_format 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 81 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 81 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 Capture DRC Violations

By default, DFT Compiler excludes sequential cells with DRC violations from scan chains. In some cases, such as spare cells that have constant data inputs, you can include violating cells in scan chains.

DFT Compiler allows you to mask cells with certain capture DRC violation types, as described in the following topics:

- [Configuring Capture DRC Violation Masking](#)
- [Reporting Capture DRC Violation Masking](#)
- [Resetting Capture DRC Violation Masking](#)

Configuring Capture DRC Violation Masking

You can mask the following capture DRC violation types during pre-DFT DRC:

- TEST-504 – Cell always captures constant zero value
- TEST-505 – Cell always captures constant one value
- D17 – Cell has a clock, set, or reset input pin that cannot capture data

When a violation type is masked, violating cells are included in scan chains. To mask a violation type, use the `set_dft_drc_rules` command. The syntax is

```
set_dft_drc_rules
  [-allow drc_list]
  [-ignore drc_list]
  [-cell cell_list]
```

The `-allow` and `-ignore` options both allow you to specify one or more violation types to mask. The difference is as follows:

- `-allow` – DFT allows violating cells to be included in scan chains, but the violations are still reported
- `-ignore` – DFT completely ignores the violating cells; the violating cells are included in scan chains and the violations are not reported

For example, the following command includes constant-capturing sequential cells in scan chains (with warnings issued during pre-DFT DRC):

```
dc_shell> set_dft_drc_rules -allow {TEST-504 TEST-505}
```

By default, the specification applies globally to the entire design. To limit the specification to certain cells, use the `-cell` option. For example, the following command includes specific noncapturing sequential cells in scan chains (with no warnings):

```
dc_shell> set_dft_drc_rules -ignore {D17} \
           -cell [get_object_name [get_cells {CONFIG_reg[*]}]]
```

You can issue multiple `set_dft_drc_rules` commands. The tool applies all command specifications cumulatively. Cell-specific specifications take precedence over global specifications.

Reporting Capture DRC Violation Masking

You can use the `report_dft_drc_rules` command to report masking specifications previously applied with the `set_dft_drc_rules` command. The syntax is

```
report_dft_drc_rules
  [-violation_drc_list]
  [-cell cell_list]
```

By default, all previously applied command specifications are reported. For example,

```
dc_shell> report_dft_drc_rules
```

Violation Name	Default Action	Specified Action	Range/Cell list
<hr/>			
TEST-504	omit	allow	all cells
TEST-505	omit	allow	BLK1
	omit	allow	BLK2
	omit	ignore	USPAREGATES

You can use the `-violation` option to restrict the report to certain violation types. For example,

```
dc_shell> report_dft_drc_rules -violation {TEST-504}
```

Violation Name	Default Action	Specified Action	Range/Cell list
<hr/>			
TEST-504	omit	allow	all cells

You can use the `-cell` option to restrict the report to certain cell-specific command specifications. For example,

```
dc_shell> report_dft_drc_rules -cell {BLK1 BLK2}
```

Violation Name	Default Action	Specified Action	Range/Cell list
<hr/>			
TEST-505	omit	allow	BLK1
	omit	allow	BLK2

Resetting Capture DRC Violation Masking

You can use the `reset_dft_drc_rules` command to remove masking specifications previously applied with the `set_dft_drc_rules` command. The syntax is

```
reset_dft_drc_rules
[-violation drc_list]
[-cell cell_list]
```

By default, all previously applied command specifications are removed. For example,

```
dc_shell> reset_dft_drc_rules
```

You can use the `-violation` option to remove only specifications for certain violation types. For example,

```
dc_shell> reset_dft_drc_rules -violation {TEST-504 TEST-505}
```

You can use the `-cell` option to remove only certain cell-specific command specifications. For example,

```
dc_shell> reset_dft_drc_rules -cell {BLK1 BLK2}
```

14

Previewing, Inserting, and Checking DFT Logic

This chapter describes how to preview your DFT design, insert the DFT logic, and check the inserted DFT logic for correct operation.

This chapter includes the following topics:

- [Previewing the DFT Logic](#)
 - [Inserting the DFT Logic](#)
 - [Post-DFT Insertion Test Design Rule Checking](#)
-

Previewing the DFT Logic

Use the `preview_dft` command to preview your scan design. The `preview_dft` command runs the same scan architecture algorithms as the `insert_dft` command, except that it reports the scan architecture to be implemented instead of actually implementing it. This allows you to preview your scan chains and DFT logic without synthesizing them and to change your specifications to explore the design space as necessary.

You should always use the `preview_dft` command to generate a DFT preview report for your design. Although you can also use the `report_scan_path -view existing_dft` command after DFT insertion to report the implemented scan chains, the `preview_dft` report includes additional DFT architecture information such as test points, test signals, DFT-inserted cores, and scan compression.

The following topics describe usage of the `preview_dft` command:

- [Running the preview_dft Command](#)
- [Previewing Additional Scan Chain Information](#)
- [Previewing Test Mode Information](#)
- [Previewing the DFT Design Using Script Commands](#)

For more information about using the `-test_wrappers` option to preview core wrapper chains, see [Previewing the Wrapper Cells on page 487](#) and [Previewing Maximized Reuse Wrapper Cells on page 488](#).

Running the preview_dft Command

Before running the `preview_dft` command, a valid test protocol must exist in memory. You can create a test protocol with the `create_test_protocol` command, or you can read an existing test protocol in with the `read_test_protocol` command. For details, see [Creating the Test Protocol on page 568](#).

The `preview_dft` command also requires that pre-DFT DRC be run. If you run the `preview_dft` command without first running the `dft_drc` command, the tool implicitly runs the `dft_drc` command before proceeding with DFT preview.

When these requirements have been met, you can generate the DFT preview report:

```
dc_shell> preview_dft
```

[Example 82](#) shows a simple DFT preview report example.

Example 82 Preview Report Generated by the preview_dft Command

```
*****
Preview_dft report
For      : 'Insert_dft' command
Design   : top
Version  : I-2013.12-SP3
Date     : Mon May 12 09:16:49 2014
*****

Number of chains: 2
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks

Scan chain '1' (MY_SI1 --> MY_SO1) contains 6 cells

Scan chain '2' (MY_SI2 --> MY_SO2) contains 6 cells
```

Previewing Additional Scan Chain Information

To show additional information about the scan chains, use the `-show` option of the `preview_dft` command:

```
dc_shell> preview_dft -show {...}
```

This option accepts a list of keywords that cause additional types of information to be included in the preview report. Valid keywords (with corresponding report examples) are:

- **cells** – Shows all scan cells and scan segments in each scan chain:

```
Scan chain '1' (test_si1 --> test_so1) contains 6 cells:
```

```
Z1F_reg[0]
Z1F_reg[1]
Z1F_reg[2] (1)
Z2F_reg[0]
Z2F_reg[1]
Z2F_reg[2]
```

- **scan_clocks** – Shows scan clock domains along the scan chains:

```
Scan chain '1' (test_si1 --> test_so1) contains 6 cells:
```

```
Z1F_reg[0] (CLK1, 55.0, falling)
...
Z2F_reg[0] (CLK2, 55.0, falling)
...
Z2F_reg[2]
```

- **scan_signals** – Shows information about DFT signals and hookup pins associated with each scan chain:

```
Scan chain '1' (MY_SI1 --> MY_SO1) contains 6 cells
```

```
Scan signals:
test_scan_in: MY_SI1 (no hookup pin)
test_scan_out: MY_SO1 (no hookup pin)
```

- **segments** – Shows information about scan segments included in the scan chains:

```
Core scan segment 'core/1' (core/test_si --> core/test_so) contains 3 cells:
```

```
core/Z2R_reg[0]
core/Z2R_reg[1]
core/Z2R_reg[2]
```

```
Other access pins:
core/test_se (test_scan_enable)
core/CLK2 (test_scan_clock)
```

Also include the `cells` keyword to see the scan cells contained in each segment.

Scan segments result from

- Scan chains within CTL-modeled cores
- Identified shift registers
- DFT-inserted and user-defined clock chains

- set_scan_group -serial_routed true specifications
- Multibit components, when the -preserve_multibit_segment option of the set_scan_configuration command is set to true
- qgates – Shows toggle suppression gates implemented along the scan chains due to the set_scan_suppress_toggling command:

(g) shows cell scan-out drives a toggle suppressing gate

Scan chain '1' (test_si1 --> test_so1) contains 3 cells:

```
Z1F_reg[0]
Z1F_reg[1]
Z1F_reg[2] (g)
```

The qgates keyword implicitly includes the scan keyword.

- voltages – Shows scan cell operating voltage information along the scan chains
 - (i) shows cell scan-out drives an isolation cell
 - (v) shows cell scan-out drives a level shifter cell

Scan chain '1' (test_si1 --> test_so1) contains 4 cells:

```
Z1F_reg[0] (voltage 1.08)
Z1F_reg[1]
Z2F_reg[0] (voltage 0.80)
Z2F_reg[1]
```

- power_domains – Shows power domains along the scan chains

(i) shows cell scan-out drives an isolation cell
 (v) shows cell scan-out drives a level shifter cell

Scan chain '1' (test_si1 --> test_so1) contains 4 cells:

```
Z1F_reg[0] (pwr domain 'pd_2')
Z1F_reg[1]
Z2F_reg[0] (pwr domain 'pd_1')
Z2F_reg[1]
```

- bidirectionals – Shows information about bidirectional conditioning logic used to enable scan paths:

Bidirectional Port	Specified Conditioning	Resolved Conditioning
BIDI[0]	Input	Input
BIDI[1]	Input	Output
BIDI[2]	Input	Input

- `tristates` – Shows information about all tristate conditioning logic used to prevent tristate contention during scan shift

Tristate net	Specified Disabling	Resolved Disabling
Z[0]	disable_all	disable_all
Z[1]	disable_all	disable_all
Z[2]	disable_all	disable_all

- `scan` – Shows scan cells that explicitly have the `scan_element` attribute set to `true`:

(t) shows cell has a true scan attribute

Scan chain '1' (MY_SI1 --> MY_SO1) contains 12 cells

core/Z2R_reg[1] (t)

This attribute is typically applied with the `set_scan_element true` command. If the `cells` keyword is not also specified, only scan cells with the `scan_element` attribute set to `true` are shown.

- `scan_summary` – Shows a short summary of the scan chains and their scan clocks:

Chain	Scan Ports	# Cells	Inst/Chain	Clock (port, time, edge)
S 1	MY_SI1 --> MY_SO1	6	Z1F_reg[0] Z2F_reg[0]	(CLK1, 55.0, falling) (CLK2, 55.0, falling)
S 2	MY_SI2 --> MY_SO2	6	Z1R_reg[0] Z2R_reg[0]	(CLK1, 45.0, rising) (CLK2, 45.0, rising)

Use this keyword by itself.

- `all` – Show all information about scan chains (equivalent to specifying all keywords except `scan_summary`)

The preview report format adapts to the keywords you specify. For example, with the `-show {scan_clocks}` option, the report shows only the scan cells at scan clock transitions along the chain, with other cells represented by an ellipsis ("..."). With the `-show {scan_clocks cells}` option, all scan cells are shown along with the scan clock transitions.

The preview report uses attributes to show where certain scan structures exist along the scan chains. For example,

```
dc_shell> preview_dft -show {cells segments}
...
(l) shows cell scan-out drives a lockup latch
(s) shows cell is a scan segment
(m) shows cell scan-out drives a multi-mode multiplexer
(L) shows test retiming flop
```

- (t) shows cell has a true scan attribute
- (w) shows cell scan-out drives a wire

```
Scan chain 'MY_CHAIN' (SI --> SO) contains 4 cells
Active in modes: Internal_scan :

core1/CLK_CHAIN (s) (m) (CLK, 55.0, falling)
(l) core2/CLK_CHAIN (s) (m) (CLK, 55.0, falling)
```

Some less common attributes, such as retiming flip-flops, are shown in the legend only when used in the report.

For more information about the keywords used with the `-show` option, see `preview_dft` man page.

Previewing Test Mode Information

If you have multiple test modes in your design, the DFT logic uses test-mode signals to select the test mode. In this case, the `preview_dft` command does the following:

- It reports the scan chain structures for each test mode.
- It reports the test-mode signals and encodings to be used for test mode selection.

[Example 83](#) shows a preview report for a design with two test modes.

Example 83 Preview Report Section Describing Multiple Test Modes

```
*****
Current mode: SHORT
*****

Number of chains: 3
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks

Scan chain '1' (test_si1 --> test_so1) contains 4 cells
Active in modes: SHORT

Scan chain '2' (test_si2 --> test_so2) contains 4 cells
Active in modes: SHORT

Scan chain '3' (test_si3 --> test_so3) contains 4 cells
Active in modes: SHORT

*****
Current mode: LONG
*****
```

```
Number of chains: 1
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks

Scan chain '1' (test_sil --> test_sol) contains 12 cells
    Active in modes: LONG
```

```
=====
Test Mode Controller Information
=====
```

Test Mode Controller Ports

```
-----
test_mode: test_mode2
test_mode: test_mode1
```

Test Mode Controller Index (MSB --> LSB)

```
-----
test_mode2, test_mode1
```

Control signal value - Test Mode

```
-----
01 SHORT - InternalTest
10 LONG - InternalTest
```

You can create multiple test modes with the `define_test_mode` command. Compressed scan designs always have multiple test modes (at least one standard scan mode and one compressed scan mode).

See Also

- [Multiple Test Modes on page 357](#) for more information about defining test modes

Previewing the DFT Design Using Script Commands

To gain more understanding of the test structures to be built, you can use following command to preview the DFT design using DFT configuration commands:

```
dc_shell> preview_dft -script
```

The result is a set of DFT commands (such as `define_test_mode`, `set_scan_path`, and `set_test_point_element`) that describe the DFT structures to be built.

Note the following:

- The command output is intended to help understand the DFT design; it is not intended to be sourced directly as a configuration script in a subsequent run.
- You can modify and apply the `set_scan_path` commands to implement your own scan cell order. However, keep in mind that the `-ordered_elements` option prevents the specified scan cells from being reordered or repartitioned using SCANDEF information.

Inserting the DFT Logic

After configuring and previewing your design, assemble the scan chains by using the `insert_dft` command:

```
dc_shell> insert_dft
```

This following topics describe how the `preview_dft` and `insert_dft` commands generate a scanned design:

- [Scan Replacement](#)
- [Scan Element Allocation and Ordering](#)
- [Test Signals](#)
- [Pad Cells](#)

Scan Replacement

Scan replacement is the process of remapping nonscan sequential cells to library cells have appropriate test pins for the chosen scan style.

DFT Compiler performs the following scan replacement tasks during the `insert_dft` command:

- Scan-replaces sequential elements if a scan replacement on the sequential elements was not performed previously, and the cell does not violate test DRC.

The `set_scan_configuration -replace false` setting disables this behavior. For more information, see [Scan Stitching Only Scan-Replaced Cells on page 217](#).

- Converts the scan elements that resulted from a test-ready compile or a previous scan insertion back to nonscan elements if test DRC violations prevent their inclusion in a scan chain, and the `set_dft_insertion_configuration -unscan true` command has been issued.

Scan Element Allocation and Ordering

DFT Compiler allocates and orders 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 previously built by the `insert_dft` command or subdesign scan chains built by the `insert_dft` command, even if their routing does not conform to default behavior.
- In Design Compiler wire load mode, allocates and orders scan elements into scan chains alphanumerically, using the full hierarchical path specification of the scan element name.
- In Design Compiler topographical mode, allocates and orders scan elements into scan chains using virtual layout information, which reduces scan routing overhead. In this case, the `preview_dft` and `insert_dft` commands issue the following message to indicate that topographical information is used:

Running DFT insertion in topographical mode.

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

By default, AutoFix is enabled for bidirectional and three-state pad cells. If the current design includes such pad cells with functional models in the logic library, the `insert_dft` command inserts DFT testability logic for 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

See Also

- [Configuring Three-State Bus AutoFixing on page 342](#) for information on AutoFixing three-state output drivers
- [Configuring Bidirectional AutoFixing on page 342](#) for information on AutoFixing bidirectional pad cells

Post-DFT Insertion Test Design Rule Checking

After you perform scan insertion, you can run the `dft_drc` command to perform design rule checking of the DFT-inserted to ensure that no violations have been introduced into your design by the scan insertion process. This is called *post-DFT DRC*.

This topic covers the following topics related to post-DFT DRC:

- [Running Post-DFT DRC After DFT 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
- Post-DFT DRC Limitations

Running Post-DFT DRC After DFT Insertion

You can perform post-DFT DRC after the `insert_dft` command completes successfully. Use the `current_test_mode` command to change the focus to each test mode of interest, then run the `dft_drc` command. For example,

```
dc_shell> insert_dft
...
dc_shell> current_test_mode wrp_if
dc_shell> dft_drc
...
dc_shell> current_test_mode wrp_of
dc_shell> dft_drc
...
dc_shell> current_test_mode ScanCompression_mode
dc_shell> dft_drc
...
```

At the beginning of its output, the `dft_drc` command issues a message confirming that post-DFT DRC is being run:

```
dc_shell> dft_drc
In mode: Internal_scan...
Design has scan chains in this mode
Design is scan routed
Post-DFT DRC enabled
...
```

If your design contains only the default `Internal_scan` test mode, you do not need to set the current test mode; the `Internal_scan` mode is the default.

Note:

Some features and flows do not support post-DFT DRC, as noted in [Post-DFT DRC Limitations on page 627](#). In such cases, use DRC in the TestMAX ATPG tool to validate the DFT-inserted design.

Checking for Topological Violations

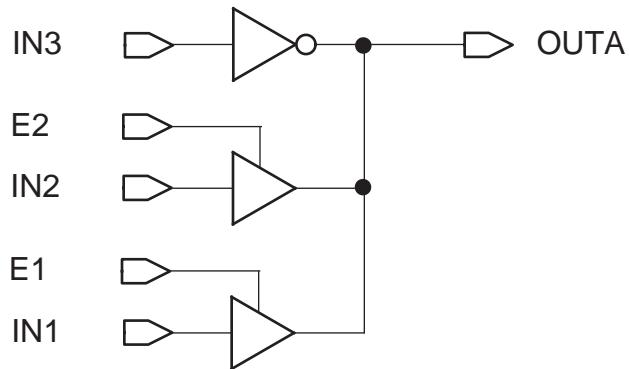
Topological checks are global connectivity checks that the `dft_drc` command performs in a structural manner.

If the `dft_drc` command cannot determine the logic function associated with a wired net, it issues the following warning 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 260](#)) results in contention on that net.

Figure 260 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 topics 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 error 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)

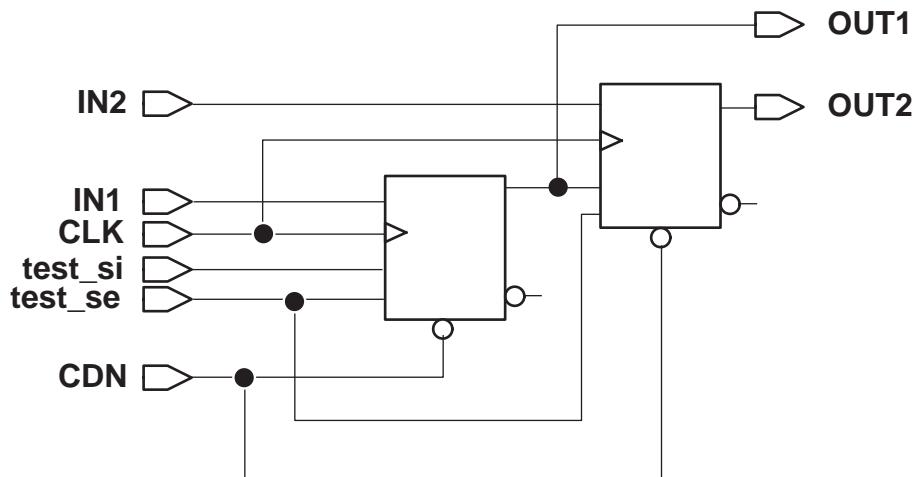
Scan chain violations completed...

The following topics provide examples of the typical causes of uncontrollable scan cells.

Incomplete Test Configuration

[Figure 261](#) shows a simple scan design with a scan chain.

Figure 261 Simple Scan Design



When reading a design from an ASCII netlist that contains existing scan chains, you must specify the test ports. If you do not identify the scan input port, DFT Compiler does not flag any violations during DRC, but it will not be able to extract scan chains.

If the scan input port information is not specified, the `dft_drc` command generates a pre-DFT DRC report even though the netlist contains scan chains:

```
dc_shell> dft_drc
In mode: all_dft...
Pre-DFT DRC enabled
```

```
Information: Starting test design rule checking. (TEST-222)
...

```

Also, the `report_scan_path -chain all` command does not report any scan chains:

```
dc_shell> report_scan_path -chain all
...
=====
TEST MODE: Internal_scan
VIEW      : Existing DFT
=====

=====
AS SPECIFIED BY USER
=====

=====
AS BUILT BY insert_dft
=====
```

No scan path defined in this mode.

To resolve this, identify the scan input ports, scan output ports, test clocks, and asynchronous sets and resets, then rerun `dft_drc`. For example,

```
set_scan_state scan_existing

set_dft_signal -view existing -type ScanEnable -port test_se
set_dft_signal -view existing -type ScanDataIn -port test_si
set_dft_signal -view existing -type ScanDataOut -port OUT2

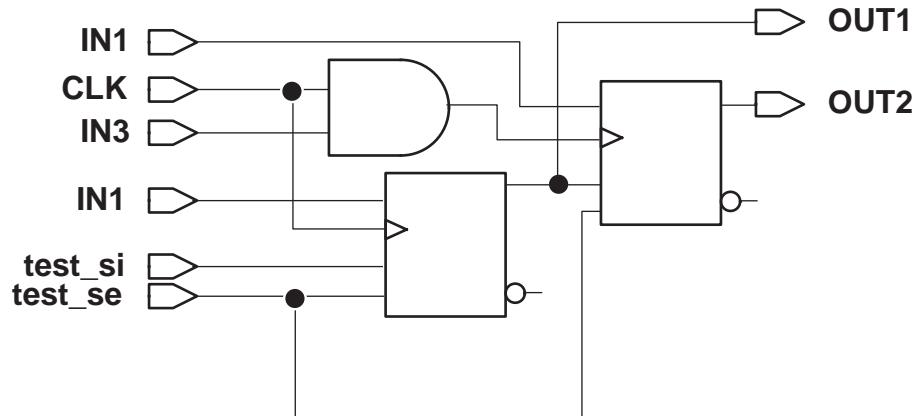
set_scan_path C1 -view existing \
    -scan_data_in test_si -scan_data_out OUT2
```

After the test ports are defined, the `dft_drc` command generates a post-DFT DRC report, and the scan chains are properly inferred.

Invalid Clock Logic

[Figure 262](#) shows a design with a combinational gated clock.

Figure 262 Combinationally Gated Clock

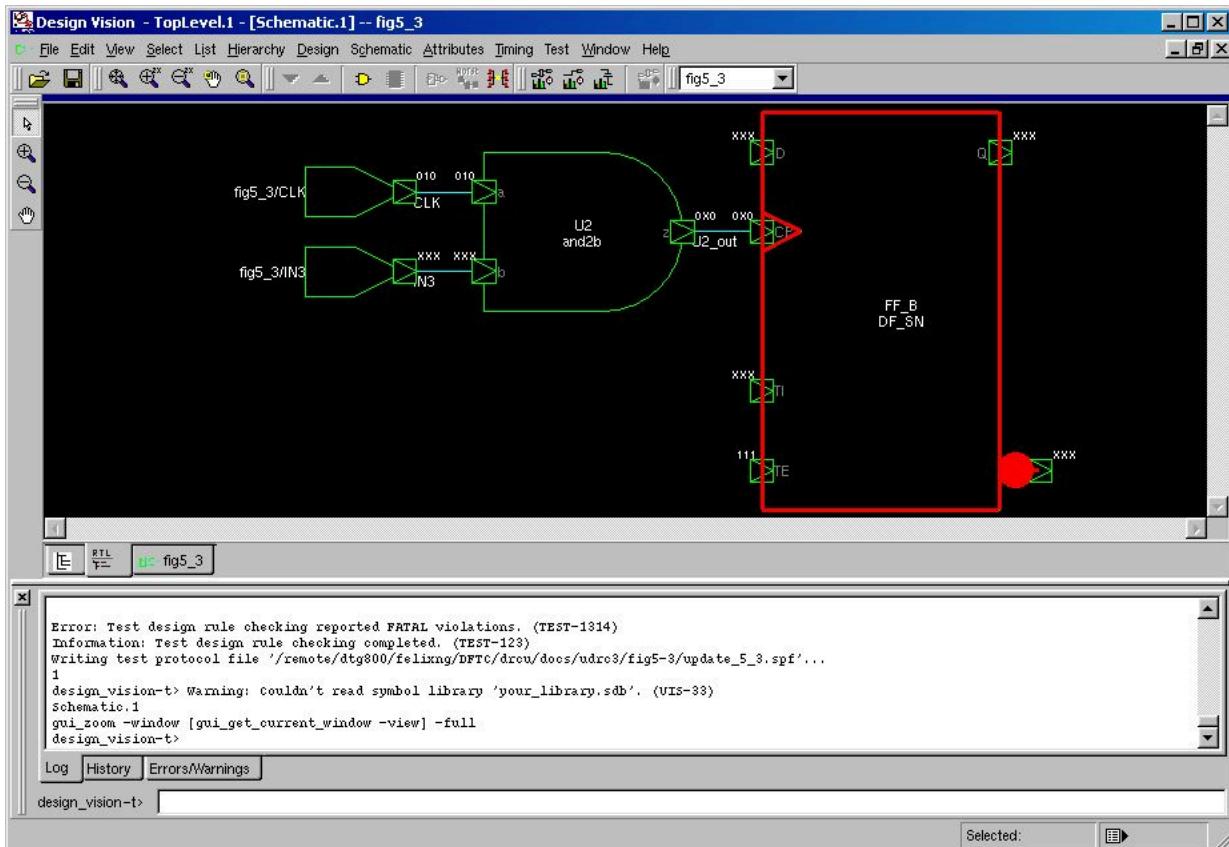


If you do not hold port IN3 at logic 1 during scan shift, pulses applied at clock port CLK might not reach the clock pin of cell FF_B; therefore, the clock input of cell FF_B violates the test clock requirements. DFT Compiler generates error messages such as these:

```
Begin Scan chain violations...
Error: Chain c1 blocked at DFF gate U1 after tracing 0 cells. (S1-1)
Scan chain violations completed...
```

Invoke the Design Vision Graphical Schematic Debugger, as shown in [Figure 263](#).

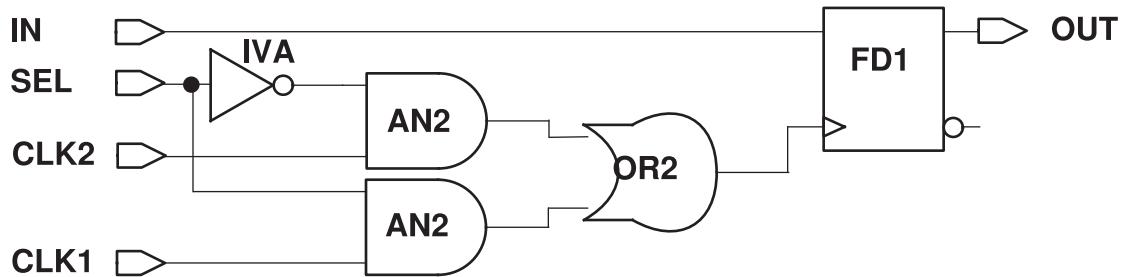
Figure 263 The Design Vision Graphical Schematic Debugger



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 264, 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 264 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 timing values.

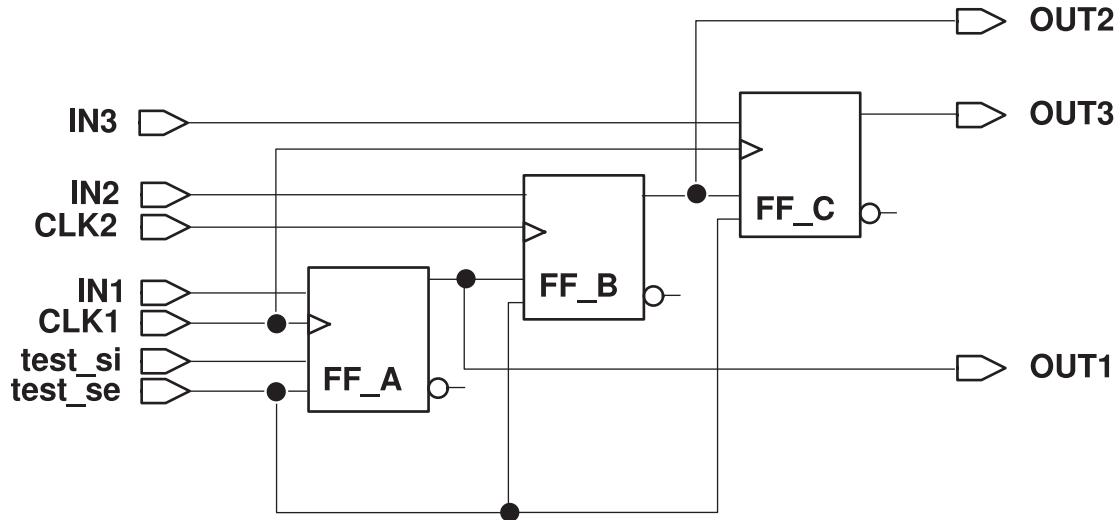
Incorrect Clock Timing Relationship

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 265 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 265 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 warning messages such as

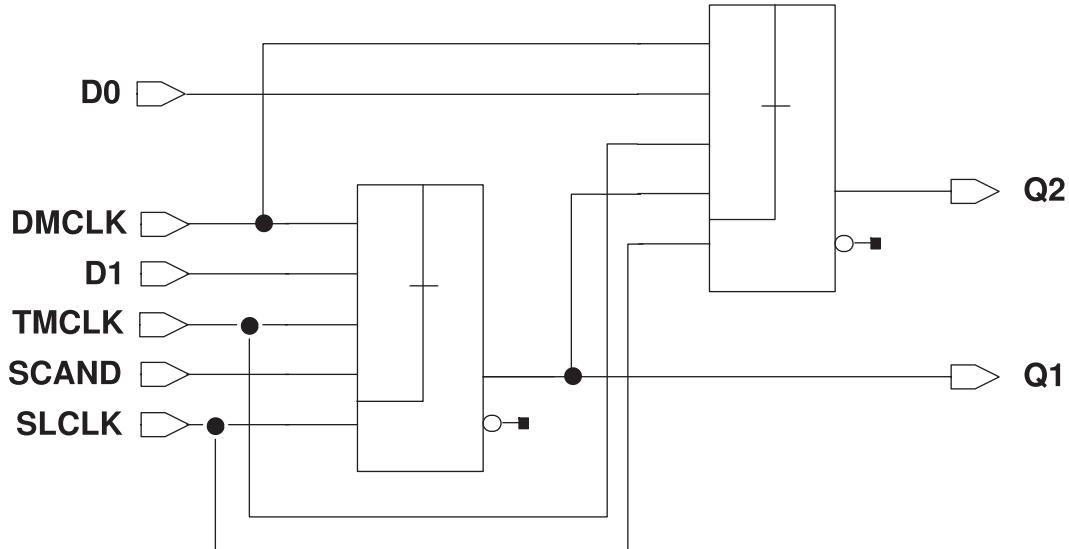
Warning: Multiple clocks (CLK1 CLK2) were used to shift scan chain c1.
 (S22-1)

If the clock timings are different, design rule checker will report warning messages such as

Warning: Dependent slave FF_B may not hold same value as master FF_A.
 (S29-1)

Figure 266 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 266 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 266:

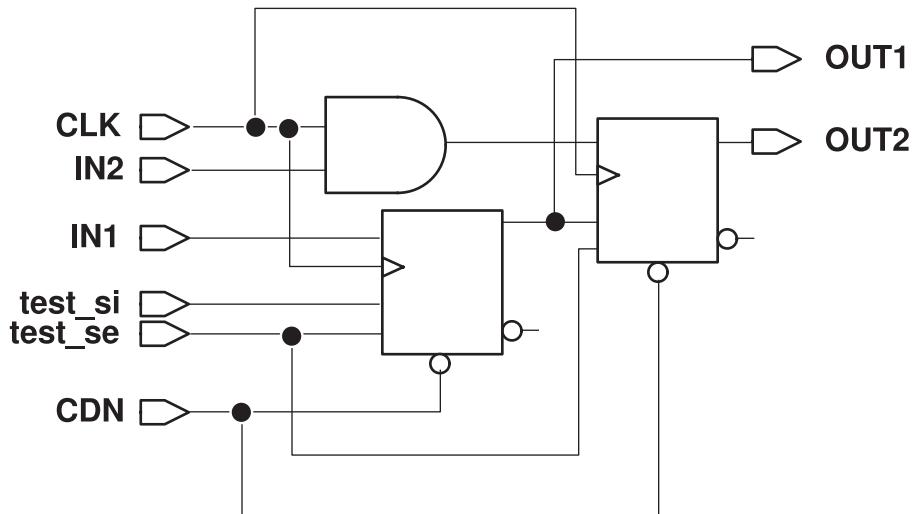
```
dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] \
    -port TMCLK

dc_shell> set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 55 65] \
    -port SLCLK
```

Nonscan Sequential Cells

Figure 267 shows a scan design with a nonscan sequential cell.

Figure 267 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 warning 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.

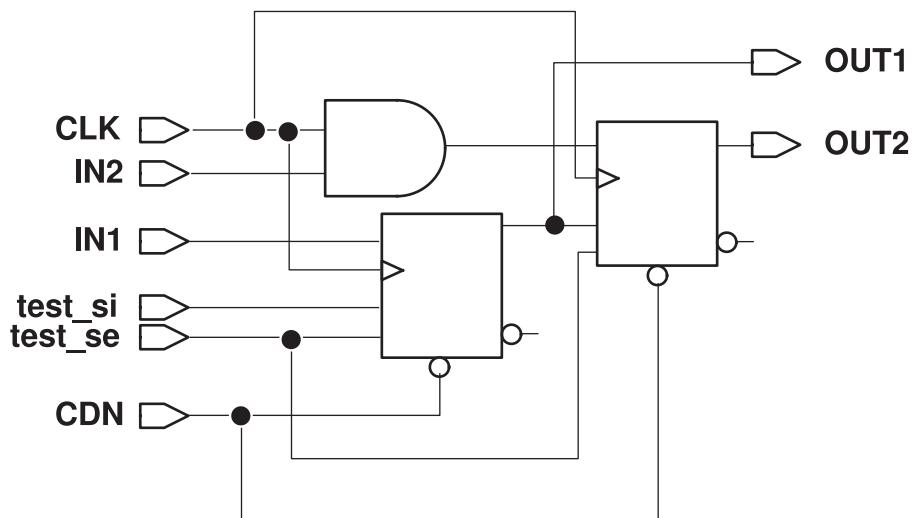
DFT Compiler generates diagnostic messages indicating the source of the violation.

The following topics provide examples of the typical causes of failed data capture.

Clock Driving Data

In the design shown in [Figure 268](#), 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 268 Design With Clock Driving Data



DFT Compiler generates this warning 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.

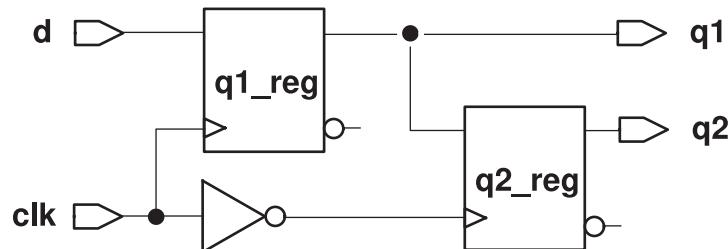
This violation usually has a minor impact on fault coverage, so make it one of the last violations you correct, if at all. Correcting this violation requires the addition of test-mode logic, which also has a minor fault coverage impact. Fixing the violation means trading one set of untested faults for 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 269](#) 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 269 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 warning 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.

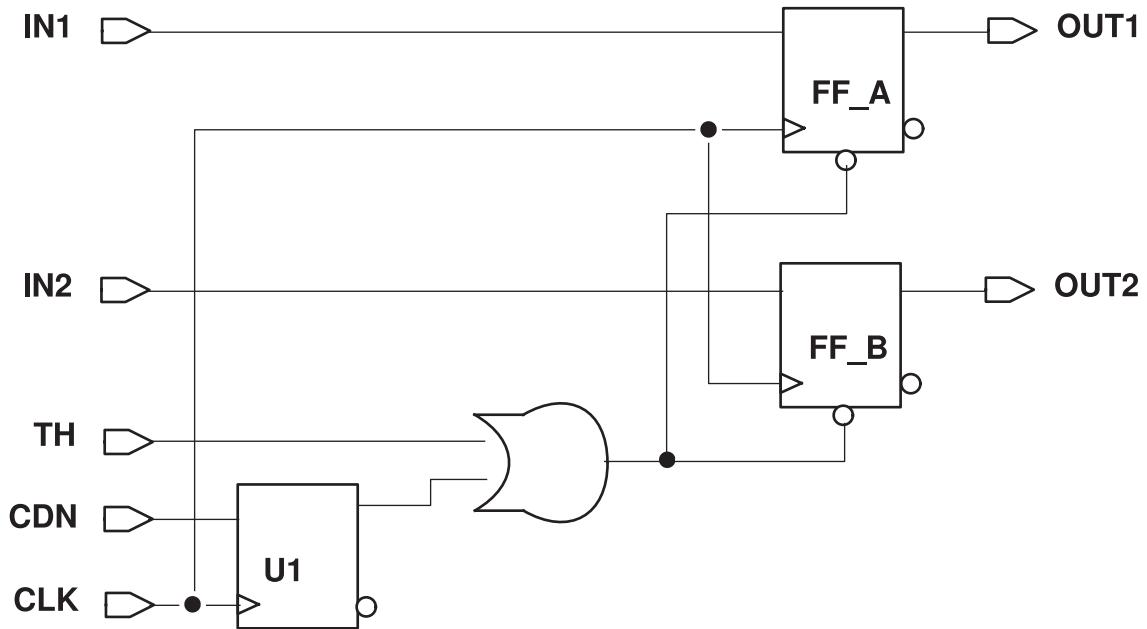
Use the Design Vision Graphical Schematic Debugger to locate and analyze the untestable functional path problem. Contact Synopsys support personnel for access to a script that loads the debugger.

In most cases, you must change the design to correct the problem.

Uncontrollable Asynchronous Pins

The asynchronous pins shown in [Figure 270](#) 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 270 Uncontrollable Asynchronous Pins

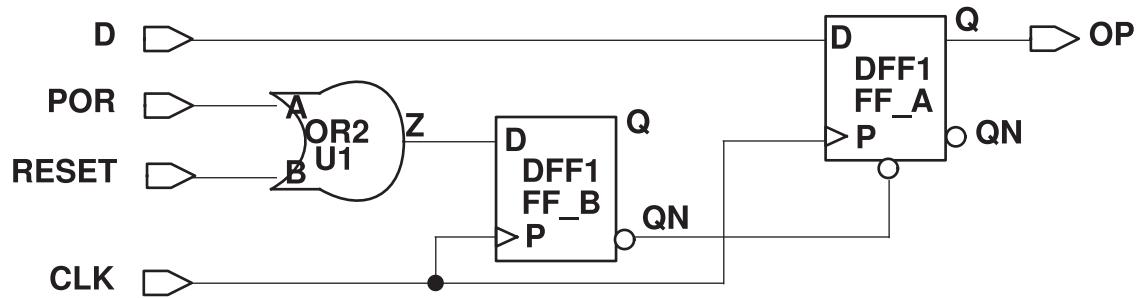


DFT Compiler will report the following:

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 271](#). You can correct this violation by inserting test-mode logic.

Figure 271 Circuit With Uncontrollable Asynchronous Clear



Post-DFT DRC Limitations

Post-DFT DRC is not supported when the following features are used:

- Integrating compressed scan cores
- Using implicit scan chains
- Performing reordering in the ASCII netlist flow
- Placing OCC controllers into bypass mode when you use the `occ_lib_cell_nor2` design attribute to use NOR2 clock ORing logic

15

Exporting Data to Other Tools

This chapter describes how to export output data from DFT Compiler to other tools, such as TestMAX ATPG.

This chapter includes the following topics:

- [Exporting a Design to TestMAX ATPG](#)
 - [Using The SCANDEF-Based Reordering Flow](#)
 - [Verifying DFT Inserted Designs for Functionality](#)
-

Exporting a Design to TestMAX ATPG

After you perform DFT insertion in DFT Compiler, you can write out the design netlist and the STIL protocol files for the test modes of interest. TestMAX ATPG reads these files, performs its own DRC check, and generates test patterns and provides fault coverage statistics for the generated pattern set.

TestMAX ATPG also provides graphical debugging capabilities for DRC violations.

To learn more about exporting your design to TestMAX ATPG, see the following topics:

- [Introduction to STIL Protocol Files](#)
- [Exporting Your Design to TestMAX ATPG](#)
- [Adjusting WaveformTable Timing for Delay Test](#)
- [Reading Designs With Black-Box Test Models Into TestMAX ATPG](#)
- [STIL Protocol File Procedure and WaveformTable Examples](#)
- [Limitations](#)

For more information about TestMAX ATPG, see TestMAX ATPG and TestMAX Diagnosis Online Help.

Introduction to STIL Protocol Files

The `write_test_protocol` command writes out a STIL protocol file (SPF) for a specified test mode. For example,

```
dc_shell> write_test_protocol \
           -test_mode Internal_scan \
           -output Internal_scan.spf
```

A STIL protocol file contains the test protocol information needed by TestMAX ATPG to understand a test mode, such as

- Scan clocks (including clock waveform and strobe timing information)
- Scan chains
- Scan compression structures
- A test setup macro that initializes the design for test
- Various other test procedures that describe how to perform load/unload, launch, and capture operations

In SPF, *macros* and *procedures* describe how to drive the design's primary inputs and observe its primary outputs over one or more clock cycles to perform a particular task. Each macro or procedure references a named WaveformTable construct that defines what signal timing to use. [Table 50](#) describes the macros and procedures that DFT Compiler creates in the SPF.

Table 50 SPF Macros and Procedures Created by DFT Compiler

Macro or Procedure Name	Referenced WaveformTable	Description
test_setup (macro)	_default_WFT_	Initializes the design for the specified test mode
load_unload (procedure)	_default_WFT_	Scans new data into the scan chains while simultaneously scanning the current captured data out of the scan chains
multiclock_capture (procedure)	_multiclock_capture_WFT_	Used for test operations that do not require a high-speed external clock, such as stuck-at capture and delay test launch and capture using an OCC clock
allclock_launch (procedure)	_allclock_launch_WFT_	Performs launch for delay test when using external clocks

Macro or Procedure Name	Referenced WaveformTable	Description
allclock_capture_(procedure)	_allclock_capture_WFT_	Performs capture for delay test when using external clocks
allclock_launch_capture_(procedure)	_allclock_launch_capture_WFT_	For delay test in full-sequential ATPG only, performs launch and capture in the same test clock period when using external clocks

Note:

The three “allclock” procedures are used for delay test, a term that includes the path delay, transition delay, and dynamic bridging fault models. Their WaveformTable timing must be manually modified before use; see [Adjusting WaveformTable Timing for Delay Test on page 632](#).

See [STIL Protocol File Procedure and WaveformTable Examples on page 633](#) for an example of how a procedure references a WaveformTable.

Procedures use the event ordering used by TestMAX ATPG: force PI, measure PO, pulse clock. For a preclock measure protocol, all three events happen within a single test clock cycle. For an end-of-cycle measure protocol, each of the three events happens in its own test clock cycle. For most designs, a preclock measure protocol should be used.

For launch and capture operations, the procedures do not explicitly force the clocks to pulse; instead, the clocks are left unconstrained so that TestMAX ATPG can choose which clocks to pulse in each test pattern.

The SPF reflects any constant signals defined on primary inputs using the `set_dft_signal -type Constant` command. However, it does not reflect assumed signal values applied with the `set_test_assume` command. These assumptions exist only within DFT Compiler in cases where the `final test_setup` initialization procedure is not yet available; you must provide a test protocol to TestMAX ATPG with an updated `test_setup` procedure that matches these assumptions.

For more information about STIL protocol files, see “[STIL Protocol Files](#)” topic in TestMAX ATPG and TestMAX Diagnosis Online Help.

See Also

- [Setting Test Timing Variables on page 581](#) for more information about configuring preclock measure and end-of-cycle measure test protocols in DFT Compiler

Exporting Your Design to TestMAX ATPG

To export your design to TestMAX 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, the following defaults achieve the best results from TestMAX ATPG:

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

These are the default settings; you do not need to add them to your script.

2. Guide netlist formatting by setting 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 for Verilog User Guide* or the *HDL Compiler for VHDL User Guide*.

3. Prior to DFT insertion, check for design rule violations by running pre-DFT DRC:

```
dc_shell> dft_drc
```

Any nonscan sequential cell or capture violation has the potential to lower fault coverage. Fix any design rule violations, then repeat the `dft_drc` command until no design rule violations are found.

For more information, see [Chapter 13, Pre-DFT Test Design Rule Checking.](#)

4. Perform DFT preview and insertion:

```
dc_shell> preview_dft
dc_shell> insert_dft
```

5. After DFT insertion, check for design rule violations by running post-DFT DRC:

```
dc_shell> dft_drc
```

Verify that all scan chains are free from violations. TestMAX ATPG cannot use scan chains with violations.

For more information, see [Post-DFT Insertion Test Design Rule Checking on page 612.](#)

Note:

Some features and flows do not support post-DFT DRC, as noted in [Post-DFT DRC Limitations on page 627](#). In such cases, use DRC in the TestMAX ATPG tool to validate the DFT-inserted design.

6. Write out the design netlist in Verilog format. For example,

```
dc_shell> change_names -hierarchy -rules verilog
dc_shell> write -format verilog -hierarchy -output my_design.v
```

7. Write out a test protocol file for each test mode of interest. For example,

```
dc_shell> write_test_protocol \
           -test_mode Internal_scan \
           -output Internal_scan.spf
dc_shell> write_test_protocol \
           -test_mode ScanCompression_mode \
           -output ScanCompression_mode.spf
```

All of the information that TestMAX ATPG requires to create ATPG test patterns, such as scan pins and constrained signals, is included in the STIL protocol file.

Adjusting WaveformTable Timing for Delay Test

Delay test is a type of ATPG test that targets timing-sensitive faults. It includes the path delay, transition delay, and dynamic bridging fault models.

The SPF created by the `write_test_protocol` command contains three “allclock” procedures, which are used for delay test. The WaveformTables for these procedures contain a copy of the `_default_WFT_` WaveformTable timing by default. To create delay test patterns, you must manually modify the external clocks to constrain the timing as follows:

- `_allclock_launch_WFT_` (referenced by `allclock_launch` procedure)

This WaveformTable describes the delay test launch clock timing for external clocks.

To constrain the timing, modify the WaveformTable timing to move the external clock edges toward the end of the test period (toward the `allclock_capture` cycle).

- `_allclock_capture_WFT_` (referenced by `allclock_capture` procedure)

This WaveformTable describes the delay test capture clock timing for external clocks.

To constrain the timing, modify the WaveformTable timing to move the external clock edges toward the beginning of the test period (toward the `allclock_launch` cycle), keeping the strobe before the first clock edge.

- `_allclock_launch_capture_WFT_` (referenced by `allclock_launch_capture` procedure)

For delay test in full-sequential ATPG only, this WaveformTable describes the delay test timing for external clocks that perform launch and capture in the same test clock period.

To constrain the timing, modify the WaveformTable timing to tighten the pulse width.

Each two-clock transition fault test consists of a launch cycle using `_allclock_launch_WFT_` timing, followed by a capture cycle using `_allclock_capture_WFT_` timing. 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 `_allclock_capture_WFT_` if necessary.

Do not modify any PLL reference clocks, or the PLLs might lose phase lock.

For more information about delay test fault models and how to modify the WaveformTable timing when using them, see TestMAX ATPG and TestMAX Diagnosis Online Help.

Reading Designs With Black-Box Test Models Into TestMAX ATPG

If you export a design that contains black-box cores with test models, the output netlist includes empty submodules for the cores. However, to test the logic inside these cores, they must have an actual netlist representation in automatic test pattern generation (ATPG).

By default, if you read in two modules with the same name into the TestMAX ATPG tool, the last one takes precedence. If you have a top-level netlist with empty submodules, read it into the TestMAX ATPG tool 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_model top
```

STIL Protocol File Procedure and WaveformTable Examples

[Example 84](#) shows a multiclock_capture procedure example. The W construct references a WaveformTable for timing.

Example 84 multiclock_capture Procedure Example

```
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 #;
        }
    }
}
```

Example 85 shows a _multiclock_capture_WFT_ WaveformTable example that provides the timing for the previous multiclock_capture procedure. The formatting has been adjusted for clarity.

Example 85 _multiclock_capture_WFT_ WaveformTable Example

```
WaveformTable "_multiclock_capture_WFT_" {
    Period '100ns';
    Waveforms {
        "all_inputs" {
            0 { '0ns' D; } }
        "all_inputs" {
            1 { '0ns' U; } }
        "all_inputs" {
            Z { '0ns' Z; } }
        "all_inputs" {
            N { '0ns' N; } }

        "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; } }
```

```

    "CLK1" {
        P { '0ns' D;
            '45ns' U;
            '55ns' D; } }

    "CLK2" {
        P { '0ns' D;
            '45ns' U;
            '55ns' D; } }
}

```

Limitations

Note the following limitation:

- TestMAX ATPG does not accept designs in which the original source was VHDL and two-dimensional arrays are used in top-level buses in the final netlist.

Using The SCANDEF-Based Reordering Flow

DFT Compiler can generate SCANDEF information that describes how scan cells in the design can be reordered and repartitioned. You can use this SCANDEF information in the IC Compiler tool to optimize scan chains and to fix timing violations using physical information. You can also use this information in other place-and-route tools.

This topic covers the following:

- [Introduction to SCANDEF](#)
- [SCANDEF Constructs](#)
- [Generating SCANDEF Information](#)
- [Generating SCANDEF Information in Hierarchical DFT Flows](#)
- [SCANDEF Examples](#)
- [Support for Other DFT Features](#)
- [Limitations of SCANDEF Generation](#)

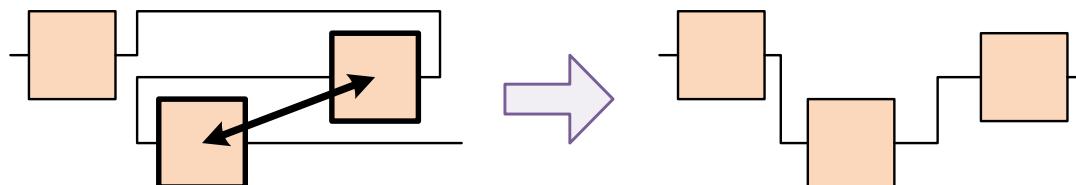
Introduction to SCANDEF

Scan-inserted designs require additional routing compared to nonscan designs. To meet die size and timing requirements, you should reduce the routing overhead as much as possible. One way to do this is to optimize scan chains based on physical information.

There are two types of scan optimization operations:

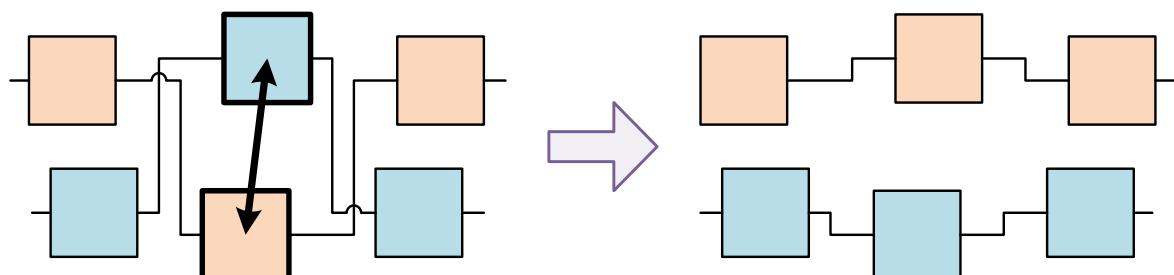
- *Scan reordering* changes the scan chain position of two (or more) scan cells within the same scan chain. [Figure 272](#) shows a scan reordering example.

Figure 272 Scan Reordering of Two Scan Cells



- *Scan repartitioning* swaps two (or more) scan cells between different scan chains, such that the original chain lengths are preserved. [Figure 273](#) shows a scan repartitioning example.

Figure 273 Scan Repartitioning of Two Scan Cells



During scan reordering and repartitioning, the layout tool must honor DFT constraints such as clock mixing, DFT partitions, multivoltage regions, and multiple test modes. However, communicating this information directly to the layout tool would be complex and error-prone.

Instead, SCANDEF communicates what reordering and repartitioning operations are possible given the DFT constraints. As a result, the layout tool does not need to understand DFT constraints; it simply optimizes as allowed by the SCANDEF.

SCANDEF Constructs

A SCANDEF file is a DEF file that uses a set of scan-specific constructs to describe scan chain information. A *stub chain* definition describes a portion of a scan chain that can be reordered within itself. A stub chain consists of a START point, a STOP point, and one or more scan elements between them. [Example 86](#) shows the first three stub chain definitions in a SCANDEF file.

Example 86 SCANDEF Example

```

VERSION 5.5 ;
NAMECASESENSITIVE ON ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN top ;

SCANCHAINS 8 ;

- 1
+ START U131 Y
+ FLOATING ZN_reg[0] ( IN SI ) ( OUT Q )
    ZN_reg[1] ( IN SI ) ( OUT Q )
    ZN_reg[2] ( IN SI ) ( OUT Q )
    ZN_reg[3] ( IN SI ) ( OUT Q )
    ZN_reg[4] ( IN SI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP ZN_reg[4] SI ;

- 2
+ START U132 Y
+ FLOATING ZN_reg[5] ( IN SI ) ( OUT Q )
+ ORDERED SR_reg[3] ( IN SI ) ( OUT Q )
    SR_reg[2] ( IN D ) ( OUT Q )
    SR_reg[1] ( IN D ) ( OUT Q )
    SR_reg[0] ( IN D ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP ZN_reg[4] SI ;

- 3
+ START test_si2
+ FLOATING IP_inst ( IN test_si1 ) ( OUT test_so1 ) (BITS 5)
    IPglue_logic_cell1 ( IN TI ) ( OUT SO )
    IPglue_logic_cell2 ( IN TI ) ( OUT SO )
+ PARTITION IPCLK_45_45
+ STOP test_so2 ;

...

```

Note the following constructs:

- The START and STOP points specify the stub chain boundaries. They can be a variety of scan chain constructs such as scan I/O ports, codec logic gates, lockup latches, reconfiguration MUXes, or buffer/inverter pins. Therefore, stub chains are not usually identical to scan chains, and the number of stub chains defined in the SCANDEF information does not necessarily match the number of scan chains in the design.
- A FLOATING section is an unordered list of cells that can be freely reordered by the layout tool. Because the cells are described as an unordered list, the scan cell order in the SCANDEF file has no requirement to match the order in the design.

- An ORDERED section describes a group of scan cells that cannot be reordered within that group, but can be reordered as a group within a stub chain. Common causes of ORDERED sections are shift registers identified by the `compile_ultra` command, scan segments defined with the `set_scan_path -ordered_elements` command, and buffers or inverters between scan cells.
- The BITS attribute indicates a scan element that represent multiple scan bits. This allows complex scan cells, such as DFT-inserted cores, to be represented in abstract form. By default, each individual scan element represents a single scan bit.
- A PARTITION name indicates that the stub chain elements can be repartitioned with those of another stub chain with the same partition name. The tool constructs partition names so that identical names indicate stub chains that are compatible for repartitioning. Partition names are made unique or omitted for stub chains whose elements cannot be repartitioned.

A stub chain can include zero or more ORDERED sections. However, it can only contain zero or one FLOATING section, as having multiple FLOATING sections within the same stub chain is meaningless.

A SCANDEF file does not necessarily contain all scan cells in the design. It contains information only about scan cells in the design that *can* be reordered or repartitioned. Scan cells or scan segments that cannot be optimized are omitted from the file.

The layout tool can reorder and/or repartition many scan cells at a time. For example, several compatible scan chains in a geographic region can be completely reconstructed, if the SCANDEF information is honored and the original scan chain lengths are preserved.

Generating SCANDEF Information

Generation of SCANDEF information is covered in the following topics:

- [Writing Out the SCANDEF Information](#)
- [Script Example](#)

Writing Out the SCANDEF Information

To generate SCANDEF information, perform the following steps after reading in your design and applying the DFT configuration:

1. Execute the `insert_dft` command.

If you are using Design Compiler topographical mode, perform a post-DFT incremental compile with the `compile_ultra -incremental -scan` command.

2. Execute the `change_names` command with the necessary name rules.

3. Generate the SCANDEF information with the `write_scan_def` command:

```
dc_shell> write_scan_def -output filename.scandef
```

This command writes out the SCANDEF information to the specified file name. It also annotates the current design in memory with the SCANDEF information.

4. Write out the design database files, depending on your layout tool:

- For the IC Compiler tool, use the `write -format ddc` command. This .ddc file contains the SCANDEF information. The IC Compiler tool does not need the SCANDEF file from the previous step, but you can use the file for reference.
- For other layout tools, use the `write -format verilog` command. The layout tool also needs the SCANDEF file from the previous step.

Note:

You must execute the `write_scan_def` command to annotate the scan ordering information onto the current design, even when using the .ddc flow.

You can use the resulting SCANDEF information in your place-and-route tool to optimize scan chain routing order. When you read SCANDEF information into the IC Compiler tool, it checks the integrity of the information against the design netlist before using it.

Script Example

[Example 87](#) shows how to generate the SCANDEF information for a typical design. The script generates a .ddc file with SCANDEF information, and also writes an ASCII SCANDEF file.

Example 87 Example SCANDEF Generation Script

```
read_file -format 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_names ...
write_scan_def -output my_def.scandef
write_test_protocol -output test_mode.spf
write -format verilog -hierarchy -output top.v
write -format ddc -hierarchy -output top.ddc
```

Generating SCANDEF Information in Hierarchical DFT Flows

In hierarchical DFT flows, you perform DFT insertion in one or more cores, then you integrate those cores at the top level. When you write out SCANDEF information at the top level, you can control whether scan optimization is allowed within each core, as described in the following topics:

- [Preventing Scan Optimization in a Core](#)
- [Allowing Scan Optimization in a Core](#)
- [Using SCANDEF Information in a Manual Core Integration Flow](#)

Preventing Scan Optimization in a Core

By default, when you generate SCANDEF information for a design with cores, the tool represents scan chains inside the core with a BITS construct that does not include any individual core-level scan elements. Therefore, scan optimization cannot optimize any individual scan elements within the core, although it can reorder and repartition the core's completed scan chains as scan segments.

[Example 88](#) shows the SCANDEF information for a top-level design that integrates a core containing two scan chains.

Example 88 SCANDEF Information for Scan Optimization Prevented in a Core

```

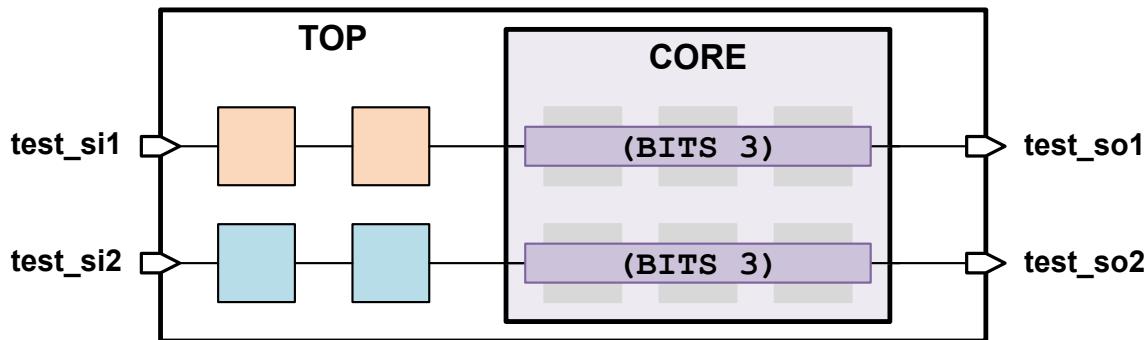
- 1
+ START PIN test_si1
+ FLOATING Z_reg[0] ( IN TI ) ( OUT Q )
    Z_reg[1] ( IN TI ) ( OUT Q )
    CORE ( IN test_si1 ) ( OUT test_so1 ) ( BITS 3 )
+ PARTITION CLK1_45_45
+ STOP PIN test_so1 ;

- 2
+ START PIN test_si2
+ FLOATING Z_reg[2] ( IN TI ) ( OUT Q )
    Z_reg[3] ( IN TI ) ( OUT Q )
    CORE ( IN test_si2 ) ( OUT test_so2 ) ( BITS 3 )
+ PARTITION CLK1_45_45
+ STOP PIN test_so2 ;

```

[Figure 274](#) shows a graphical representation of the SCANDEF information from the previous example.

Figure 274 Schematic Example for Scan Optimization Prevented in a Core



Use this default behavior for cores that are

- IC Compiler block abstractions

Such cores are physically completed cores in layout; no further scan optimization can be performed on them in layout.

Allowing Scan Optimization in a Core

In some cases, you might want to allow scan optimization for a DFT-inserted core. For example, the core might be included as a logical (but not physical) level of hierarchy in the top-level design so that the core-level gates can be freely optimized along with the top-level gates.

To allow scan optimization for one or more core instances, specify them with the `-expand_elements` option when generating the SCANDEF information for the top-level design. For example,

```
dc_shell> write_scan_def -expand_elements {CORE} -output top.scandef
```

The tool incorporates the core-level SCANDEF information into the generated top-level SCANDEF information. Correspondingly, the cores themselves must contain SCANDEF information, which is accomplished in the core-level run by using the `write_scan_def` command before writing out the core design in .ddc or .ctlddc format.

[Example 89](#) shows the SCANDEF information for a top-level design integrating a core with two scan chains represented in expanded form. Note that the individual scan elements inside the core can be reordered and repartitioned with elements outside the core.

Example 89 SCANDEF Information for Scan Optimization Allowed in a Core

```
- 1
+ START PIN test_si1
+ FLOATING Z_reg[0] ( IN TI ) ( OUT Q )
    Z_reg[1] ( IN TI ) ( OUT Q )
    CORE/Z_reg[0] ( IN TI ) ( OUT Q )
    CORE/Z_reg[1] ( IN TI ) ( OUT Q )
```

```

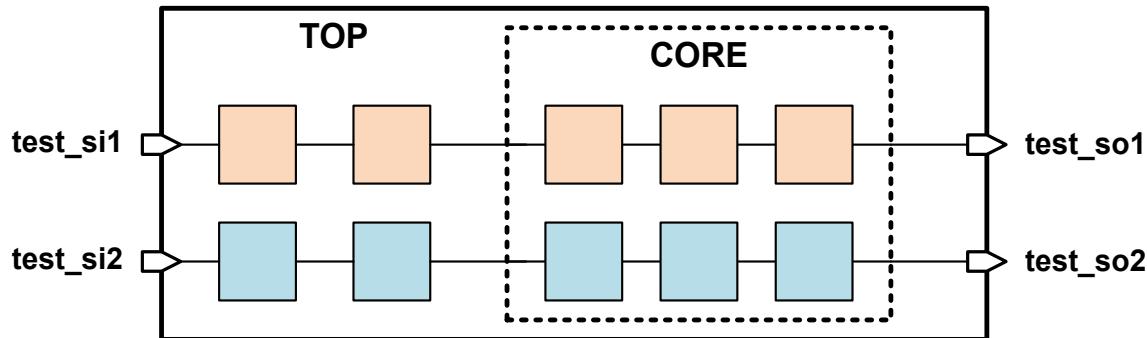
    CORE/Z_reg[2] ( IN TI ) ( OUT Q )
+ PARTITION CLK1_45_45
+ STOP PIN test_so1;

- 2
+ START PIN test_si2
+ FLOATING Z_reg[2] ( IN TI ) ( OUT Q )
    Z_reg[3] ( IN TI ) ( OUT Q )
    CORE/Z_reg[3] ( IN TI ) ( OUT Q )
    CORE/Z_reg[4] ( IN TI ) ( OUT Q )
    CORE/Z_reg[5] ( IN TI ) ( OUT Q )
+ PARTITION CLK1_45_45
+ STOP PIN test_so2;

```

[Figure 275](#) shows a graphical representation of the SCANDEF information from the previous example.

Figure 275 Schematic Example for Scan Optimization Allowed in a Core



If a core does not contain SCANDEF information, the `write_scan_def` command issues a warning:

```
Warning: SCANDEF information for design instance %s is not available.
NETLIST information is available. SCANDEF for design instance %s will be
expanded using netlist information.
```

In this case, DFT Compiler expands the indicated core's scan chains by exploring the core netlist structure. Basic reordering requirements such as clock mixing are inferred from the top level, but any user-applied core-level scan constraints (such as scan group or scan path definitions) are lost.

Use the `-expand_elements` option for cores that are

- Design Compiler block abstractions
- Design Compiler full-netlist designs

Such cores are not yet physically completed cores in layout, and further scan optimization can be performed on them in layout.

See Also

- [Writing Out the SCANDEF Information on page 638](#) for more information about writing out a core design that contains SCANDEF information

Using SCANDEF Information in a Manual Core Integration Flow

In a manual core integration flow, the scan pins of DFT-inserted cores are preconnected in the top-level design and no DFT insertion is performed by DFT Compiler. As a result, the tool does not create any SCANDEF information for the top-level design.

However, you can run scripts that post-process and merge core-level SCANDEF files so that they can be applied to the top-level design in the layout tool. [SolvNet article 017172, “Converting Block-Level SCANDEF to Upper-Level SCANDEF”](#) provides a Perl script that you run in a Linux shell.

SCANDEF Examples

This topic shows how various DFT scenarios are represented in SCANDEF. The examples use a design with six scan cells (FF1 through FF6) and scan-in and scan-out ports for two scan chains (SI1, SI2, SO1, and SO2).

Note:

Depending on your DFT configuration, the tool might use head or tail scan-cell pins instead of scan ports as START and STOP pins, which prevents those scan cells from being optimized. For more information, see [SolvNet article 022408, “Determining START and STOP Points in a SCANDEF File.”](#)

Default (Two Scan Chains)

```
set_scan_configuration -chain_count 2
```

The SCANDEF information is as follows:

```
- 1
+ START PIN SI1
+ FLOATING FF1 ( IN TI ) ( OUT QN )
    FF2 ( IN TI ) ( OUT QN )
    FF3 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO1 ;

- 2
+ START PIN SI2
+ FLOATING FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
    FF6 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO2 ;
```

Both stub chains represent complete scan chains (from scan-in to scan-out). The partition names are identical, which allows scan cells to be swapped (repartitioned) between stub chains.

Mixed Clock Edges

In this specific example, FF1, FF2, and FF3 are clocked by the trailing clock edge.

```
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks
```

The SCANDEF information is as follows:

```
- 1_SG1
+ START PIN SI1
+ FLOATING FF1 ( IN TI ) ( OUT QN )
      FF2 ( IN TI ) ( OUT QN )
+ PARTITION CLK_55_55
+ STOP FF3 TI ;

- 1_SG2
+ START FF3 QN
+ FLOATING FF4 ( IN TI ) ( OUT QN )
      FF5 ( IN TI ) ( OUT QN )
      FF6 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO1 ;
```

A single chain is created because clock mixing is enabled. However, two stub chains are required because scan cells of different clock edges cannot be swapped with each other. The partition name includes the clock edge to implement this restriction. FF3 is fixed as the output of the first stub chain; it cannot be reordered or repartitioned.

set_scan_path With No Elements

```
set_scan_configuration -chain_count 2
set_scan_path MYCHAIN1 -exact_length 2
```

The SCANDEF information is as follows:

```
- 2
+ START PIN SI2
+ FLOATING FF3 ( IN TI ) ( OUT QN )
      FF4 ( IN TI ) ( OUT QN )
      FF5 ( IN TI ) ( OUT QN )
      FF6 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO2 ;

- MYCHAIN1
+ START PIN SI1
+ FLOATING FF1 ( IN TI ) ( OUT QN )
      FF2 ( IN TI ) ( OUT Q )
```

```
+ PARTITION CLK_45_45
+ STOP PIN SO1 ;
```

The stub chain name reflects the scan chain name and the chain lengths reflect the exact length requirement. Otherwise, there are no restrictions; the cells can be reordered and repartitioned.

set_scan_path With Unordered Elements

```
set_scan_configuration -chain_count 2
set_scan_path MYCHAIN1 -include_elements {FF2 FF1}
```

The SCANDEF information is as follows:

```
- 2
+ START PIN SI2
+ FLOATING FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
    FF6 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO2 ;
- MYCHAIN1
+ START PIN SI1
+ FLOATING FF1 ( IN TI ) ( OUT QN )
    FF2 ( IN TI ) ( OUT QN )
    FF3 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45_SNPS_UNIQUE_PARTITION_NAME_00001
+ STOP PIN SO1 ;
```

FF1 and FF2 can be reordered within their stub chain, so they are included in a FLOATING section. However, the `set_scan_path` specification requires that they remain in their scan chain, so a unique partition name is assigned. (This restriction also applies to FF3, which was added for scan chain balancing.)

set_scan_path With Ordered Elements

```
set_scan_configuration -chain_count 2
set_scan_path MYCHAIN1 -ordered_elements {FF2 FF1}
```

The SCANDEF information is as follows:

```
- 2
+ START PIN SI2
+ FLOATING FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
    FF6 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO2 ;
- MYCHAIN1
+ START PIN SI1
```

```
+ FLOATING FF3 ( IN TI ) ( OUT QN )
+ ORDERED FF2 ( IN TI ) ( OUT QN )
    FF1 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45_SNPS_UNIQUE_PARTITION_NAME_00001
+ STOP PIN SO1 ;
```

The ORDERED section requires that FF2 and FF1 can only move within their stub chain as a unit, and the unique partition name ensures that they remain in their scan chain.

Scan Elements That Cannot Be Reordered or Repartitioned

```
set_scan_configuration -chain_count 2
set_scan_path MYCHAIN1 -ordered_elements {FF2 FF1} -complete true
```

The SCANDEF information is as follows:

```
- 2
+ START PIN SI2
+ FLOATING FF3 ( IN TI ) ( OUT QN )
    FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
    FF6 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO2 ;
```

FF2 and FF1—and their scan chain—do not appear in the SCANDEF information at all. The tool detects that they cannot be reordered within their stub chain or repartitioned with other chains, so it omits them to create more efficient SCANDEF information.

Unrouted Scan Groups

```
set_scan_configuration -chain_count 1
set_scan_group MYGROUP1 -include_elements {FF2 FF1} -serial_routed false
```

The SCANDEF information is as follows:

```
- 1_SG1
+ START PIN SI1
+ FLOATING FF3 ( IN TI ) ( OUT QN )
    FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
+ PARTITION CLK_45_45
+ STOP FF6 TI ;

- 1_SG2
+ START FF6 Q
+ FLOATING FF1 ( IN TI ) ( OUT QN )
    FF2 ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45_MYGROUP1
+ STOP PIN SO1 ;
```

FF2 and FF1 are an unordered (unrouted) group, so they can be reordered within their group. However, the unique partition name ensures they cannot be repartitioned out of their stub chain.

Serial-Routed Scan Groups

```
set_scan_configuration -chain_count 1
set_scan_group MYGROUP1 -include_elements {FF2 FF1} -serial_routed true
```

The SCANDEF information is as follows:

```
- 1_SG1
+ START PIN SI1
+ FLOATING FF3 ( IN TI ) ( OUT QN )
    FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
+ PARTITION CLK_45_45
+ STOP FF6 TI ;
```

FF2 and FF1 do not appear in the SCANDEF information at all. The tool detects that they cannot be reordered within their group or repartitioned with other chains, so it omits them to create more efficient SCANDEF information.

CTL-Modeled Core

In this specific example, a CTL-modeled core with a single scan chain is included.

```
set_scan_configuration -chain_count 1
```

The SCANDEF information is as follows:

```
- SUB_GP1
+ START PIN SI1
+ FLOATING FF1 ( IN TI ) ( OUT QN )
    FF2 ( IN TI ) ( OUT QN )
    FF3 ( IN TI ) ( OUT QN )
    FF4 ( IN TI ) ( OUT QN )
    FF5 ( IN TI ) ( OUT QN )
    FF6 ( IN TI ) ( OUT QN )
    block1 (IN test_si1) (OUT test_so1) ( BITS 20 )
+ PARTITION CLK_45_45
+ STOP PIN SO1 ;
```

By default, the CTL model contents are abstracted by the BITS parameter, which indicates the length of that scan segment. However, the contents can be expanded if needed—see [Generating SCANDEF Information in Hierarchical DFT Flows on page 640](#).

Inferred Shift Register

In this specific example, FF5 and FF6 are inferred as a shift register.

```
set_scan_configuration -chain_count 2
```

The SCANDEF information is as follows:

```

- 1
+ START PIN SI1
+ FLOATING FF1_reg ( IN TI ) ( OUT QN )
+ ORDERED FF5_reg ( IN TI ) ( OUT Q )
    FF6_reg ( IN D ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO1 ;

- 2
+ START PIN SI2
+ FLOATING FF2_reg ( IN TI ) ( OUT QN )
    FF3_reg ( IN TI ) ( OUT QN )
    FF4_reg ( IN TI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN SO2 ;

```

The shift register elements are captured in an ORDERED section. It is contained in a partition with a nonunique name, which allows it to be repartitioned into other compatible stub chains.

PARTITION Name Conventions

In a SCANDEF file, a PARTITION name indicates that the stub chain elements can be repartitioned with those of another stub chain with the same partition name. The tool constructs partition names so that identical names indicate stub chains that are compatible for repartitioning. Partition names are made unique or omitted for stub chains whose elements cannot be repartitioned.

The partition naming convention for the different scenarios is as follows:

- MUX-D style without multivoltage

<clock_name>_<capture_time_of_first_state_of_first_segment_of_chain>_<launch_time_of_last_state_of_last_segment_of_chain>

- MUX-D style with multivoltage

<clock_name>_<capture_time_of_first_state_of_first_segment_of_chain>_<launch_time_of_last_state_of_last_segment_of_chain>_<voltage_domain>_<power_domain>

- LSSD style when `test_lssd_no_mix` is FALSE

SNPS_LSSD_<clock_name>_<master_clock_name>_<slave_clock_name>_<voltage_domain>_<power_domain>

- LSSD style when `test_lssd_no_mix` is TRUE
`SNPS_LSSD_<clock_name>_<chain_system_clock_name>_<master_clock_name>_<slave_clock_name>_<voltage_domain>_<power_domain>`
- LSSD style with X-chains
`LSSD_X_<clock_name>_<master_clock_name>_<slave_clock_name>_<voltage_domain>_<power_domain>`
- Scan-enabled LSSD style without multivoltage
`<clock_name>_<capture_time_of_first_state_of_first_segment_of_chain>_<launch_time_of_last_state_of_last_segment_of_chain>`
- Scan-enabled LSSD style with multivoltage
`<clock_name>_<capture_time_of_first_state_of_first_segment_of_chain>_<launch_time_of_last_state_of_last_segment_of_chain>_<voltage_domain>_<power_domain>`
- Multiple test-mode SCANDEF generation (when mode-specific DFT specifications exist)
`<clock_name>_<capture_time_of_first_state_of_first_segment_of_chain>_<launch_time_of_last_state_of_last_segment_of_chain>_M1[_M2_...additional_modes]`

For wrapper chains, `WRPSI_`, `WRPSO_` and `WRPS_` are the corresponding keywords used to represent the different wrapper chains.

Support for Other DFT Features

The following DFT features are supported:

- Standard scan and compressed scan
- User-defined test modes
- Internal pins flow
- Memories with test models
- Multivoltage designs
- Hierarchical flows (with test models)
- On-chip clocking controller (OCC) flows

- Core wrapping
- Shift registers

Limitations of SCANDEF Generation

Note the following limitations of SCANDEF generation:

- Manual post-DFT modifications to the scan structure or scan element naming are not supported.
- Only incremental compiles and the `change_names` command update the stored SCANDEF information. Manual post-DFT design modifications that affect the scan architecture or scan element naming are not supported. Examples include:
- Hierarchy unquification or ungrouping
 - ECO modification to scan chains or logic that affects DFT operation
 - Modifying IEEE 1801 Standard (UPF) power intent to insert isolation and/or level-shifter cells after DFT insertion
 - For stub chains that terminate at a reconfiguration MUX, the STOP pin is the scan-in pin of the last scan cell instead of the input pin of the reconfiguration MUX. This prevents the last scan cell in the stub chain from being reordered or repartitioned.
 - `set_scan_path` specifications with the `-include_elements` or `-ordered_elements` options use a unique partition name to keep those elements in the chain. Any additional scan elements added to that chain for balancing cannot be repartitioned, but they can be reordered.
 - Boundary-scan chains are not supported.
 - Scan extraction flows that have combinational logic between two adjacent scan flip-flops are not supported.
 - `set_scan_group` specifications might not be represented properly.
 - Some `set_scan_path` specifications, when applied to a specific test mode other than the first-defined test mode, are not represented properly.

For details, see [SolvNet article 2314593, “SCANDEF Generation Limitations for Multiple Test Modes.”](#)

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 topics describe the verification process:

- [Verification Setup File Generation](#)
 - [Test Information Passed to the Verification Setup File](#)
 - [Script Example](#)
 - [Formality Tool Limitations](#)
-

Verification Setup File Generation

By default, Design Compiler synthesis automatically creates a verification setup file in your working directory. The automated setup file has the extension .svf 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 the Formality tool.

The syntax is as follows:

```
set_svf
    file_name
    [-append]
    [-off]
```

Argument Definitions

`file_name`

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 reissue the `set_svf` command with the `-append` option.

Test Information Passed to the Verification Setup File

When you run the `insert_dft` command, the following DFT specific information is recorded in the verification setup file:

- Scan-enable signals are disabled.
- Test modes are disabled wherever they are used (for example, AutoFix or scan compression).
- Constants are passed to the file.
- Core wrapper shift (`wrp_shift`) is disabled.
- The TCK, TMS, and TRST ports of boundary-scan designs are held at 0 and the TDO port is not verified.

The setup information is reported in the assumptions summary report.

For more information about verifying design logic with Formality, see [Formality User Guide](#).

Script Example

[Example 90](#) shows you how to use a verification setup file for functionality checking in the Formality tool.

Example 90 Formality Script Example For Equivalence Checking

```
# enable automatic setup to disable scan/test logic
set synopsys_auto_setup true

# set the verification setup file location
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 feature is not supported:

- Internal pins flow

Part 3: DFTMAX Compression

16

Introduction to DFTMAX

DFTMAX compression provides synthesis-based scan data compression technology to lower the cost of testing complex designs, particularly when fabricated with advanced process technologies. These deep-submicron (DSM) designs can have subtle manufacturing defects that are only detected by applying DSM tests, such as at-speed and bridging tests, in addition to stuck-at tests. The extra patterns needed to achieve high test quality for these designs can increase both the test time and the test data, resulting in higher test costs. DFTMAX compression reduces these costs by delivering a significant test data and test time reduction with very low silicon area overhead.

The following topics introduce you to DFTMAX compression:

- [The DFTMAX Compression Architecture](#)
 - [DFTMAX Compression Requirements](#)
 - [Multicore Processing](#)
 - [Limitations](#)
-

The DFTMAX Compression Architecture

The DFTMAX compression architecture is described in the following topics:

- [The DFTMAX Codec](#)
 - [Decompressor Operation](#)
 - [Compressor Operation](#)
 - [The Congestion-Aware DFTMAX Codec](#)
-

The DFTMAX Codec

DFTMAX compressed scan appears similar to standard scan at the chip-level interface, but it contains combinational compression logic and uses many more scan chains of shorter lengths within the chip core. As scan input values are shifted in, the decompressor distributes them across numerous scan chains. The distribution method is accomplished with a patented hardware scheme and a process that allows chip-level scan input values

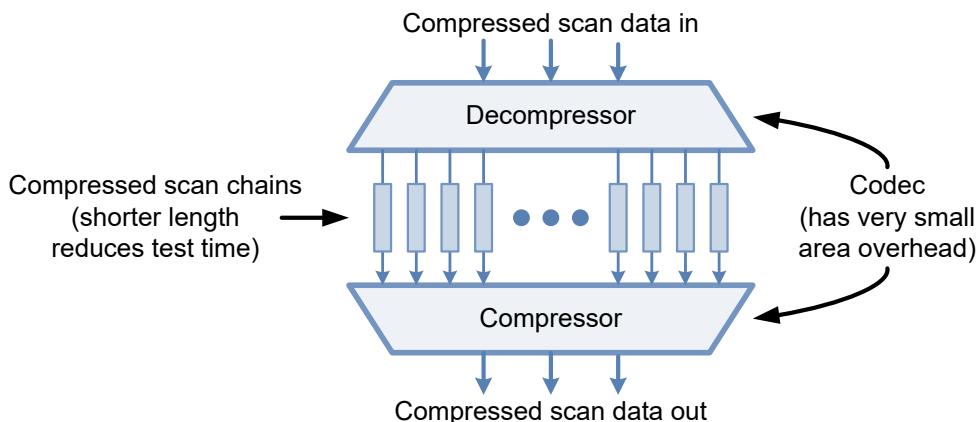
to be placed into various scan chains. To maximize test coverage and minimize pattern count, the decompressor adapts to the needs of automatic pattern generation to supply the required values in the scan chain cells.

DFTMAX compression provides the following key benefits and features:

- A significant test time and test data reduction compared to standard scan
- Similar ease-of-use as standard scan
- Concurrent optimization of area, power, timing, physical constraints, and test constraints through a synthesis-based implementation
- Pin-limited test optimizations
- Unknown logic value (X) handling
- Flexible scan channel configurations to support multisite testing and wafer-level burn-in

[Figure 276](#) shows the DFTMAX compression architecture.

Figure 276 DFTMAX Compression Architecture



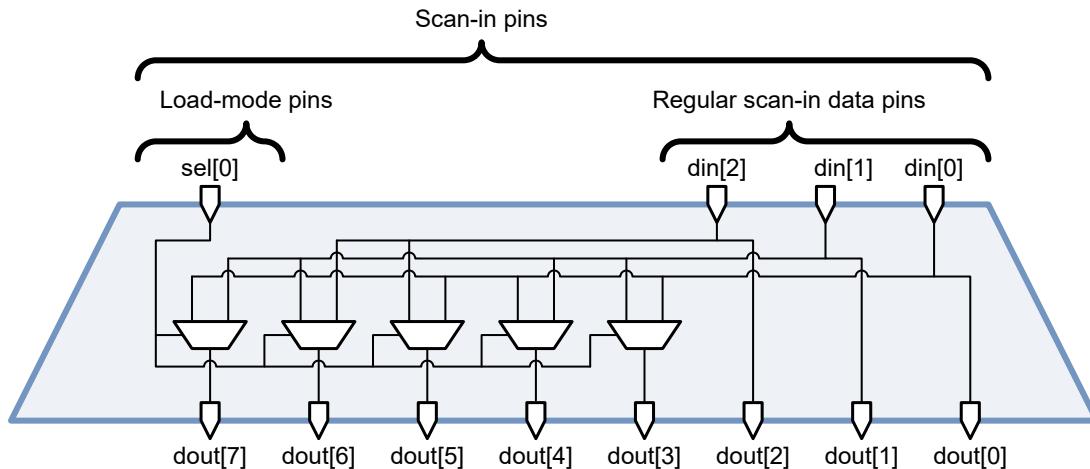
DFTMAX compression divides standard scan chains into a larger number of shorter chains, called *compressed scan chains*, which reduces tester time. The decompressor controls the flow of scan data into the scan chains. The compressor reduces the captured data from the larger number of compressed scan chains so that it can be observed through the scan-out ports. The combination of the decompressor and compressor wrapped around the scan chains is called the *codec*, which is short for compressor-decompressor.

The codec significantly reduces the amount of test data needed to comprehensively test the chip. In turn, this lowers automatic test equipment (ATE) memory requirements and allows additional deep submicron (DSM) test patterns.

Decompressor Operation

[Figure 277](#) shows a decompressor logic structure example. The decompressor outputs are driven by different combinations of scan-in data pins, either directly or through MUXes. One or more scan-in data pins, called *load-mode* pins, are dedicated to the MUX select signals.

[Figure 277 Decompressor Logic Structure Example for 4-to-8 Decompressor](#)



This logic structure takes advantage of the fact that not every scan cell must be uniquely controllable in every pattern. Typically, only a sparse set of scan cells are required to be controlled in a pattern. In each shift clock cycle, ATPG can choose load-mode and scan data values that steer these required values into the compressed scan chains.

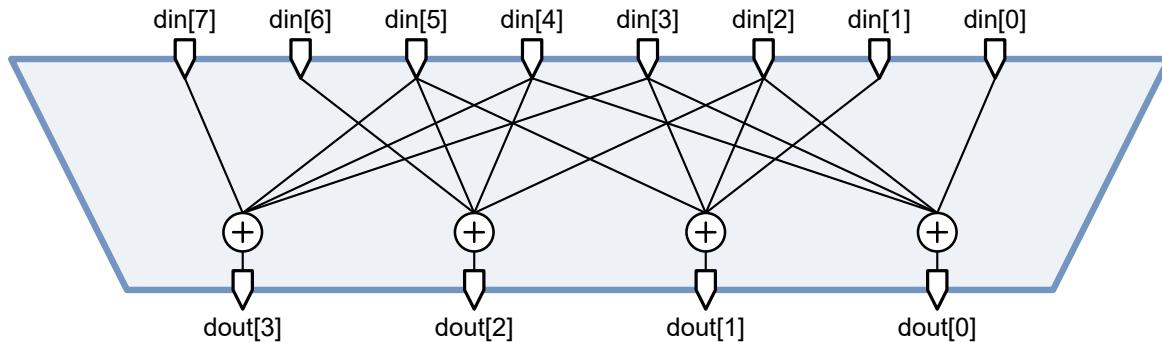
Some designs might have complex logic that requires more scan cells to be controlled in each pattern. As the decompressor input width increases, the number of load-mode and scan-in data pins increases to provide additional controllability at the decompressor outputs.

The decompressor output width is equal to the number of compressed scan chains. As the compressed chain count increases, more data steering logic configurations are needed. If the compressed chain count is increased too high, the data steering configurations must repeat, which can reduce the ability of ATPG to steer data into the compressed chains.

Compressor Operation

[Figure 278](#) shows a compressor logic structure example. The compressor outputs are driven by different combinations of compressed scan chains, combined using XOR logic. An incorrect data value (fault) from a compressed scan chain results in a specific signature of incorrect values at the compressor outputs.

Figure 278 Compressor Logic Structure Example for 8-to-4 Compressor



The compressor input width is equal to the number of compressed scan chains. As the compressed chain count increases, more XOR configurations are needed. If the chain count is increased too high, the XOR configurations (and therefore the compressed chain signatures) must repeat, which can impact the diagnosability of the design.

As the compressor output width increases, the number of fault signatures observed at each output port decreases. This can improve the diagnosability of the design, especially when multiple faults must be simultaneously diagnosed.

The Congestion-Aware DFTMAX Codec

If you are using Design Compiler Graphical (which is enabled by specifying the `-spg` option of the `compile_ultra` command), the TestMAX DFT tool

- Builds congestion-aware decompressor and compressor structures to reduce congestion
- Performs scan chain reordering and repartitioning in the incremental compile to reduce scan chain wire length

See Also

- [Performing Congestion Optimization on Compressed Scan Designs on page 706](#) for more information about using DFTMAX compression in a Design Compiler Graphical flow

DFTMAX Compression Requirements

You need to consider both design and pin requirements when using DFTMAX compression. The following two topics describe these requirements.

Design Requirements

Designs that use DFTMAX compression are generally scan-replaced, that is, test-ready. The supported DFTMAX flows include top-down and bottom-up methodologies. When starting with an RTL design, run the `compile` or `compile_ultra` command with the `-scan` option to bring your design into a test-ready state.

Pin Requirements

DFTMAX compression reuses the scan ports and scan-enable signals that were used in regular scan mode in DFT Compiler. Only one additional test-mode port is required to differentiate between scan compression mode and internal scan mode.

If you already have a test-mode port in your design, you can define it and specify the hookup point by using the following command:

```
set_dft_signal -type TestMode -hookup_pin
```

If you do not have a test-mode port, the `insert_dft` command automatically creates one for you. Note that you cannot use the same test-mode port for the on-chip clocking (OCC), AutoFix, or scan compression mode. If you want to associate a particular test-mode port with the OCC or AutoFix test-mode port, you can do so by using the `-control_signal` option of the `set_ autofix_configuration` command to control the order of the specified TestMode pins.

For accessing the compressed scan chains in compressed scan mode using a single scan-in pin and scan-out pin, use DFTMAX Ultra. You can define the minimal number of access pins for compressed scan mode with the following command:

```
set_scan_compression_configuration -xtolerance default \
    -inputs 1 -outputs 1
```

When using DFTMAX compression with the high X-tolerance capability, a minimum of two scan-in pins and one scan-out pin are supported. You can define the minimal number of access pins in compressed scan mode with high X-tolerance enabled by using the following command:

```
set_scan_compression_configuration -xtolerance high \
    -inputs 2 -outputs 1
```

See Also

- [High X-Tolerance Scan Compression on page 735](#) for more information about the high X-tolerance scan compression architecture

Multicore Processing

DFT Compiler supports multicore processing. To enable this feature, run the following command:

```
set_host_options -max_cores maximum_number_of_cores
```

The *maximum_number_of_cores* value, which specifies the number of cores for threading, should be a positive integer that is 2 or greater.

You must specify the number of cores the tool can use before running the `insert_dft` command.

The *maximum_number_of_cores* setting is persistent throughout a given Design Compiler shell session. Therefore, if you specified this setting during an initial compile stage and did not quit the session, the same setting remains in effect during the DFT insertion stage.

The setting is not saved into a .ddc file. If you quit the session before DFT insertion and then start a new shell, DFT insertion will use the tool's default settings.

If you request more cores than are available, the `insert_dft` command proceeds, using the number of cores that actually are available.

Use the `remove_host_options` command to revert to the tool defaults.

License Usage

For multicore processing, one DFT Compiler or TestMAX DFT license supports up to eight cores, that is, one license is required for every eight cores.

The tool checks out licenses per the specifications defined in the `set_host_options` command.

If you do not have access to a sufficient number of licenses, the `insert_dft` command issues a (SEC-50) error similar to the following:

```
Error: All 'Test-Compression-Synthesis' licenses are in use (SEC-50)
The current users of this feature are:
designer at runhost, started on Thursday 5/7 at 15:52
```

The following examples show commands that are common in a DFT flow, along with the number of required licenses for the multicore operation of each command:

- `compile_ultra -scan -spg` command with power constraints, using 16 cores
 - 2 DC Expert
 - 2 DC Ultra
 - 2 DFT Compiler

- 2 Power Compiler
- 2 DesignWare
- 2 DC-Extension
- `insert_dft` command with compression and no congestion, using 16 cores
 - 2 DFT Compiler
 - 2 DFTMAX
- `insert_dft` command with compression and congestion, using 16 cores
 - 2 DFT Compiler
 - 2 DFTMAX
 - 2 DC Expert
 - 2 DC Ultra
 - 2 DC-Extension
- `insert_dft` command with streaming compression and congestion, using 16 cores
 - 2 DFT Compiler
 - 2 DFTMAX
 - 2 DFTMAX Ultra
 - 2 DC Expert
 - 2 DC Ultra
 - 2 DC-Extension

Consistent with the `compile_ultra` command, licenses are not automatically released until the end of the Design Compiler shell session or until you explicitly issue the `remove_license` command.

If you request more licenses than are available per licensing scheme, the `insert_dft` command stops and issues an error message similar to the following:

```
Error: All 'Test-Compression-Synthesis' licenses are in use. (SEC-50)
The current users of this feature are: designer at runhost, started on
Thursday 5/7 at 15:52
```

Limitations

This topic covers current limitations and known issues associated with DFTMAX compression.

Current Limitations

The following features are currently not supported:

- You cannot use the existing scan flow to insert compression logic into a design that already has standard scan chains at the top level.
 - You cannot use the `set_scan_path` command, unless it is used with a multiple test-mode specification.
-

DFTMAX Compression Limitations

Note the following limitations of DFTMAX compression:

- There is no graphical design rule checking (DRC) debugging support in the Synopsys Design Vision™ GUI for compressor design rule violations.
- There is no plan to support the ability to read back Verilog patterns.
- Write patterns nshifts is not supported with DFTMAX compression.
- You cannot write out a parallel Verilog testbench when you read in the image saved after running DRC in regular scan mode.
- Each pattern in scan compression mode pattern is dependent on the next pattern. Because of this, you cannot reorder any ATPG patterns, including basic scan patterns.

17

Using DFTMAX Compression

If you are currently implementing standard scan logic, you can insert DFTMAX compression into your design by using a single additional command. Typically, no other changes to your design are required.

This chapter describes the processes associated with using DFTMAX compression. It includes the following topics:

- [Top-Down Flat Compressed Scan Flow](#)
- [Top-Down Flat Compressed Scan Flow With DFT Partitions](#)
- [DFTMAX Scan Compression and Multiple Test Modes](#)
- [Excluding Scan Chains From Scan Compression](#)
- [Scan Compression and OCC Controllers](#)
- [Specifying a Different Scan Pin Count for Compressed Scan Mode](#)
- [Adding Compressed Chain Lock-Up Latches](#)
- [Reducing Power Consumption in DFTMAX Designs](#)
- [Forcing a Compressor With Full Diagnostic Capabilities](#)
- [Performing Congestion Optimization on Compressed Scan Designs](#)
- [Using AutoFix With Scan Compression](#)

Top-Down Flat Compressed Scan Flow

This topic describes the top-down flat flow with DFTMAX compression.

[Example 91](#) shows a basic top-down flat compressed scan insertion script for a test-ready design previously compiled with the `compile -scan` command. The three commands in bold indicate the commands added to the scan script to enable scan compression.

Example 91 Script for Enabling Compressed Scan in the Top-Down Flat Insertion Flow

```
read_netlist test_ready.ddc
```

```

set_dft_configuration -scan_compression enable

set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing [list 45 55] ;# default strobe is at 40

set_scan_configuration -chain_count 3 -clock_mixing mix_clocks
set_scan_compression_configuration -chain_count 8

create_test_protocol
dft_drc
preview_dft
insert_dft

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output design.v

write_test_protocol -output scan.spf \
    -test_mode Internal_scan
write_test_protocol -output scancompress.spf \
    -test_mode ScanCompression_mode

```

To insert compressed scan logic in your design, use the `-scan_compression` option of the `set_dft_configuration` command. This is the only required command to enable compressed scan insertion.

```
dc_shell> set_dft_configuration -scan_compression enable
```

When scan compression is enabled, the `insert_dft` command inserts compressed scan logic into the design and defines the following two test modes:

- Compressed scan mode

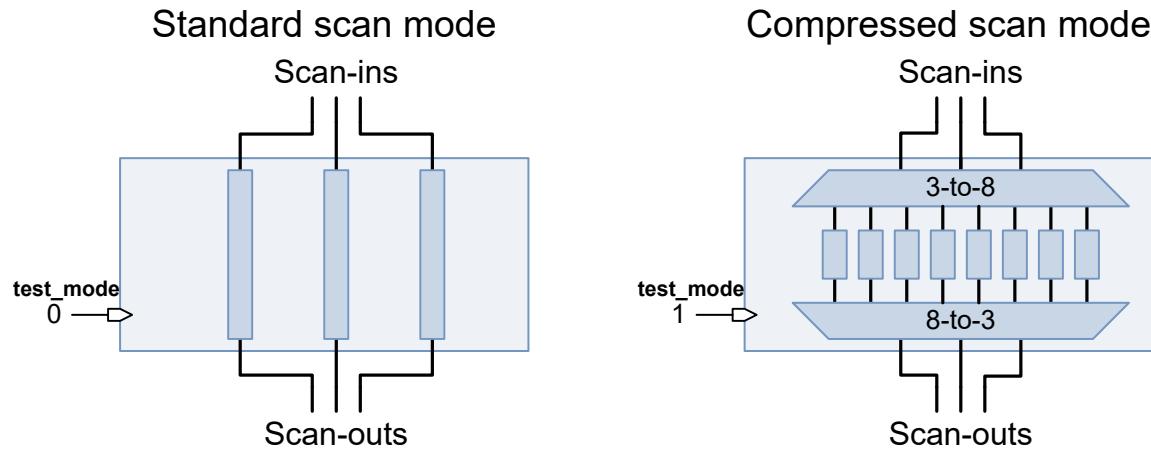
This mode configures the scan elements as short chains driven by decompressors. The default name for this test mode is `ScanCompression_mode`.

- Standard scan mode

This mode joins the short compressed scan chains to reconfigure them into longer standard scan chains. This is also known as standard scan mode. The default name for this test mode is `Internal_scan`.

These test modes are created automatically during compressed scan insertion. You do not need to create them or reference them. [Figure 279](#) shows the scan structures for the two test modes created by [Example 91](#).

Figure 279 Standard Scan and Compressed Scan Modes



At least one test-mode signal is required to select between standard scan mode and compressed scan mode. If a `TestMode` signal is defined with the `set_dft_signal` command, it is used for mode selection. If no test-mode signals are defined, a test-mode port is created and used. Test-mode encodings are created that map the test-mode signal values to each scan mode.

Note:

For more information about working with multiple test modes in DFT Compiler, including information on specifying test-mode encodings, see [Multiple Test Modes on page 357](#).

A compressed scan mode is always associated with a corresponding standard scan mode. The standard scan mode associated with a compressed scan mode is known as its *base mode*. The base mode controls aspects of scan configuration that are common to both modes, such as scan port definitions, scan signal hookup pin definitions, and top-level test access structures.

The `set_scan_configuration` command configures aspects of the standard scan mode, while the `set_scan_compression_configuration` command configures aspects of the compressed scan mode. In [Example 91 on page 663](#), three standard scan chains and eight compressed scan chains are created.

You can use the following options of the `set_scan_compression_configuration` command to control the compressed scan mode chain counts, in order of highest precedence first:

- `-max_length chain_length`

The `-max_length` option specifies the maximum compressed scan chain length. The tool attempts to build the number of compressed scan chains needed to meet this requirement.

- `-chain_count chain_count`

The `-chain_count` option specifies the number of compressed scan chains. The tool attempts to build compressed scan chains with the necessary lengths to meet this requirement.

- `-minimum_compression_ratio`

The `-minimum_compression` option specifies the minimum amount of compression. This is a relative method of specifying the compressed scan chain count. The standard scan chains are subdivided into compressed scan chains according to this ratio, along with a 20 percent pattern inflation factor to account for compression overhead:

```
num_compressed_chains = num_standard_chains * ratio * 1.2
```

You can use any of these options to directly or indirectly specify the scan compression ratio for your design. If none of these options are specified, a minimum compression value of 10 is used. The maximum compressed chain count is 32000.

If you use the high X-tolerance codec architecture, the X-masking architecture places an additional upper limit, as a function of the scan-in and scan-out pin count, on the maximum number of compressed scan chains to ensure 100 percent X-tolerance. For more information, see [High X-Tolerance Scan Compression on page 735](#).

After the standard scan and compressed scan modes have been configured, you can use the `preview_dft` command to see the scan architecture and test-mode signal details before scan insertion is performed, as shown in [Example 92](#).

Example 92 Output From the preview_dft Command for a Compressed Scan Configuration

```
*****
Current mode: Internal_scan
*****

Number of chains: 3
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks

Scan chain '1' (test_si1 --> test_so1) contains 22 cells
  Active in modes: Internal_scan
```

```

Scan chain '2' (test_si2 --> test_so2) contains 21 cells
  Active in modes: Internal_scan

Scan chain '3' (test_si3 --> test_so3) contains 21 cells
  Active in modes: Internal_scan

*****
Current mode: ScanCompression_mode
*****

Number of chains: 8
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix

Scan chain '1' contains 8 cells
  Active in modes: ScanCompression_mode

(...omitted...)

Scan chain '8' contains 8 cells
  Active in modes: ScanCompression_mode

=====
Test Mode Controller Information
=====

Test Mode Controller Ports
-----
test_mode: test_mode

Test Mode Controller Index (MSB --> LSB)
-----
test_mode

Control signal value - Test Mode
-----
0 Internal_scan - InternalTest
1 ScanCompression_mode - InternalTest

```

During scan insertion, the `insert_dft` command creates and instantiates two scan compression designs: one for the compressor and one for the decompressor. By default, the `insert_dft` command instantiates these blocks at the top level of the current design. However, for a top-down flat run, you might want to insert the codec logic into a core-level hierarchical block. For information on inserting the codec logic within a hierarchical instance, see [Specifying a Location for DFT Logic Insertion on page 280](#).

After compressed scan is inserted, you can use the `report_scan_path` command to see details of the scan chains in the standard scan and compressed scan modes:

```
dc_shell> report_scan_path -test_mode all
```

Write out test protocols for both test modes using the `write_test_protocol -test_mode` command:

```
dc_shell> write_test_protocol -output scan.spf \
           -test_mode Internal_scan
dc_shell> write_test_protocol -output scancompress.spf \
           -test_mode ScanCompression_mode
```

After you have the netlist and protocol files, you can generate patterns in the TestMAX ATPG tool.

Compressed scan insertion has the following requirements:

- Compressed scan requires a preclock strobe. You should ensure that the `test_default_strobe` variable is set so that the strobe occurs before the active edges of the test clock waveforms. The default DFT Compiler test timing values meet this requirement.
- Compressed scan insertion requires an HDL-Compiler license.

Top-Down Flat Compressed Scan Flow With DFT Partitions

The following topics describe how you can use DFT partitions in a compressed scan flow:

- [When to Use DFT Partitions in a Scan Compression Flow](#)
- [Configuring Partition Codecs](#)
- [Choosing a Partitioned Codec Insertion Method](#)
- [Per-Partition Scan Compression Configuration Commands](#)
- [Limitations of DFT Partitions in Scan Compression Flow](#)
- [DFT Partition Script Example](#)

See Also

- [Partitioning a Scan Design With DFT Partitions on page 285](#) for general information about DFT partitions

When to Use DFT Partitions in a Scan Compression Flow

For larger designs, inserting a single top-level codec that spans multiple blocks might lead to routing congestion or timing issues due to long routes. For these designs, you can use DFT partitions to insert multiple codecs that provide localized scan compression for different blocks.

This can be accomplished in the following ways:

- Using a top-down flat flow, where multiple codecs are inserted at the same time during top-level DFT insertion
- Using a bottom-up hierarchical flow, where a codec is inserted during DFT insertion for each block, then the block-level codec signals are connected and integrated during top-level DFT insertion

If the entire design does not fit into memory, a bottom-up hierarchical flow must be used. These flows require you to perform bottom-up DFT insertion, which in turn requires block-level DFT constraints, DRC checks, and well-defined hierarchical boundaries. For more information about these bottom-up hierarchical flows, see [Chapter 18, Hierarchical Adaptive Scan Synthesis](#).

However, some designs cannot use a bottom-up hierarchical flow. Constraints might not be available at block-level boundaries, or the available design hierarchy boundaries might not provide the desired mapping of codec blocks to design logic. For these designs, you can use the top-down flat partition flow.

You can specify separate codec configurations for each partition, and all codecs are inserted at the same time during top-level DFT insertion. This top-down flat partition flow provides the same multiple codec flexibility as the bottom-up flows, but without the need for multiple runs.

Configuring Partition Codecs

You can specify the standard and compressed scan mode characteristics for each partition. Not all configuration commands support per-partition specification. See [Per-Partition Scan Compression Configuration Commands on page 672](#) for details.

[Example 93](#) shows an example of global and partition-specific configuration commands.

Example 93 Configuring Two Codecs in a Partition Flow

```
# apply global DFT configuration settings
set_dft_configuration -scan_compression enable -pipeline_scan_data enable
set_dft_signal -view existing_dft \
    -type ScanClock -timing [list 45 55] -port CLK
set_dft_signal -view spec -type TestMode -port TM
set_pipeline_scan_data_configuration \
    -head_pipeline_stages 2 -tail_pipeline_stages 2

# define DFT partitions
define_dft_partition P1 -include {BLK1}
define_dft_partition P2 -include {BLK2}

# configure DFT partition P1
current_dft_partition P1
```

```

set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type ScanDataIn -port {SI1 SI2}
set_dft_signal -view spec -type ScanDataOut -port {SO1 SO2}
set_scan_configuration -chain_count 2
set_scan_compression_configuration -chain_count 4
set_dft_location -include {CODEC} BLK1

# configure DFT partition P2
current_dft_partition P2
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type ScanDataIn -port {SI3 SI4 SI5}
set_dft_signal -view spec -type ScanDataOut -port {SO3 SO4 SO5}
set_scan_configuration -chain_count 3
set_scan_compression_configuration -chain_count 6
set_dft_location -include {CODEC} BLK2

```

In a partition flow, codecs are still inserted at the top level of the current design by default. You can use the `set_dft_location` command to specify the hierarchical block where the codec is to be inserted. For more information, see [Specifying a Location for DFT Logic Insertion on page 280](#).

See Also

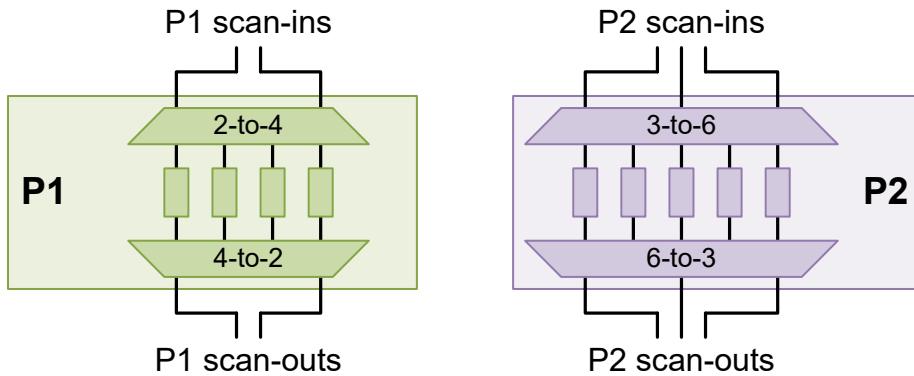
- [Configuring DFT Partitions on page 287](#) for information about the correct order of global and per-partition configuration commands
- [Per-Partition Scan Configuration Commands on page 289](#) for the list of DFT configuration commands that support per-partition specification
- [Per-Partition Scan Compression Configuration Commands on page 672](#) for the list of scan compression configuration commands that support per-partition specification

Choosing a Partitioned Codec Insertion Method

Two different codec insertion methods are available in the DFT partition flow.

By default, the tool creates a separate codec for each partition. The scan-in and scan-out signals associated with each partition are used for that partition's codec connections. Figure 280 shows the dedicated codec architecture for [Example 93 on page 669](#).

Figure 280 Partition Scan Chains With Dedicated Codec Architectures



For improved testability, the tool can split a single shared codec architecture across the partitions. This partitioned codec architecture is enabled by setting the following variable:

```
dc_shell> set_app_var test_enable_codec_sharing true
```

Note:

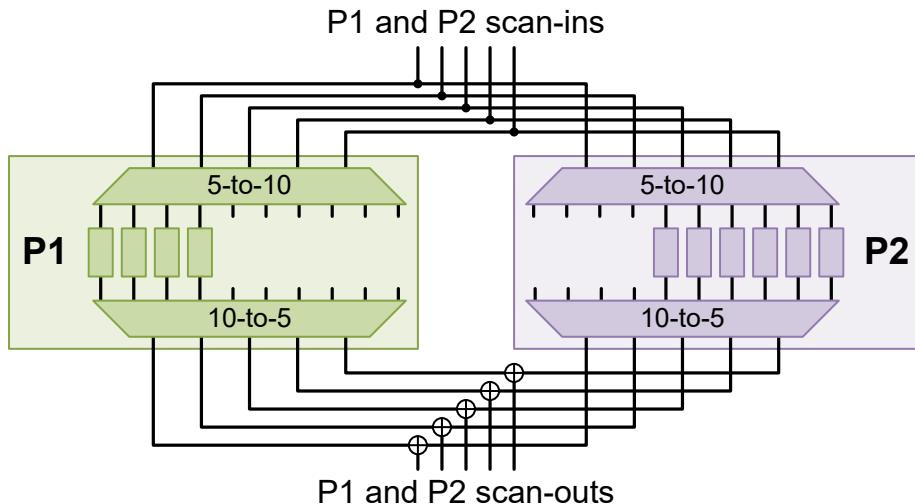
This variable enables the partitioned codec architecture, which is different than the shared codec I/O feature described in [Sharing Codec Scan I/O Pins on page 773](#).

The partitioned codec architecture uses the full set of scan-in and scan-out pins across all partitions. This codec architecture is then replicated for each partition, where only a dedicated subset of decompressor outputs and compressor inputs is connected to the compressed scan chains in each partition. The compressor outputs are combined using an XOR observability tree.

[Figure 281](#) shows the partitioned codec architecture for [Example 93 on page 669](#).

The five scan-in and scan-out pins from the standard scan mode are used to drive the decompressor inputs and capture the compressor outputs. Each decompressor has ten output pins, resulting from the sum of the compressed scan chain counts across all partitions. The same codec architecture is used in each partition, but a different range of compressor outputs and decompressor inputs is connected for each partition. In partition P1, the first four decompressor outputs and compressor inputs are used. In partition P2, the next six decompressor outputs and compressor inputs are used.

Figure 281 Partition Scan Chains With Partitioned Codec Architectures



Note:

To optimize the logic around the unconnected decompressor outputs and compressor inputs, perform an incremental compile after DFT insertion using either the `compile_ultra -scan -incremental` command or the `compile -scan -incremental -boundary_optimization` command.

The partitioned codec architecture has testability efficiency similar to inserting a single codec in an unpartitioned flow. Because all available scan-in and scan-out pins connect to each codec, this method provides better controllability, observability, and X-tolerance to the partitions. However, this insertion method does have some limitations:

- All scan-in and scan-out pins must be shared across all codecs.
- Multiple compressed scan modes are not supported.
- Codecs that compress OCC clock chains are not supported.

Per-Partition Scan Compression Configuration Commands

This topic lists the commands you can use to configure DFTMAX scan compression on a per-partition basis. Commands not listed in this section should be applied as part of the global DFT configuration settings.

See Also

- [Per-Partition Scan Compression Configuration Commands on page 672](#) for the per-partition commands that are not specific to the scan compression flow

set_scan_compression_configuration

The following `set_scan_compression_configuration` options can be specified on a per-partition basis:

- `-minimum_compression`
- `-chain_count`
- `-max_length`
- `-location`
- `-inputs`
- `-outputs`
- `-shared_inputs`
- `-shared_outputs`
- `-shared_codec_controls`
- `-identical_cores`
- `-scramble_identical_outputs`
- `-shared_block_select`
- `-shift_power_chain_length`
- `-shift_power_chain_ratio`
- `-shift_power_clock`
- `-shift_power_disable`

set_dft_location

The following `set_dft_location` compression logic types can be specified on a per-partition basis with the `-include` and `-exclude` options:

- `CODEC`
- `SERIAL_REG`

set_dft_signal

The following `set_dft_signal` options can be specified on a per-partition basis:

- `-type codec_enable [-codec]`

Limitations of DFT Partitions in Scan Compression Flow

The following limitation applies to DFT partitions in DFTMAX compression flows:

- Partition-specific scan-enable signals are not supported by low-power compressor gating, which is enabled with the `set_scan_compression_configuration -min_power true` command.

DFT Partition Script Example

The following script demonstrates the use of compressed scan with multiple DFT partitions, including per-partition scan-in, scan-out, and codec location specifications.

Example 94 Compressed Scan Insertion With Multiple Partitions

```
read_ddc ./design_test_ready.ddc
current_design block
link

# global DFT configuration
set_dft_configuration -scan_compression enable
set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port clk

# define DFT partitions
define_dft_partition partition1 -include {inst1 inst2}
define_dft_partition partition2 -include {inst3 inst4}

# configure each DFT partition
current_dft_partition partition1
set_dft_signal -view spec -type ScanDataIn -port P1_SI*
set_dft_signal -view spec -type ScanDataOut -port P1_SO*
set_scan_configuration -chain_count 5
set_scan_compression_configuration -chain_count 60
set_dft_location -include {CODEC} inst1

current_dft_partition partition2
set_dft_signal -view spec -type ScanDataIn -port P2_SI*
set_dft_signal -view spec -type ScanDataOut -port P2_SO*
set_scan_configuration -chain_count 8
set_scan_compression_configuration -chain_count 60
set_dft_location -include {CODEC} inst4

current_dft_partition default_partition
set_dft_signal -view spec -type ScanDataIn -port PD_SI*
set_dft_signal -view spec -type ScanDataOut -port PD_SO*
set_scan_configuration -chain_count 6
set_scan_compression_configuration -chain_count 60
set_dft_location -include {CODEC} inst5
```

```
# insert DFT
create_test_protocol
report_dft_partition
preview_dft -show all
insert_dft
dft_drc

# write output files
write_scan_def -output ./design.scandef
write -format ddc -hierarchy -output ./scan_inserted_design.ddc
write_test_protocol -output scan.spf -test_mode internal_scan
write_test_protocol -output ascan.spf -test_mode ScanCompression_mode
```

DFTMAX Scan Compression and Multiple Test Modes

When you insert compressed scan into your design, the tool creates two test modes by default:

- A compressed scan mode

The default name for this test mode is ScanCompression_mode.

- A standard scan mode

The default name for this test mode is Internal_scan.

Just as you can create multiple standard scan modes with standard scan, you can also create multiple compressed scan modes with DFTMAX compression. This capability uses the same multiple test-mode creation, configuration, and reporting commands as used with multiple standard scan modes. For more information, see [Multiple Test Modes on page 357](#).

Usage of multiple compressed scan modes is covered in the following topics:

- [Defining Multiple Compressed Scan Modes](#)
- [Per-Test-Mode Scan Compression Configuration Commands](#)
- [Multiple Test-Mode Script Examples](#)

Defining Multiple Compressed Scan Modes

Use the `-usage` option of the `define_test_mode` command to specify the type of scan to be inserted in each mode. In [Example 95](#), two standard scan modes and two compressed scan modes are defined.

Example 95 Defining Multiple Standard Scan and Compressed Scan Modes

```
define_test_mode STDSCAN1 -usage scan
define_test_mode STDSCAN2 -usage scan
define_test_mode COMPSCAN1 -usage scan_compression
define_test_mode COMPSCAN2 -usage scan_compression
```

You can optionally define the test-mode signals and test signal encodings that activate each of these modes. For more information, see [Defining the Encoding of a Test Mode on page 360](#).

When you use the default compressed scan mode for scan insertion, the accompanying standard scan mode is always considered to be the base mode. There is no need to explicitly specify this base mode relationship when a single compressed scan mode is inserted. When you define multiple compressed scan modes, each compressed scan mode must have an associated standard scan base mode.

After defining the compressed scan and standard scan test modes, you must also specify the accompanying base mode relationships for each compressed scan mode. You define these relationships with the `-base_mode` option of the `set_scan_compression_configuration` command. In [Example 96](#), two compressed scan modes are paired with their standard scan base modes.

Example 96 Providing Base Mode Relationships for Compressed Scan Modes

```
set_scan_configuration -test_mode STDSCAN1 -chain_count 2
set_scan_configuration -test_mode STDSCAN2 -chain_count 3
set_scan_compression_configuration -test_mode COMPSCAN1 \
  -base_mode STDSCAN1 -chain_count 4
set_scan_compression_configuration -test_mode COMPSCAN2 \
  -base_mode STDSCAN2 -chain_count 5
```

Although each compressed scan mode must have a corresponding base mode, there is no complementary requirement. You can create as many additional standard scan modes as needed.

Multiple compressed scan modes can share a common base mode. Define the common base mode with the `set_scan_configuration` command, then reference this base

mode using the `-base_mode` option of each `set_scan_compression_configuration` command:

Example 97 Sharing a Base Mode Relationships Across Multiple Compressed Scan Modes

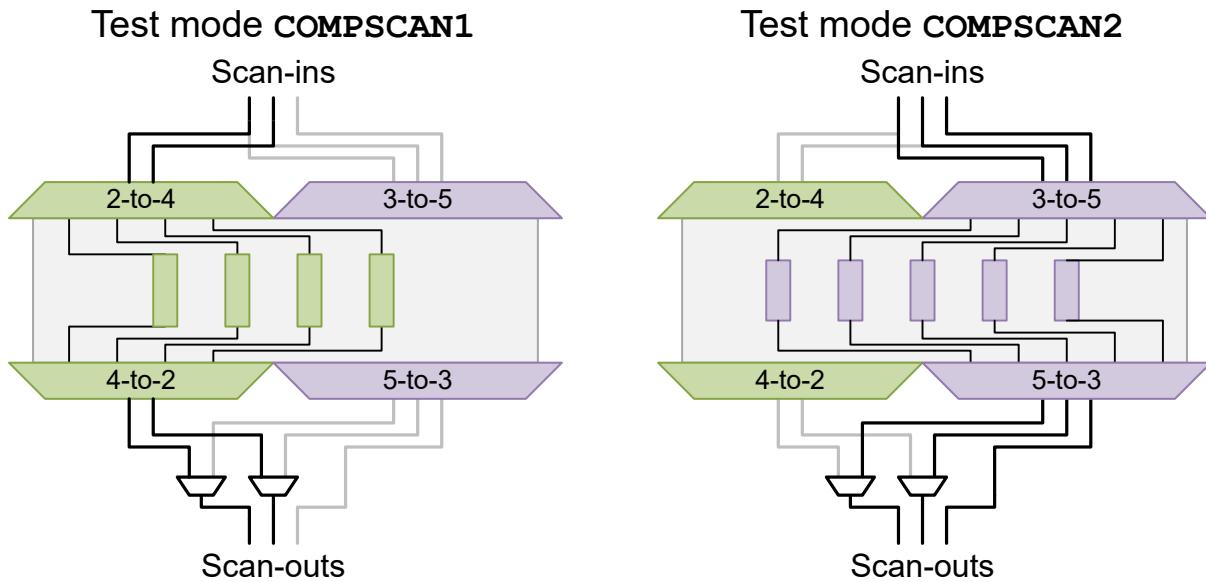
```
set_scan_configuration -test_mode STDSCAN -chain_count 2
set_scan_compression_configuration -test_mode COMPSCAN1 \
    -base_mode STDSCAN -chain_count 4
set_scan_compression_configuration -test_mode COMPSCAN2 \
    -base_mode STDSCAN -chain_count 5
```

For multiple compressed scan modes, the same relationship exists between each compressed scan mode and its corresponding base mode as with a single compressed scan mode and standard scan mode. By default, each compressed scan mode uses all available scan-in and scan-out pins from its base mode. In [Example 96](#), compressed mode COMPSCAN1 creates a decompressor with two inputs, and compressed mode COMPSCAN2 creates a decompressor with three inputs.

If you want to use a different number of scan-in or scan-out pins in a compressed scan mode from what is used in its base mode, you can use the `-inputs` and `-outputs` options of the `set_scan_compression_configuration` command. For more information, see [Specifying a Different Scan Pin Count for Compressed Scan Mode on page 691](#).

The `insert_dft` command inserts a separate codec for each compressed scan mode. This is true even if the codec architectures between two compressed scan modes are identical. Scan chain reconfiguration MUXs are added to provide the necessary scan chain paths for all standard scan and compressed scan modes. Additional MUX logic is added to enable one codec at a time based on the test-mode signal decoding logic. [Figure 282](#) shows how the scan compression codecs in [Example 96](#) are connected.

Figure 282 *Multiple Codecs for Multiple Compressed Scan Modes*



Per-Test-Mode Scan Compression Configuration Commands

This topic lists the commands and options you can use to configure compressed scan insertion for specific test modes. Additional commands are available to configure other aspects of DFT insertion for multiple test modes.

For information about other DFT commands that can be applied to specific test modes, see [Supported Test Specification Commands for Test Modes on page 367](#).

For information on how to order global and mode-specific configuration commands in your scripts, see [Recommended Ordering of Global and Mode-Specific Commands on page 365](#).

set_scan_compression_configuration

The following `set_scan_compression_configuration` options can be applied to specific test modes:

- `-base_mode`
- `-max_length`
- `-chain_count`
- `-minimum_compression`
- `-inputs`

- -outputs
- -shared_inputs
- -shared_outputs
- -identical_cores
- -scramble_identical_outputs
- -shift_power_chain_length
- -shift_power_chain_ratio
- -shift_power_clock
- -shift_power_disable
- -synchronize_chains

Note:

Although the `set_scan_compression_configuration` command applies to the current test mode by default, the `-test_mode` option is typically used together with the `-base_mode` option so that the relationship between the test mode and base mode is explicitly highlighted.

set_scan_path

Use the `set_scan_path -test_mode test_mode_name` command to provide scan chain specifications for each test mode in your design. The scan path specification can be given for any chains in any defined test mode and can include scan data in and scan data out pin specifications, with both port and hookup arguments. If the scan path specification applies to a test mode which has the usage specified as `scan_compression`, then the scan path statements can use the `-hookup` option to specify compressed chains, but they cannot use a port argument.

Multiple Test-Mode Script Examples

The following topics provide examples of multiple test-mode scripts:

- [Multiple Standard Scan Modes and One Compressed Scan Mode](#)
- [Multiple Standard Scan and Compressed Scan Modes](#)
- [Standard Scan Flow Using Multiple Test Modes and Partitions](#)
- [Scan Compression Flow Using Multiple Test Modes and Partitions](#)

Multiple Standard Scan Modes and One Compressed Scan Mode

The following script example demonstrates multiple standard scan modes and a single compressed scan mode.

Example 98 Multiple Standard Scan Modes and One Compressed Scan Mode

```
## Define the scan in and scan out pins, which will be used in
## all test modes.
## These modes are my_base1, scan_compression1, and burn_in.
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, usage and encoding
define_test_mode my_base1 -usage scan \
  -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1}
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}

# Enable DFTMAX compression
set_dft_configuration -scan_compression enable

# Configure DFTMAX compression
set_scan_compression_configuration -base_mode my_base1 -chain_count 32 \
  -test_mode scan_compression1 -xtolerance high

# Configure the basic scan modes
set_scan_configuration -chain_count 4 -test_mode my_base1
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
  -test_mode burn_in

# Enable rapid scan stitching feature
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
```

Chapter 17: Using DFTMAX Compression

DFTMAX Scan Compression and Multiple Test Modes

```

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 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 the test protocol
create_test_protocol

# Run pre-DFT DRC
dft_drc

## Preview test structures to be inserted
preview_dft -show all

## Run test insertion
insert_dft

# run post-DFT DRC in scan_compression1 test mode
current_test_mode scan_compression1
report_dft_signal
dft_drc -verbose

# run post-DFT DRC in my_base1 test mode
current_test_mode my_base1
report_dft_signal
dft_drc -verbose

# run post-DFT DRC in burn_in test mode
current_test_mode burn_in
report_dft_signal
dft_drc -verbose

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output vg/top_scan_mm.v
write_test_protocol -test_mode scan_compression1 \
    -output stil/scan_compression1.stil \
    -names verilog stil/scan_compression2.stil -names verilog
write_test_protocol -test_mode my_base1 -output stil/my_base1.stil \
    -names verilog
write_test_protocol -test_mode burn_in -output stil/burn_in.stil \
    -names verilog

```

Multiple Standard Scan and Compressed Scan Modes

The following script example demonstrates multiple standard scan modes and multiple compressed scan modes.

Example 99 Multiple Standard Scan Modes and Multiple Compressed Scan Modes

```
## Define the scan in and scan out pins, which will be used in
## all test modes.
## These modes are my_base1, my_base2, scan_compression1,
## scan_compression2, and burn_in.
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 DFTMAX compression
set_dft_configuration -scan_compression enable

# Configure DFTMAX compression
set_scan_compression_configuration -base_mode my_base1 -chain_count 32 \
    -test_mode scan_compression1 -xtolerance high
set_scan_compression_configuration -base_mode my_base2 -chain_count 256 \
    -test_mode scan_compression2 -xtolerance 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
```

Chapter 17: Using DFTMAX Compression

DFTMAX Scan Compression and Multiple Test Modes

```

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
create_test_protocol

## Run pre-DFT DRC
dft_drc

## Preview test structures to be inserted
preview_dft -show all

## Run test insertion
insert_dft

current_test_mode scan_compression1
report_dft_signal

```

```

dft_drc -verbose

current_test_mode scan_compression2
report_dft_signal
dft_drc -verbose

current_test_mode my_base1
report_dft_signal
dft_drc -verbose

current_test_mode my_base2
report_dft_signal
dft_drc -verbose

current_test_mode burn_in
report_dft_signal
dft_drc -verbose

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output vg/top_scan_mm.v
write_test_protocol -test_mode scan_compression1 \
    -output stil/scan_compression1.stil -names verilog
write_test_protocol -test_mode scan_compression2 \
    -output stil/scan_compression2.stil -names verilog
write_test_protocol -test_mode my_base1 -output stil/my_base1.stil \
    -names verilog
write_test_protocol -test_mode my_base2 -output stil/my_base2.stil \
    -names verilog
write_test_protocol -test_mode burn_in -output stil/burn_in.stil \
    -names verilog

```

Standard Scan Flow Using Multiple Test Modes and Partitions

The following script example demonstrates multiple standard scan modes and partitions. This example does not insert compressed scan.

Example 100 Multiple Standard Scan Modes With Partitions

```

read_ddc ./design_test_ready.ddc

current_design block
set_dft_signal -view existing_dft -type ScanClock \
    -timing [list 45 55] -port clk

define_test_mode test_mode1 -usage scan
define_test_mode test_mode2 -usage scan

define_dft_partition partition1 -include [list inst1 inst2]
define_dft_partition partition2 -include [list inst3 inst4]

current_dft_partition part1
set_scan_configuration -exact_length 40 -test_mode test_mode1
set_scan_configuration -exact_length 80 -test_mode test_mode2

```

```

current_dft_partition part2
set_scan_configuration -exact_length 40 -test_mode test_mode1
set_scan_configuration -exact_length 80 -test_mode test_mode2

current_dft_partition default_partition
set_scan_configuration -exact_length 40 -test_mode test_mode1
set_scan_configuration -exact_length 80 -test_mode test_mode2

create_test_protocol
report_dft_partition
preview_dft -show all
insert_dft
dft_drc

write_scan_def -output ./design.scandef
write -format ddc -hierarchy -output ./scan_inserted_design.ddc
write_test_protocol -output test_mode1.spf -test_mode test_mode1
write_test_protocol -output test_mode2.spf -test_mode test_mode2

```

Scan Compression Flow Using Multiple Test Modes and Partitions

The following script example demonstrates a single compressed scan mode, multiple standard scan modes, and partitions:

Example 101 Standard Scan and Compressed Scan Modes With Partitions

```

read_ddc ./design_test_ready.ddc

current_design block
set_dft_configuration -scan_compression enable

define_test_mode my_scan_comp -usage scan_compression
define_test_mode test_mode1 -usage scan
define_test_mode test_mode2 -usage scan

define_dft_partition partition1 -include [list inst1 inst2]
define_dft_partition partition2 -include [list inst3 inst4]

current_dft_partition part1
set_scan_configuration -chain_count 4 -test_mode test_mode1
set_scan_configuration -chain_count 8 -test_mode test_mode2
set_scan_compression_configuration -location inst1 \
    -test_mode my_scan_comp -base_mode test_mode1

current_dft_partition part2
set_scan_configuration -chain_count 8 -test_mode test_mode1
set_scan_configuration -chain_count 10 -test_mode test_mode2
set_scan_compression_configuration -location inst3 \
    -test_mode my_scan_comp -base_mode test_mode1

current_dft_partition default_partition
set_scan_configuration -chain_count 2 -test_mode test_mode1

```

```

set_scan_configuration -chain_count 4 -test_mode test_mode2
set_scan_compression_configuration -location inst5 \
    -test_mode my_scan_comp -base_mode test_mode1

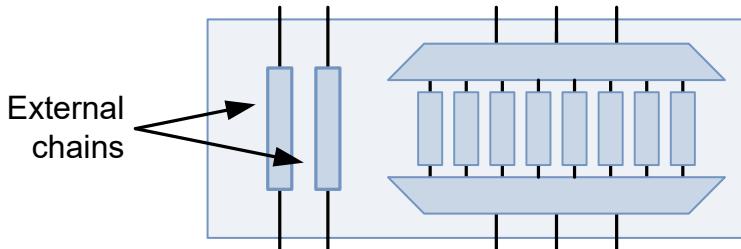
create_test_protocol
report_dft_partition
preview_dft -show all
insert_dft
dft_drc
write_scan_def -output ./design.scandef
write -f ddc -hierarchy -output ./scan_inserted_design.ddc
write_test_protocol -output scan.spf -test_mode test_mode1
write_test_protocol -output ascan.spf -test_mode my_scan_comp

```

Excluding Scan Chains From Scan Compression

In some cases, you might need to exclude specific scan cells from scan compression by keeping them on a separate uncompressed scan chain. Such a scan chain is called an *external chain*. [Figure 283](#) shows two external chains in a compressed scan design.

Figure 283 External Chains in a Compressed Scan Design



To define an external chain, use the `set_scan_path` command as follows:

```

set_scan_path scan_chain_name
    -view spec -test_mode all
    -complete true -dedicated_scan_out true
    -ordered_elements ordered_list
    -scan_data_in port_name -scan_data_out port_name

```

The external chains consume connections in the scan I/O connection budget. In compressed scan mode, the tool automatically uses the remaining scan connections for the codec; you do not need to specify the `-inputs` and `-outputs` options of the `set_scan_compression_configuration` command. The commands in [Example 102](#) configure the scan structure shown in [Figure 283](#).

Example 102 Configuring External Chains in a Compressed Scan Design

```

# define two external chains
set_scan_path EC1 ... ;# external chain 1
set_scan_path EC2 ... ;# external chain 2

```

```
# set standard scan chain count to 5;
# this also sets the scan I/O budget for both scan modes to 5
set_scan_configuration -chain_count 5

# codec uses remaining 3 scan I/O connections;
# you do not need to specify "-inputs 3 -outputs 3"
set_scan_compression_configuration -chain_count 8
```

Note the following aspects of external chain definitions:

- The `-ordered_elements` option specifies the order of the scan cells in the chain. To provide an unordered list and allow the tool to manage ordering requirements such as clock mixing, use the `-include_elements` option instead.
- The `-test_mode all` option applies the external chain definition to both the standard scan and compressed scan modes. To limit the definition to a specific compressed scan mode, specify it with the `-test_mode` option. The specified test mode must be previously defined with the `define_test_mode` command.
- The scan input and output ports must be previously defined with the `set_dft_signal` command. You cannot define external scan chains that use automatically created scan data ports.
- If you are using the pipelined scan data feature, external chains are treated the same as other scan chains. This means,
 - In the automatically inserted pipelined scan data flow, the tool inserts pipeline registers around the external scan chains the same way it does with other scan chains.
 - In the user-defined pipelined scan data flow, you must create and define pipeline registers with head and tail depths that match other scan chains.

See Also

- [HASS and Hybrid Flow Limitations on page 733](#) for limitations of integrating cores that contain external chains

Scan Compression and OCC Controllers

On-chip clocking (OCC) controllers allow on-chip clock sources to be used for at-speed capture during device testing. In an OCC controller flow, the *clock chain* is a special scan segment that provides control over the at-speed capture pulse sequence generated by the OCC controller.

In a scan compression flow, the clock chain can be compressed or external (uncompressed), as described in the following topics:

- [Using Compressed Clock Chains](#)
- [Defining External Clock Chains](#)

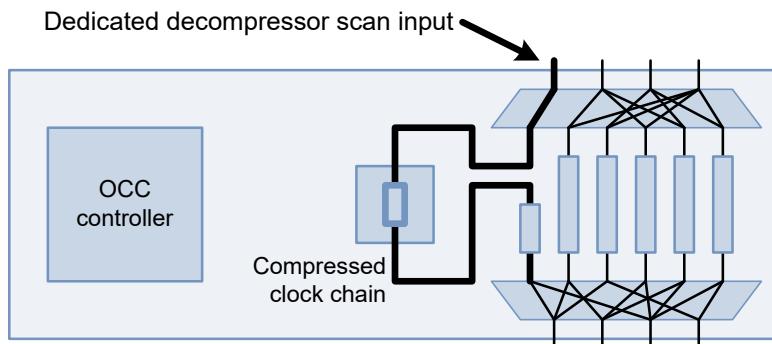
See Also

- [Chapter 12, On-Chip Clocking Support](#) for more information about OCC controllers

Using Compressed Clock Chains

When you insert DFTMAX scan compression along with a DFT-inserted or user-defined OCC controller, the clock chain is placed between the decompressor and compressor by default. The decompressor drives the clock chain along with the other compressed scan chains, but it dedicates a decompressor scan input to the clock chain as shown in [Figure 284](#).

Figure 284 Compressed Clock Chain in a Compressed Scan Design



The decompressor scan input path passes through the decompressor to the clock chain input. This allows the clock chain values to be controlled without imposing ATPG constraints on other scan cells. Such a clock chain is called a *compressed clock chain* because it exists between the decompressor and compressor, even though it is driven by a dedicated scan-in signal as if it was uncompressed.

Note:

For codecs with few scan inputs and high compression ratios, DFTMAX compression might be forced to share the decompressor scan input with other compressed chains. This happens if the codec would otherwise not be implementable.

The dedicated scan-in signal reduces the number of scan-in signals available for data decompression into the remaining compressed chains. You should consider this

when determining compression architecture parameters such as scan input count and compressed chain count.

The clock chain, which is clocked on the trailing edge, is always placed at the beginning of its compressed scan chain. Additional scan cells can follow the clock chain for length-balancing purposes, as allowed by the clock-mixing settings applied to the current design. The compressed scan chain then proceeds into the compressor in the normal way.

If you are using the high X-tolerance feature, the compressed clock chain reduces the maximum compressed scan chain limit that can be created for a given number of scan-in signals. For more information, see [Scan-In and Scan-Out Requirements on page 737](#).

You can use the `preview_dft -show {cells scan_clocks}` command to see which compressed scan chain contains the clock chain. The clock chain is marked with a clock chain segment attribute (o) and a scan segment attribute (s):

```
*****
Current mode: ScanCompression_mode
*****
Number of chains: 16
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix

(1) shows cell scan-out drives a lockup latch
(s) shows cell is a scan segment
(m) shows cell scan-out drives a multi-mode multiplexer
(o) shows cell is a clock chain segment
(w) shows cell scan-out drives a wire

Scan chain '1' contains 7 cells
Active in modes: ScanCompression_mode :

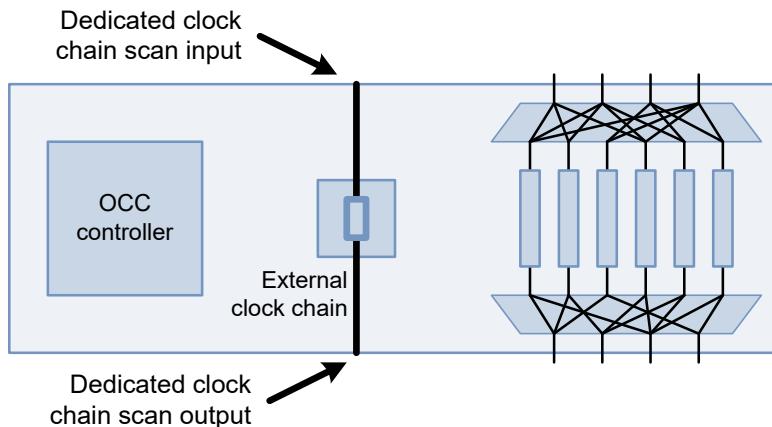
snps_clk_chain_0/clock_chain (s) (o) (UPLL/CLKO, 55.0, falling)
Z1_reg[0] (UPLL/CLKO, 45.0, rising)
Z1_reg[1]
Z1_reg[2] (m)
```

For details on previewing scan segments, see [Previewing Additional Scan Chain Information on page 604](#).

Defining External Clock Chains

External clock chains are uncompressed and exist outside the codec as shown in [Figure 285](#).

Figure 285 External Clock Chain in a Compressed Scan Design



External clock chains are normally used only with the following features:

- Shared codec I/O

Compressed clock chain inputs cannot be shared. To avoid multiple unshareable inputs across shared-I/O cores, use external clock chains, which can be concatenated into a single top-level clock chain across the cores. See [Codec I/O Sharing With OCC Controllers on page 787](#).

- DFTMAX Ultra

External clock chains are implemented by default for designs with DFTMAX Ultra compression; they do not require explicit specification as described in this section. See [Using OCC Controllers With DFTMAX Ultra Compression on page 944](#).

To manually define the complete external clock chain, use the `set_scan_path` command. This method allows you to use specific scan-in and scan-out signals for the clock chain.

For example,

```
set_dft_signal -view spec -type ScanDataIn -port OCC_SI
set_dft_signal -view spec -type ScanDataOut -port OCC_SO
set_scan_path \
    MY_clock_chain -class occ \
    -include_elements {\ \
        snps_clk_chain_2/clock_chain \
        CORE1/clock_chain_name \
        CORE2/clock_chain_name} \
    -complete true \
    -scan_data_in OCC_SI -scan_data_out OCC_SO \
    -test_mode all
```

The `-class occ` option indicates that the scan path specification defines a clock chain. Use the `-include_elements` option to allow the tool to change the element order, or use

the `-ordered_elements` option to use only your specified order. The `-test_mode all` option must be specified.

Include all top-level and core-level clock chains that comprise the complete clock chain, as follows:

- Top-level DFT-inserted OCC controllers

Specify the name of the clock chain that DFT insertion will build. For more information, see [SolvNet article 018046, “How Can I Control Scan Stitching of OCC Controller Clock Chains?”](#)

- Top-level user-defined OCC controllers

Specify the name of the clock-chain scan group, which must be previously defined with the `set_scan_group` command.

- Core-level clock chain segments

Specify the core-level clock-chain segment names. You can use the `preview_dft -show {cells segments}` command to help determine their names.

You can define multiple external clock chains, if needed.

If you are using DFT partitions, all clock chains to be concatenated must belong to the same partition. See [SolvNet article 2675107, “Concatenating OCC Clock Chains From Multiple DFT Partitions.”](#)

If you have DFTMAX-only cores with compressed clock chains, do not include these compressed clock chains in the external clock-chain definition. These compressed clock chains operate normally when the core is active in its DFTMAX mode.

See Also

- [Excluding Scan Chains From Scan Compression on page 686](#) for general information on defining external chains

Specifying a Different Scan Pin Count for Compressed Scan Mode

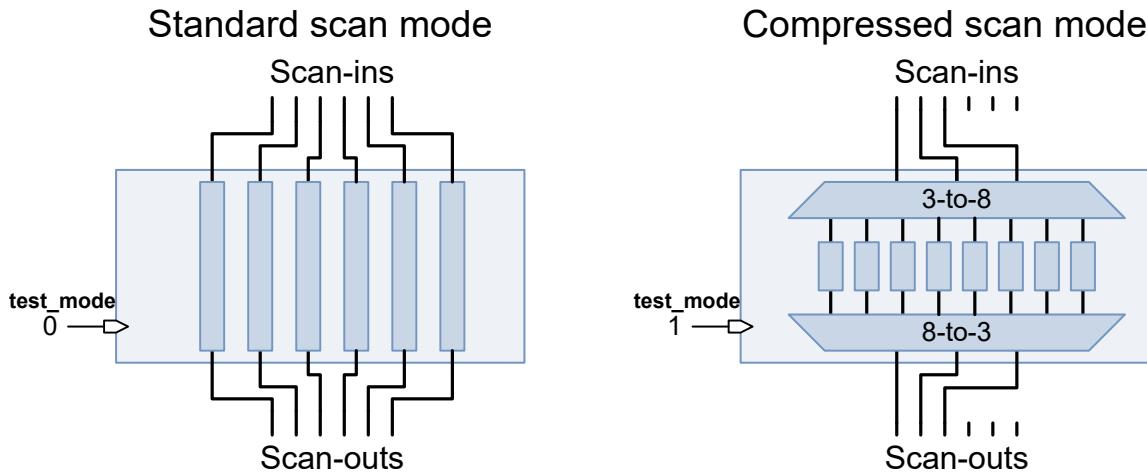
In a compressed scan mode, the decompressor inputs are driven by scan-in pins and the compressor outputs drive scan-out pins. By default, DFTMAX compression uses the full set of scan-in and scan-out pins from the associated base mode for the scan compression codec construction as shown in [Figure 279 on page 665](#). If you want to use a different number of scan-in and scan-out pins for the codec construction, you can use the `-inputs` and `-outputs` options of the `set_scan_compression_configuration` command.

Figure 286 shows the codec architecture created by the `-inputs` and `-outputs` options in Example 103.

Example 103 Specifying a Different Compressed Scan Pin Configuration

```
set_scan_configuration -chain_count 6 -clock_mixing mix_clocks
set_scan_compression_configuration -chain_count 8 -inputs 3 -outputs 3
```

Figure 286 Different Compressed Scan Pin Codec Architecture



You can use any number of scan-in and scan-out pin connections for a compressed scan mode relative to the chain count of its base mode. The only requirement is that the scan-in pin and scan-out pin counts are less than the compressed scan chain count.

You can also use the `-inputs` and `-outputs` options to specify asymmetrical scan I/O configurations for scan compression, where the number of scan-in pins differs from the number of scan-out pins, as shown in Example 104.

Example 104 Specifying an Asymmetrical Scan Pin Configuration

```
set_scan_configuration -chain_count 6 -clock_mixing mix_clocks
set_scan_compression_configuration -chain_count 8 -inputs 5 -outputs 3
```

The minimum number of scan-in and scan-out pins for a compressed scan mode is

- One scan-in and one scan-out pin when default X-tolerance is used

```
set_scan_compression_configuration \
-xtolerance default -inputs 1 -outputs 1
```

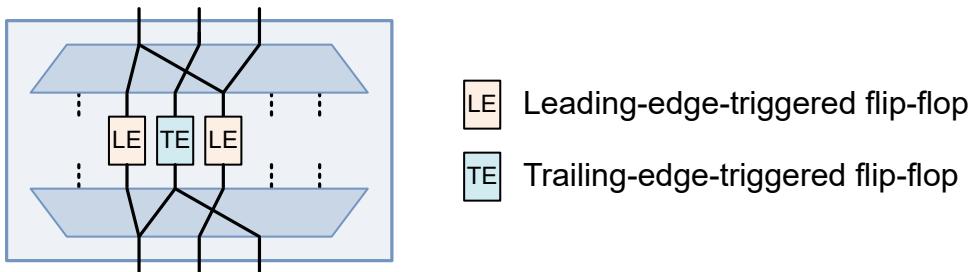
- Two scan-in pins and one scan-out pin when high X-tolerance is used

```
set_scan_compression_configuration \
-xtolerance high -inputs 2 -outputs 1
```

Adding Compressed Chain Lock-Up Latches

Shift speeds are limited by the ability to propagate data between an I/O pad cell and the compressed scan chains. The MUX and XOR gates that are introduced in compressed scan can add further delay to the scan paths at the scan-in and scan-out paths, respectively. Because of the one-to-many relationship of compressed scan chains to scan I/Os, it is possible for leading-edge and trailing-edge scan cells to share the same scan input or output port, as shown in [Figure 287](#). In these cases, the resulting mix of launch and capture clock edges reduces the usable clock period.

Figure 287 Compressed Scan Chains With Multiple Edge Polarities



This mixed-edge penalty against shift frequency in compressed scan mode can be avoided by using lock-up latches at the start or end of the leading-edge or trailing-edge compressed chains, respectively. By selectively inserting lock-up latches at the end or start of the compressed chains, the output and input flip-flops are synchronized to the same clock edge. The inputs of the scan chains are always synchronized to the trailing edge, and the outputs of the scan chains to the leading edge.

To synchronize the scan chains, use the following command:

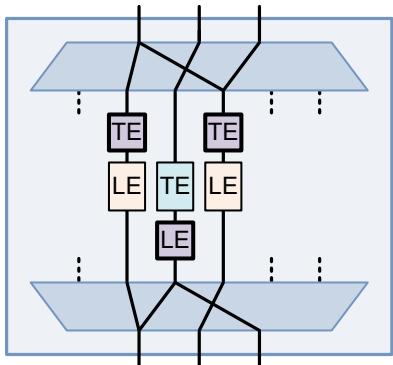
```
set_scan_compression_configuration
    -synchronize_chains head | tail | all | none
```

You can specify the following values for the `-synchronize_chains` option:

- `head` synchronizes the first shift state of all compressed scan chains to the trailing clock edge.
- `tail` synchronizes the last shift state of all compressed scan chains to the leading clock edge.
- `all` synchronizes the first and last shift states of all compressed scan chains.
- `none` does not add lock-up latches to the ends of compressed scan chains for synchronization. This is the default.

[Figure 288](#) shows how the previous scan compression logic is created when the `-synchronize_chains all` option is used.

Figure 288 Compressed Scan Chains With Multiple Edge Polarities and Synchronization



- [LE] Leading-edge-triggered flip-flop
- [TE] Trailing-edge-triggered flip-flop
- [LE] Leading-edge lock-up latch
- [TE] Trailing-edge lock-up latch

Note the following restrictions and behaviors:

- Only scan chains coming from the compressor-decompressor are synchronized. All other chains, such as user-defined logic chains and phase-locked loop (PLL) chains, are ignored during synchronization.
- Because the inserted lock-up latches are bypassed in standard scan mode, a C3 violation is reported for these latches during standard scan mode DRC.
C3 Clock PI's off state failed to allow transparency of nonscan DLAT S
- The synchronization specification is ignored if you use any of the following features:
 - `set_scan_configuration -add_test_retimming_flops`
 - `set_scan_configuration -insert_terminal_lockup true`
 - Pipelined scan data

Reducing Power Consumption in DFTMAX Designs

You can reduce the power consumption of designs with scan compression by using the following features:

- [Reducing Compressor Power When Codec Is Inactive](#)
- [Reducing Scan Shift Power Using Shift Power Groups](#)

Reducing Compressor Power When Codec Is Inactive

In a compressed scan architecture, an XOR compression tree combines the shift outputs from all compressed chains into a reduced set of scan out data signals. This XOR compression tree is needed only during scan shifting in that codec's compressed scan mode. At other times, the compression logic is not needed, but it will still toggle when the

tail scan flip-flops of the compressed chains toggle. This is a particular concern during mission mode, when the flip-flops are clocked at their full operating frequency.

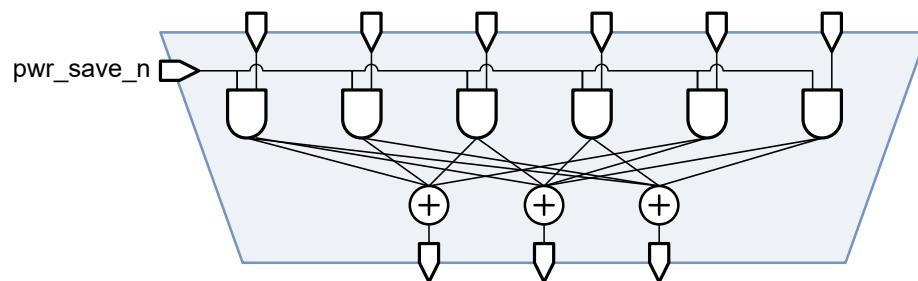
To address this, the tool can insert gating at the inputs to the XOR compression tree to eliminate this toggling activity and reduce power consumption in other modes of operation.

To enable XOR compressor gating, specify the following option:

```
dc_shell> set_scan_compression_configuration -min_power true
```

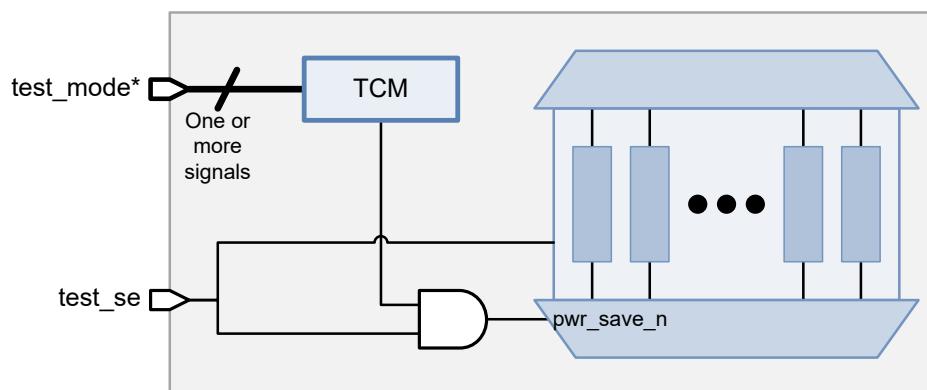
When you enable XOR compressor gating, the tool inserts gating at the inputs to the XOR compression tree as shown in [Figure 289](#). Every compressor input is AND-gated with an active-low pwr_save_n signal before going to the XOR tree.

Figure 289 Example of a Default X-Tolerant Compressor With Gated Inputs



The pwr_save_n gating signal is generated by combining the scan-enable signal and the test control module (TCM) signal for the codec's compression mode, as shown in [Figure 290](#). By generating this gating signal from the existing test-mode and scan-enable signals, no additional dedicated control signal is required at the block boundary. In a CTL test model flow, this also means that no update to a block's test model is needed when XOR compressor gating is added to a block.

Figure 290 Example of Top-Level Compressor Power Gating Control



Preserving Compressor Gating Cells During Optimization

When combinational gating exists at the inputs to a DFTMAX XOR compression tree, it is possible for the gate remapping algorithms to push gating cells further into the compression tree when common terms exist in at the inputs of XOR compression gates. When the gating cells are pushed further into the tree, toggling can occur at the upstream XOR cells, and the power reduction effectiveness is reduced. To preserve the full power reduction benefits, logic synthesis optimization must not move the gating cells into the compression tree.

In the .ddc flow, this is automatically handled. the tool applies the `size_only` attribute to the gating cells as soon as they are created, so that optimization cannot remap the gating cell functionality into a different gate-level structure. Because this attribute persists in a .ddc flow, the gating cell placement and the power reduction benefits are preserved.

However, in the Verilog flow, this attribute does not persist. To get the same power-saving benefits, run the `write_script` command before you write out the Verilog netlist. Extract the commands that apply the `size_only` attribute to the compressor gating cells. When the netlist is read back in, use these commands to reapply the `size_only` attribute to the power gating logic.

Reducing Scan Shift Power Using Shift Power Groups

You can use shift power groups to reduce power consumption during scan shift. This feature is described in the following topics:

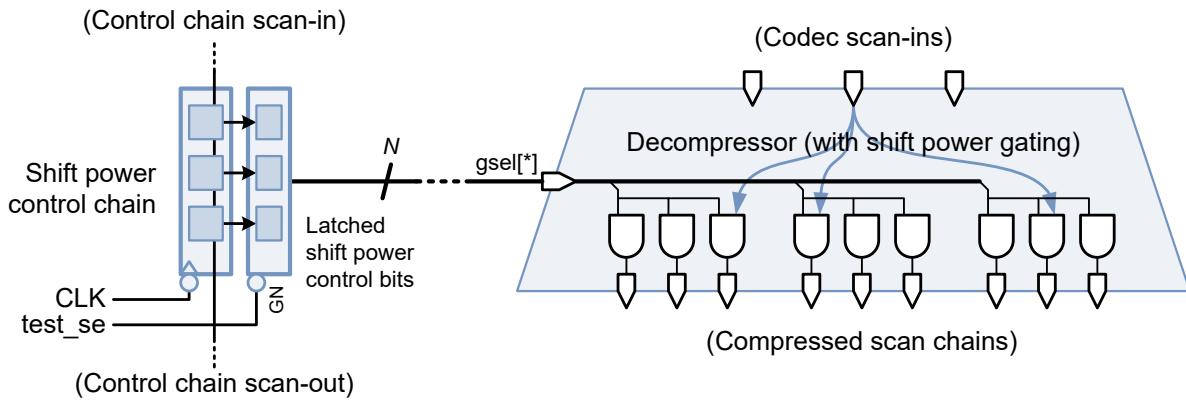
- [The Shift Power Groups Architecture](#)
- [Scan-Enable Signal Requirements for Shift Power Groups](#)
- [Configuring Shift Power Groups](#)
- [Integrating Cores With Shift Power Groups in Hierarchical Flows](#)
- [Configuring Shift Power Groups in TestMAX ATPG](#)
- [Using Shift Power Groups With Other DFT Features](#)
- [Limitations of Shift Power Groups](#)

The Shift Power Groups Architecture

During scan shift, there is significant toggle activity in the scan chains. At high scan shift frequencies, this can result in higher-than-desired shift power consumption.

The shift power groups feature helps reduce power consumption during scan shift in DFTMAX compressed scan modes. This feature inserts AND gates at the decompressor outputs before each compressed scan chain. The chains are gated in groups that are controlled by a shift power control (SPC) chain, as shown in [Figure 291](#).

Figure 291 Shift Power Groups Decompressor Architecture



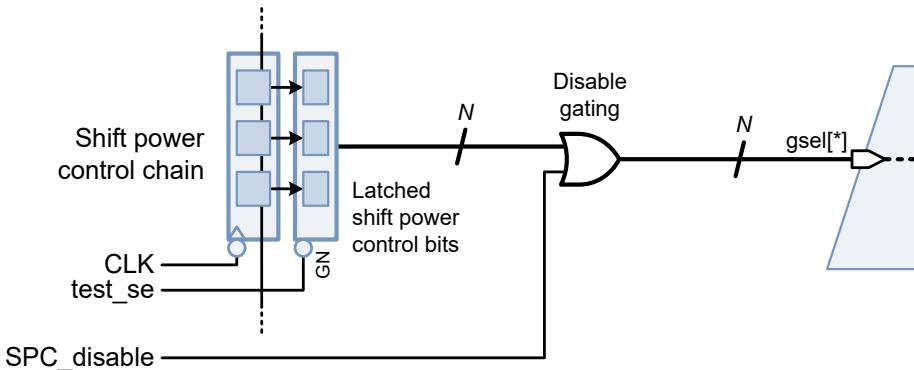
The SPC chain is an external (uncompressed) chain outside the DFTMAX codec. When scan-in completes, the SPC registers contain the group mask values for the next pattern. The de-asserted scan-enable signal, `test_se`, latches these bits into shadow latches that retain the mask values for scan-in of the next pattern.

TestMAX ATPG configures the group masking in each pattern, depending on the power constraints and the number of care bits in each chain group. The larger number of short chains inherent to scan compression provide finer granularity for this control. Masked groups load constant values into their chains, which reduces overall toggle activity.

SPC chains must be external chains because a compressed SPC chain would gate itself, preventing it from reliably loading in each pattern.

The shift power logic also includes a hardware disable signal that, when asserted, disables the shift power logic by enabling all compressed chains, as shown in [Figure 292](#). This signal must be de-asserted or asserted prior to DRC, depending on whether the shift power groups feature is enabled in TestMAX ATPG or not, respectively.

Figure 292 Shift Power Disabling Logic



Scan-Enable Signal Requirements for Shift Power Groups

When the DFTMAX shift-power codec scan-enable signal is de-asserted, the shift power logic latches the values from the control chain. Therefore, for proper operation, *this scan-enable signal must be held in the inactive state in all capture procedures.*

The STIL protocol file created by the tool does not apply this constraint. As a result, you must manually constrain any scan-enable signals used by DFTMAX shift-power codecs, as described in [Configuring Shift Power Groups in TestMAX ATPG on page 701](#).

Alternatively, if you use a custom STIL protocol file, you can update it to constrain the scan-enable signals in all capture procedures.

Configuring Shift Power Groups

To configure the shift power groups feature, do the following:

1. Enable the shift power groups feature.

```
dc_shell> set_scan_compression_configuration \
           -shift_power_groups true
```

2. Specify the configuration of the compressed chain groups.

- To directly specify the number of compressed chain groups, and therefore the length of the SPC chain, use the `-shift_power_chain_length` option:

```
dc_shell> set_scan_compression_configuration \
           -shift_power_chain_length 16
```

- To specify the number of compressed chains in each group, which makes the SPC chain length a function of the compressed chain count, use the `-shift_power_chain_ratio` option:

```
dc_shell> set_scan_compression_configuration \
           -shift_power_chain_ratio 12
```

These options are mutually exclusive.

3. Define the shift power groups disable signal.

```
dc_shell> set_dft_signal -view spec -type TestControl \
           -port SPC_DISABLE
```

```
dc_shell> set_scan_compression_configuration \
           -shift_power_disable SPC_DISABLE
```

You can define the disable signal using the `-port` and/or `-hookup_pin` options of the `set_dft_signal` command. For an “internal pins” hookup pin, you must use a `test_setup` protocol that de-asserts the disable signal.

4. Configure the shift power control chain.

- If no OCC controllers (DFT-inserted or user-defined) are configured in the current design, you must configure an external SPC chain.

Specify the clock, scan-in, and scan-out signals to use for the SPC chain:

```
dc_shell> set_scan_compression_configuration \
           -shift_power_clock CLK
```

```
dc_shell> set_scan_path SPC -class spc \
           -scan_data_in SPC_IN \
           -scan_data_out SPC_OUT \
           -test_mode all
```

You do not need to specify SPC scan path elements; the SPC chain is automatically included in the specification.

- If OCC controllers (DFT-inserted or user-defined) are configured in the current design, configure an external clock chain:

```
dc_shell> set_scan_path OCC -class occ \
           -scan_data_in OCC_IN \
           -scan_data_out OCC_OUT \
           -test_mode all ;# includes the SPC chain too
```

An external clock chain is required because a compressed clock chain would be gated, preventing it from reliably loading in each pattern.

By default, the tool automatically appends the SPC chain to the clock chain. It is clocked by the ATE clock unless specified otherwise with the `-shift_power_clock` option of the `set_scan_compression_configuration` command.

Alternately, you can explicitly define a separate external SPC chain as previously described, which provides independent access to the OCC and SPC chains when the core is integrated.

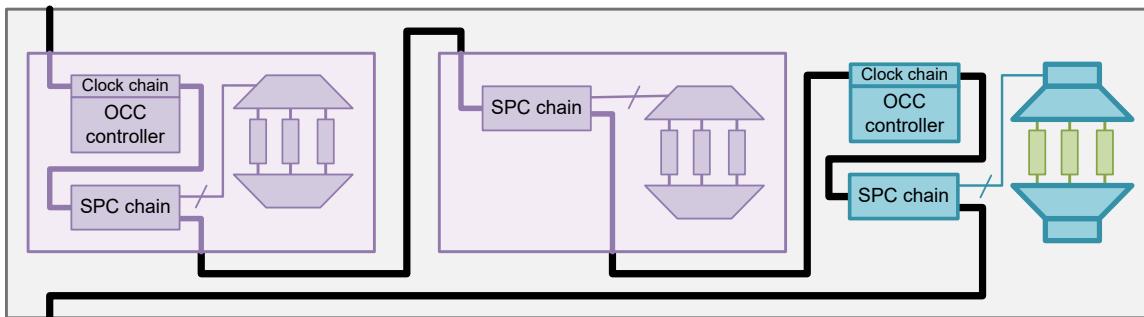
Integrating Cores With Shift Power Groups in Hierarchical Flows

This topic describes how to integrate cores with shift power groups.

Configuring the Control Chain for Shift Power Groups Cores

When you integrate cores that use shift power groups, you must define a top-level external control chain that includes all core-level and top-level clock chains and/or SPC chains, as shown in [Figure 293](#).

Figure 293 External Control Chain in a Shift Power Groups Design



Use the `set_scan_path` command to define the top-level external control chain as follows:

- If any core-level or top-level clock chains exist or will be inserted, then define the external chain using the `-class occ` option.
- If only core-level or top-level SPC chains exist or will be inserted, then define the external chain using the `-class spc` option.
- All core-level clock chains and SPC chains *must be explicitly included in the specification* using the `-include_elements` option. They are not automatically included.
- All top-level clock chains *must be explicitly included in the specification* using the `-include_elements` option. They are not automatically included.
- Top-level SPC chains *are automatically included* in the external chain.

The following example includes core-level clock chains and SPC chains along with top-level clock chains and SPC chains:

```
set_scan_path clock_chain -class occ \
    -include_elements { \
        core1/SPC \
        core2/SPC \
        coreOCC1/OCC \
        coreOCC2/OCC \
        snps_clk_chain_2/clock_chain} \
    -complete_true \
    -scan_data_in OCC_SI \
    -scan_data_out OCC_SO \
    -test_mode all
# (the top-level SPC chain is automatically included)
```

If you concatenate external control chains from pipelined cores, those cores must be created with beginning and ending retiming registers to avoid edge-related concatenation issues at the top level. See the retiming register information in [Using Shift Power Groups With Other DFT Features on page 702](#).

Connecting Core-Level Shift Power Disable Signals

When integrating cores that contain shift power groups, you must manually hook up core-level shift power disable signals to a top-level disable signal.

You can use one of the following methods:

- Include preexisting connections to the cores in your top-level RTL.
- Use ECO commands, such as `disconnect_net` and `connect_pin`, to make the connections to the cores.

You can share a single disable signal or use multiple disable signals.

All shift power disable signals must be *de-asserted* (set to logic 0) to *enable* the shift power logic. The DFT-created disable signal for a top-level codec is already de-asserted in the SPF. Additional disable signals must be manually de-asserted by defining constant signals on them. For example,

```
dc_shell> set_dft_signal -view existing_dft -type Constant \
    -port SPC_CORE_DISABLE* -active_state 0
```

Configuring Shift Power Groups for a Top-Level Codec

If you are implementing a top-level codec, you must configure shift power groups for that codec using the pertinent options of the `set_scan_compression_configuration` command. For more information, see [Configuring Shift Power Groups on page 698](#).

Configuring Shift Power Groups in TestMAX ATPG

Use the following commands in TestMAX ATPG to configure ATPG use of the shift power groups hardware:

```
DRC_T> set_drc -spc_chain SPC_chain_name
DRC_T> set_atpg -shift_controller_peak probability_value
```

SPC_chain_name is the name of the scan path that contains the SPC chain.

probability_value is the maximum percentage of scan cells that can switch in a shift cycle. TestMAX ATPG rejects patterns that exceed this switching percentage.

As described in [Scan-Enable Signal Requirements for Shift Power Groups on page 698](#), you must also constrain the scan-enable signal used by the shift-power logic to be de-asserted during capture. For example,

```
DRC_T> add_pi_constraints 0 SE_port ;# signal is active-high
```

The STIL protocol file (SPF) created by the TestMAX DFT tool enables shift power groups by default. When enabled, you must configure the feature with the preceding commands, otherwise the compressed scan chains will fail DRC due to chain blockages.

Alternatively, you can assert the shift power disable signal, in which case the DFTMAX codec degenerates to a non-shift-power codec and no shift power configuration commands are needed.

Using Shift Power Groups With Other DFT Features

The shift power groups feature interacts with other DFT features as follows:

- Multiple test modes

You can use shift power groups with multiple test modes, including multiple DFTMAX compression modes. Configure the SPC chain in each DFTMAX compression mode. See [Per-Test-Mode Scan Configuration Commands on page 678](#) for supported options.

The control chain must be external only in DFTMAX compression modes. If desired, you can use the `-test_mode` option of the `set_scan_path` specification to limit the external chain specification to those modes (instead of `all`); the control chains are incorporated into regular scan chains in other modes.

If shift power groups are used, they must be used in all DFTMAX test modes. You cannot mix codecs with and without shift power groups across test modes.

- DFT partitions

You can use shift power groups with DFT partitions. Configure the SPC chain in each partition that contains a DFTMAX codec. See [Per-Partition Scan Configuration Commands on page 672](#) for supported options.

Note that although SPC chains can be created for multiple partitions, they are all stitched into the single external control chain specified by the `set_scan_path` command.

If shift power groups are used, they must be used in all partitions that contain a DFTMAX codec. You cannot mix codecs with and without shift power groups across partitions.

- Retiming registers

When you enable beginning and/or ending retiming registers, SPC chains (and clock chains) are clocked on the leading clock edge instead of the trailing clock edge. This facilitates control chain concatenation at the top level.

Limitations of Shift Power Groups

Note the following limitations of shift power groups:

- This feature applies only to the compressed scan chains it is configured for. Standard scan modes are unaffected.
- When shift power groups are used, they must be used
 - In all DFTMAX test modes
 - In all codecs in the design (across both cores and DFT partitions)

You cannot mix DFTMAX codecs with and without shift power groups within the same design.

- The shift power control (SPC) chain must be an external (uncompressed) chain, which you explicitly define using the `set_scan_path` command.
- When using OCC controllers, you must use external (uncompressed) clock chains.
- When integrating cores that contains shift power groups,
 - You must manually hook up the core-level shift power disable signal to a top-level shift power disable signal.
 - In the Hybrid integration flow, you must explicitly configure the top-level codec using the `-inputs` and `-outputs` options of the `set_scan_compression_configuration` command. Otherwise, incorrect SPF might be generated.
- The `report_scan_path` command does not report SPC chain information.

In the TestMAX ATPG tool, the following requirement applies:

- You must use the `add_pi_constraints` command to constrain the scan-enable signal to be de-asserted during scan capture.

In the TestMAX Diagnosis tool, the following requirement applies:

- Diagnosis capability is limited. High-resolution diagnostics are not supported when shift power groups are used. Assert the shift power disable signal to generate patterns for high-resolution diagnostics.

During ATPG, the following tasks are not supported when using shift power groups:

- Analyzing X effects or X sources performed during a TestMAX ATPG simulation
- Comparing simulation results from either a VCD simulation file, the internal patterns from the fast-sequential simulator, or the internal patterns from the full-sequential simulator

- Reporting total (cumulative) power data with the `report_power` command after performing multiple (incremental) ATPG runs
- Saving patterns and fault lists to files at a specified checkpoint interval during ATPG pattern generation
- Saving a GZIP-compressed parallel pattern set that can be simulated during the ATPG process
- Assigning ATPG constraints during an IDDQ measure strobe when the IDDQ fault model is selected

Forcing a Compressor With Full Diagnostic Capabilities

R10 DRC violations indicate that two or more compressed scan chains share the same XOR compression signature at the scan outputs. As a result, a single fault detected at a scan output cannot be uniquely mapped back through the compression logic to a specific scan chain during diagnosis. This is also called *aliasing*.

To force DFT insertion to implement only compressors with full diagnostic capabilities, that is, compressors that do not have any R10 DRC violations, set the following option:

```
dc_shell> set_scan_compression_configuration -force_diagnosis true
```

Note that this option does not change how the compressor logic is built; it simply causes DFT insertion to stop instead of complete if the compressor would have R10 DRC violations.

[Table 51](#) shows the maximum number of compressed scan chains that can be built for a given set of scan-out pins without an R10 DRC violation. Note that these limits are lower than the maximum upper limits shown in [Table 52 on page 737](#).

Table 51 Compressed Scan Chain Limits for Avoiding R10 DRC Violations

Number of scan-out pins	Maximum number of chains
2	3
3	7
4	15
5	31
6	63
7	127

Number of scan-out pins	Maximum number of chains
8	255
9	510
10	1012
11	1980
12	3796
13	7098
14	12910
15	22818
16	32000

For P scanout pins, the maximum number of chains N is computed as follows:

$$N(P) = \sum_{k=1}^{\min(P,8)} C_k^P$$

where the number of k -combinations C is computed by

$$C_k^P = \frac{P!}{k!(P-k)!}$$

If you exceed these limits when the `-force_diagnosis` option is set to `true`, DFT insertion stops with a TEST-1603 message:

Warning: The compressor generated might have lower diagnostics precision.
(TEST-1603)
Information: Scan routing is not complete. Signals 'serial' or
`scan_enables`' need to be routed. (TEST-899)
Information: DFT insertion was not successful. There were unrecoverable
processing errors. (TEST-211)
0

R10 violations can be issued by post-DFT DRC analysis or by DRC in the TestMAX ATPG tool.

See Also

- [SolvNet article 036993, "What Do R10 and R11 DRC Violations Mean?"](#) for more information about R10 violations

Performing Congestion Optimization on Compressed Scan Designs

As the target scan chain compression ratio increases, the number of connections between the codec logic and compressed scan chains increases, and the number of reconfiguration MUX connections increases. This also increases the possibility of routing congestion. The TestMAX DFT tool provides congestion reduction algorithms that use Design Compiler Graphical technology to reduce the congestion introduced by scan compression logic.

To use this feature, you must perform the initial compile with congestion optimization using the `compile_ultra -scan -spg` command. See [Example 105](#).

Example 105 Inserting DFTMAX Scan Compression in a Design Compiler Graphical Flow

```
compile_ultra -scan -spg

set_dft_configuration -scan_compression enable
# ...other DFT configuration settings...

preview_dft
insert_dft
```

In this case, the `preview_dft` and `insert_dft` commands issue the following message:

Running Scan Compression with congestion optimization enabled.

Note:

The compressed scan congestion optimization feature does not work in multiple test-mode flows.

See Also

- [Physical DFT Features in Design Compiler on page 137](#) for more information about reordering and repartitioning optimizations performed for all scan designs

Using AutoFix With Scan Compression

When you insert compressed scan into a design, a test-mode signal is used to enable standard scan or compressed scan. The AutoFix feature requires a separate test-mode signal to enable the testability fixing logic added by DFT Compiler. These test-mode signals cannot be shared, because the AutoFix testability fixes must be activated for both values of the compressed scan test-mode pin, that is, in both the standard scan and compressed scan modes.

When you configure the AutoFix control signal with the `-control_signal` option of the `set_ autofix_ configuration` command, specify a test-mode signal that is dedicated

to enabling the AutoFix logic. If you have not specified any test-mode encodings with the `define_test_mode -encoding` command, the tool will avoid using the AutoFix control signal when it chooses test-mode signals for the test-mode encodings. If you are specifying your own test-mode encodings, you should avoid using the AutoFix control signal in your encodings.

See Also

- [Using AutoFix on page 333](#) for more information about using AutoFix to fix design testability issues

One-Pass DFTMAX Example With AutoFix

The following example performs compressed scan DFT insertion with AutoFix enabled.

Example 106 One-Pass DFTMAX Flow With AutoFix

```
# Define the clocks and asynchronous signals
set_dft_signal -view existing_dft -type ScanMasterClock -timing {45 55} \
    -port clk_st
set_dft_signal -view existing_dft -type ScanMasterClock -timing {55 45} \
    -port clk_st_inv
set_dft_signal -view existing_dft -type Reset -active_state 0 \
    -port rst_st

# Enable DFTMAX compression, AutoFix for clocks, resets, sets, and buses
set_dft_configuration -scan_compression enable \
    -fix_clock enable \
    -fix_reset enable \
    -fix_set enable \
    -fix_bus enable \
    -fix_bidirectional enable \
    -control_points enable \
    -observe_points enable
# Configure DFTMAX compression
set_scan_compression_configuration -minimum_compression 10 \
    -xtolerance high -max_length 20
set_scan_configuration -chain_count 8

# Set the global AutoFix settings to use data clock_autofix_clock_s
# and control TEST_MODE
set_dft_signal -view existing_dft -type ScanMasterClock -timing {45 55} \
    -port clock_autofix_clock_s
set_dft_signal -view spec -type TestMode -port TEST_MODE
set_dft_signal -view spec -type TestData -port clock_autofix_clock_s
set_autofix_configuration -type clock \
    -include_elements [get_object_name [get_cells -hierarchical *]] \
    -control_signal TEST_MODE \
    -test_data clock_autofix_clock_s
```

```

# Define the cells to fix and ports to use for clock AutoFix
set_dft_signal -view existing_dft -type ScanMasterClock -timing {45 55} \
    -port clock_ autofix_clock_sc
set_dft_signal -view spec -type TestData -port clock_ autofix_clock_sc
set_ autofix_element [get_object_name [get_cells dd_c/*]] -type clock \
    -control_signal TEST_MODE \
    -test_data clock_ autofix_clock_sc

# Define the cells to fix and ports to use for reset AutoFix
set_dft_signal -view existing_dft -type Reset -active_state 0 \
    -port reset_ autofix_reset
set_ autofix_element [get_object_name [get_cells -hierarchical *]] -type reset \
    -control_signal TEST_MODE \
    -test_data reset_ autofix_reset

# Set up testpoint insertion using TEST_MODE as the mode port
# and clk_st as the testpoint clock
set_test_point_element -type force_01 -clock_signal clk_s \
    -control_signal TEST_MODE \
    -test_points_per_source_or_sink 1 {dd_c/\o_data_reg[3]/D}
set_test_point_element -type observe -clock_signal clk_st \
    -control_signal TEST_MODE \
    -test_points_per_source_or_sink 1 {dd_c/\o_data_reg[3]/Q}

# Set up port to use for DFTMAX test mode control
set_dft_signal -view spec -type TestMode -port TEST_COMPRESS

# Set up scan enable to use i_rd pin
set_dft_signal -view spec -type ScanEnable -port i_rd

set_dft_insertion_configuration -synthesis_optimization none

## Create the test protocol
create_test_protocol

## Run pre-DFT DRC
dft_drc -verbose

## Preview test structures to be inserted
preview_dft -show all

## Run test insertion
insert_dft

```

One-Pass DFTMAX Example With AutoFix and Multiple Test Modes

The following script example performs compressed scan DFT insertion of multiple test modes with AutoFix enabled.

Example 107 One-Pass DFTMAX Flow With AutoFix, Using Multiple Test Modes

```

read_verilog db/my_design.v
current_design top
uniquify
link

# Create ports to use as test-mode selection for test modes defined
# with define_test_mode
create_port -direction in TM1
create_port -direction in TM2
create_port -direction in TM3

# Test-mode ports must be defined with set_dft_signal -view spec TestMode
# before use in define_test_mode encoding
set_dft_signal -view spec -type TestMode \
    -port {TM1 TM2 TM3} -test_mode all

# Define the test modes for this design
define_test_mode my_base1 -usage scan \
    -encoding {TM1 0 TM2 0 TM3 1}
define_test_mode scan_compression1 -usage scan_compression \
    -encoding {TM1 1 TM2 0 TM3 0}
define_test_mode burn_in -usage scan \
    -encoding {TM1 0 TM2 1 TM3 1}

# Define the clocks and asynchronous signals
set_dft_signal -view existing_dft -type ScanMasterClock \
    -timing {45 55} -port sys_clk -test_mode all
set_dft_signal -view existing_dft -type ScanMasterClock \
    -timing {55 45} -port sys_clk_inv -test_mode all
set_dft_signal -view existing_dft -type Reset -active_state 0 \
    -port sys_reset -test_mode all

# Enable DFTMAX compression, AutoFix for clocks, resets, sets, and buses
set_dft_configuration -scan_compression enable \
    -fix_clock enable \
    -fix_reset enable \
    -fix_set enable \
    -fix_bus enable \
    -fix_bidirectional enable \
    -control_points enable \
    -observe_points enable

# Configure the test modes
set_scan_compression_configuration -minimum_compression 10 \
    -xtolerance high -base_mode my_base1 \
    -test_mode scan_compression1
set_scan_configuration -chain_count 8 -test_mode my_base1
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
    -test_mode burn_in

# Define the cells to fix and ports to use for clock AutoFix

```

Chapter 17: Using DFTMAX Compression Using AutoFix With Scan Compression

```

set_dft_signal -view existing_dft -type ScanMasterClock -timing {45 55} \
  -port clock_autofix_clock_s -test_mode all
set_dft_signal -view spec -type TestData -port clock_autofix_clock_s \
  -test_mode all
set_dft_signal -view spec -type TestMode -port TEST_MODE \
  -test_mode all
set_ autofix_element [get_object_name [get_cells -hierarchical *]] \
  -type clock -control_signal TEST_MODE \
  -test_data clock_autofix_clock_s

# Define the cells to fix and ports to use for reset AutoFix
set_dft_signal -view existing_dft -type Reset -active_state 0 \
  -port reset_autofix_reset -test_mode all
set_ autofix_element [get_object_name [get_cells -hierarchical *]] \
  -type reset -control_signal TEST_MODE \
  -test_data reset_autofix_reset

# Set up testpoint insertion using TEST_MODE as the mode port
# and sys_clk as the testpoint clock
set_test_point_element -type force_01 -clock_signal sys_clk \
  -control_signal TEST_MODE \
  -test_points_per_source_or_sink 1 {dd_c/o_data_reg[3]/D}
set_test_point_element -type observe -clock_signal sys_clk \
  -control_signal TEST_MODE \
  -test_points_per_source_or_sink 1 {dd_c/o_data_reg[3]/Q}

# Set up scan enable to use i_rd pin
set_dft_signal -view spec -type ScanEnable -port i_rd -test_mode all

# Enable rapid scan stitching
set_dft_insertion_configuration -synthesis_optimization none

# Create the test protocol
create_test_protocol

# Run pre-DFT DRC
dft_drc -verbose

# Preview test structures to be inserted
preview_dft -show all

# Generate a report specific to AutoFix
report_ autofix_configuration -type all

# Run insert_dft to insert into design
insert_dft

# List the modes inserted and report the test model
list_test_modes
report_test_mode1

# Run post-DFT DRC
current_test_mode scan_compression1

```

Chapter 17: Using DFTMAX Compression Using AutoFix With Scan Compression

```
report_dft_signal
dft_drc -verbose

current_test_mode my_base1
report_dft_signal
dft_drc -verbose

current_test_mode burn_in
report_dft_signal
dft_drc -verbose

# Write test protocol for use in TestMAX ATPG
write_test_protocol -test_mode scan_compression1 \
    -output stil/10x_xtol_moxie_autofix.stil -names verilog
write_test_protocol -test_mode my_base1 \
    -output stil/10x_xtol_moxie.scan_autofix.stil -names verilog

# Write out the scan inserted design
change_names -rules verilog -hierarchy
write -format verilog -hierarchy \
    -output vg/10x_xtol_moxie_top_scan_autofix.v
write -format ddc -hierarchy \
    -output db/10x_xtol_moxie_top_scan_autofix.ddc
```

18

Hierarchical Adaptive Scan Synthesis

This chapter explains how to run hierarchical adaptive scan synthesis flows. It also describes how to integrate compressed scan cores, standard scan cores, and test-ready, top-level, sequential user-defined logic.

In the hierarchical adaptive scan synthesis (HASS) flow, scan compression logic is placed at the block level, and all cores with scan compression logic are integrated at the chip level. This approach helps reduce the routing congestion prevalent in multimillion-gate designs.

The Hybrid flow is an extension of the HASS flow that provides additional support for compressed scan insertion for user-defined logic. A normal HASS flow supports insertion of standard scan chains for user-defined logic, and the Hybrid flow supports insertion of compressed scan for user-defined logic.

This chapter includes the following topics:

- [The HSS Flow](#)
 - [The HASS Flow](#)
 - [The Hybrid Flow](#)
 - [Using Multiple Test Modes in Hierarchical Flows](#)
 - [Top-Level Integration Script Examples](#)
 - [HASS and Hybrid Flow Limitations](#)
-

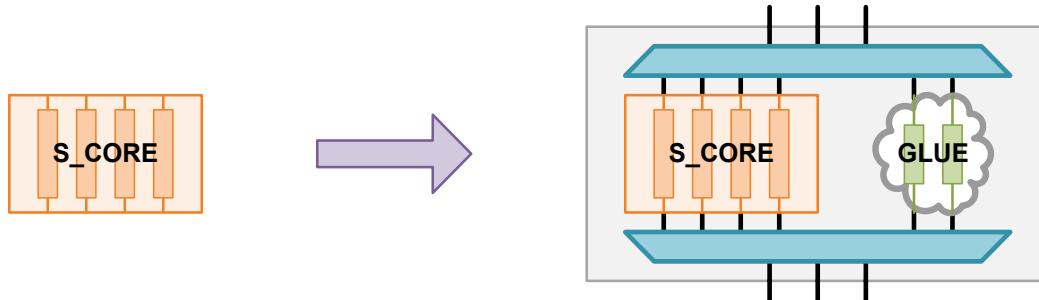
The HSS Flow

You can use the *hierarchical scan synthesis* (HSS) flow to insert scan when one or more standard scan cores are present. The compressed scan HSS flow is similar to the standard scan HSS flow described in [Hierarchical Scan Synthesis on page 124](#), except that scan compression is added around standard scan cores at the top level.

When the TestMAX DFT tool inserts scan compression in the HSS flow, it applies compression to standard scan cores as well as top-level logic. Scan chains inside standard scan cores are treated as scan segments.

[Figure 294](#) shows an example of a compressed scan HSS flow.

Figure 294 The Compressed Scan HSS Flow



The following logic types are supported in the compressed scan HSS flow:

- Standard scan cores

These cores can be represented by the full netlist or a CTL test model. The TestMAX DFT tool incorporates the core-level scan chains into scan compression as scan chain segments. These scan chain segments can be concatenated and rebalanced inside the codec as needed, but they cannot be subdivided into smaller scan chains.

- Test-ready cores that are scan-replaced, but do not yet have scan chains

The tool incorporates the test-ready logic into scan compression, creating compressed scan chains as needed to meet the scan chain requirements.

- Cores that have not yet been scan-replaced

The tool performs scan replacement before applying scan compression.

- Top-level glue logic that might or might not be test-ready

The tool performs scan replacement if needed, then it applies scan compression.

The compressed scan HSS flow is automatically applied whenever one or more standard scan cores are present and scan compression is enabled with the `set_dft_configuration` command:

```
dc_shell> set_dft_configuration -scan_compression enable
```

You do not need to specify any additional integration options with the `set_scan_compression_configuration` command to enable the compressed scan HSS flow.

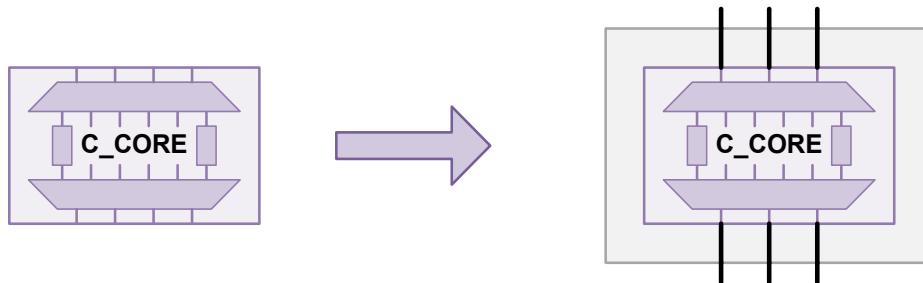
The HASS Flow

Sometimes a design is too large for scan compression to be inserted using a top-down flat or HSS flow. Or, a design might include some reused compressed scan cores. In these cases, a bottom-up hierarchical flow is needed to assemble compressed scan cores at the top level and perform core integration.

You can use the *hierarchical adaptive scan synthesis* (HASS) flow to perform top-level core integration with one or more compressed scan cores. The HASS flow is similar to the hierarchical scan synthesis (HSS) flow described in [Hierarchical Scan Synthesis on page 124](#), except that the HASS flow adds support for compressed scan cores.

[Figure 295](#) shows an example of the HASS integration flow.

Figure 295 HASS Integration of a Compressed Scan Core



In the HASS flow, no scan compression logic is added at the top level. The TestMAX DFT tool promotes the scan connections of the compressed scan cores to top-level scan connections.

A compressed scan core contains scan chain logic that can operate in both standard scan and compressed scan modes. A *standard scan* core contains scan chains that only operate in standard scan mode. When a mix of compressed scan and standard scan cores are integrated at the top level in the HASS flow, the test modes operate as follows:

- In standard scan mode, all cores operate in their standard scan modes.
- In compressed scan mode, compressed scan cores operate in their compressed scan mode, while the standard scan cores continue to operate in standard scan mode.

Preparing Cores in the HASS Flow

In the HASS flow, at least one top-level compressed scan core is required for top-level integration. Additional cores, containing compressed scan or standard scan logic, can also be provided.

[Example 108](#) shows a typical compressed scan insertion script used in the HASS flow.

Example 108 Typical Core-Level Compressed Scan Insertion Script

```
read_ddc core1_test_ready.ddc
current_design core1
set_scan_configuration -chain_count 10
set_dft_configuration -scan_compression enable
set_dft_signal -view existing_dft -port CLK -type \
    ScanClock -timing {45 55}

create_test_protocol
dft_drc
preview_dft
insert_dft

current_test_mode Internal_scan
dft_drc
current_test_mode ScanCompression_mode
dft_drc

write -format ddc -hierarchy -output core1.ddc
write_test_model -format ddc -output core1.ctlddc
change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output core1.v
```

Each core must have CTL test model information so that the tool can perform top-level integration. If the block fits in memory during top-level integration, you can use the `write` command to write a design .ddc file that contains the full design netlist as well as the CTL test model information:

```
dc_shell> write -format ddc -hierarchy -output design_name.ddc
```

If the block is large, you can use the `write_test_model` command to write out a test-model-only .ddc file that contains the CTL test model along with an interface-only representation of the core that allows the test model to be linked at the top level:

```
dc_shell> write_test_model -format ddc -output design_name.ctlddc
```

You can use either format for standard scan and compressed scan cores in the HASS flow.

HASS Integration of Compressed Scan Cores

To enable the HASS flow at the top level, use the following commands:

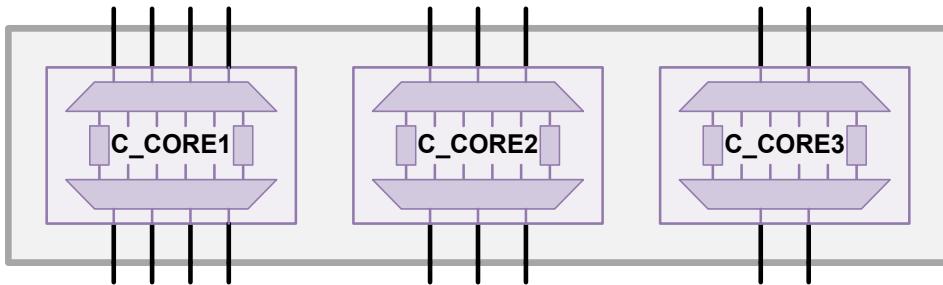
```
dc_shell> set_dft_configuration -scan_compression enable
dc_shell> set_scan_compression_configuration -integration_only true
```

The first command enables scan compression, and the second command enables the HASS flow.

The HASS flow does not concatenate or rebalance the scan chain connections of compressed scan cores at the top level. Each core-level scan chain is promoted to a top-level scan chain with dedicated scan pin connections in both the standard scan and compressed scan modes. The top-level scan count, and therefore the scan pin budget, is determined by the number of scan chains in the compressed scan cores.

[Figure 296](#) shows the results from the HASS flow when three compressed scan cores are present. All nine codec connections are promoted to nine top-level scan-in and scan-out connections. No other scan chain count is possible.

Figure 296 HASS Integration of Three Compressed Scan Cores



If you issue a `set_scan_configuration -chain_count` command requesting fewer scan chains, the `preview_dft` and `insert_dft` commands issue the following warning:

```
Warning: Cells with 8 new incompatible clock domains have not been
assigned to scan chains. Cannot honor -chain_count specification of 7.
Try using set_scan_configuration -clock_mixing mix_edges or
-clock_mixing
mix_clocks. (TEST-355)
```

If you issue a `set_scan_configuration -chain_count` command requesting more scan chains, the `preview_dft` and `insert_dft` commands issue the following warning:

```
Warning: Only 8 scan chain elements are free. Cannot honor -chain_count
specification of 9. (TEST-348)
```

When only compressed scan cores exist, you do not need to specify a chain count with the `set_scan_configuration -chain_count` command. However, you can specify the expected chain count so that the tool verifies the actual number of scan chains against the expected number.

HASS Integration of Additional Uncompressed Scan Logic

The HASS flow also supports the presence of uncompressed scan logic in addition to the compressed scan blocks by creating standard scan chains. The following uncompressed logic types are supported during top-level integration:

- Standard scan cores

These cores are represented by the full netlist or a CTL test model. The TestMAX DFT tool incorporates the core-level scan chains into top-level scan chains as scan chain segments. These scan chain segments can be concatenated and rebalanced at the top level as needed, but they cannot be subdivided into smaller scan chains.

- Test-ready cores that are scan-replaced, but do not yet have scan chains

The tool architects standard scan chains that are active in both the standard scan and compressed scan modes.

- Cores that are not scan-replaced

The tool performs scan replacement, then architect standard scan chains that are active in both the standard scan and compressed scan modes.

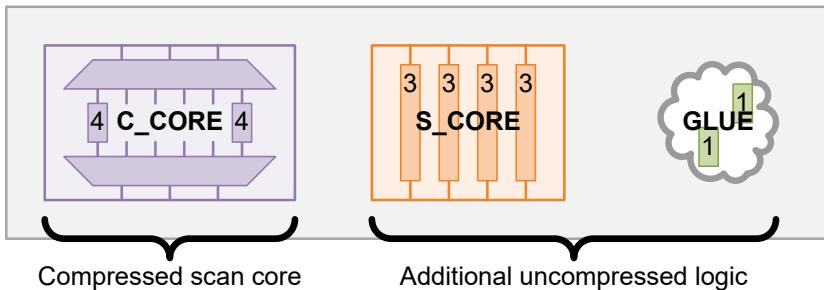
- Top-level glue logic that might or might not be test-ready

The tool performs scan replacement if needed, then architect standard scan chains that are active in both the standard scan and compressed scan modes.

The scan chain connections for compressed cores are always promoted to top-level scan chain connections. The scan architecture behavior for the additional uncompressed logic depends on whether a target chain count is specified with the `set_scan_configuration -chain_count` command.

[Figure 297](#) shows a top-level design example with a compressed scan core, a standard scan core, and some top-level glue logic. The compressed scan core C_CORE contains 24 flip-flops divided into 6 compressed scan chains of 4 flip-flops each, with 3 scan-in and scan-out pins. The standard scan core S_CORE contains 12 flip-flops, split into 4 scan chains. The top-level glue logic GLUE contains 2 scan-replaced flip-flops with no scan chains.

Figure 297 Design Example Before HASS Integration

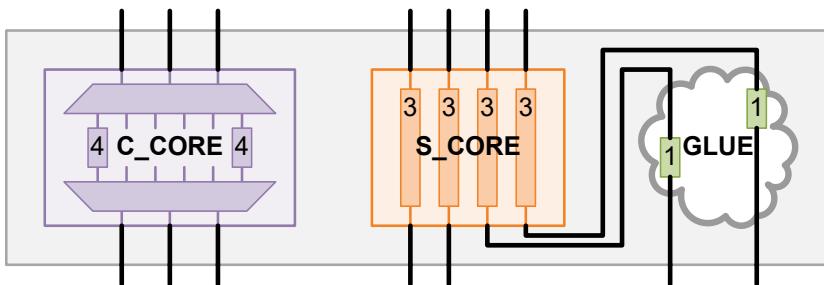


When no target chain count is specified, the following goals are used for scan architecture of the additional logic:

- For the compressed scan mode of operation, the tool attempts to architect scan chains that do not exceed the longest compressed scan chain inside a compressed scan core. This preserves the test compression characteristics of the compressed scan cores.
- For the standard scan mode of operation, the tool follows the default rules of scan chain architecture where the minimum number of scan chains is built that meet any applied scan architecture requirements.

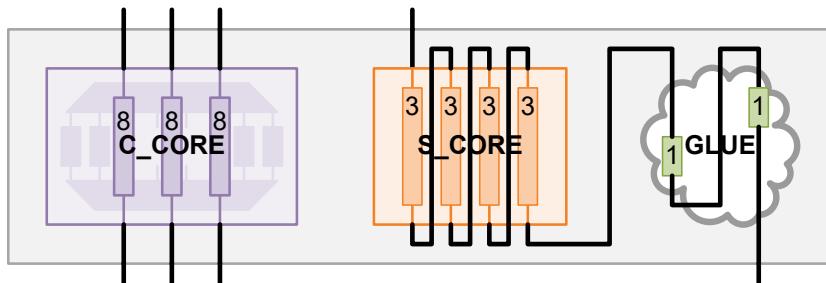
[Figure 298](#) shows the resulting HASS top-level integration results operating in compressed scan mode. The codec scan chain connections from C_CORE are promoted directly to top-level connections. For the remaining uncompressed logic, standard scan chains are architected so that the chain length of the compressed core is not exceeded.

Figure 298 Compressed Scan Mode After HASS Integration With No Chain Count Specified



[Figure 299](#) shows the resulting HASS top-level integration results operating in standard scan mode. The standard scan chain connections from C_CORE are promoted directly to top-level connections. For the remaining uncompressed logic, a single standard scan chain is created that includes joined scan segments from S_CORE and the scan flip-flops from the GLUE logic.

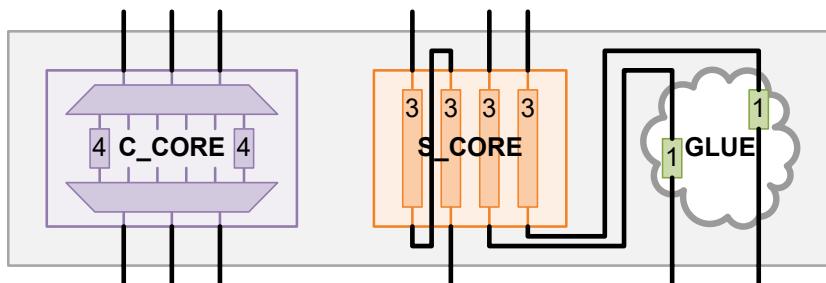
Figure 299 Standard Scan Mode After HASS Integration With No Chain Count Specified



Note that when a chain count is not specified, the scan chain counts might be different between the standard scan and compressed scan modes.

When a scan chain count is specified with the `set_scan_configuration -chain_count` command, the tool attempts to honor the specified chain count for both the standard scan and compressed scan modes. [Figure 300](#) shows the HASS top-level integration results for the compressed scan mode when the `set_scan_configuration -chain_count 6` command is specified. Because the specified scan chain count applies to both the standard scan and compressed scan modes, the scan architecture of the uncompressed logic is the same in both modes.

Figure 300 Compressed Scan Mode After HASS Integration With -chain_count 6 Specified



If you issue a `set_scan_configuration -chain_count` command requesting fewer scan chains than is possible, the `preview_dft` and `insert_dft` commands issue the following warning:

```
Warning: Cells with 4 new incompatible clock domains have not been
assigned to scan chains. Cannot honor -chain_count specification of 3.
Try using set_scan_configuration -clock_mixing mix_edges or
-clock_mixing
mix_clocks. (TEST-355)
```

If you issue a `set_scan_configuration -chain_count` command requesting more scan chains than is possible, the `preview_dft` and `insert_dft` commands issue two warnings, one for the total set of scan chains and one for the uncompressed logic chains:

Warning: Only 9 scan chain elements are free. Cannot honor `-chain_count` specification of 10. (TEST-348)

Warning: Only 6 scan chain elements are free. Cannot honor `-chain_count` specification of 7. (TEST-348)

Since compressed scan mode chains are usually short, you should take care to manage the relative lengths of standard scan and compressed scan chains at the top level. This might require management of the scan chain lengths of any standard scan cores, as well as providing an adequate scan pin budget at the top level to avoid excessive standard scan chain concatenation. If further reduction in chain length is needed, you can use the Hybrid flow. For more information, see [The Hybrid Flow on page 720](#).

After top-level integration, you can perform DRC of the standard scan mode using the `dft_drc` command. However, the tool does not support DRC of the top-level compressed scan mode. DRC checking for the compressed scan mode is performed in the TestMAX ATPG tool.

In the HASS flow, a single test-mode pin that is shared across all scan cores is required for selecting standard scan or compressed scan mode. Complex test-mode encodings are not supported.

The Hybrid Flow

In the HASS flow, existing compressed scan core chains are promoted to top-level chains and standard scan is used to access all other logic, including top-level user-defined logic. If there is a large amount of top-level logic, an imbalance between the compressed scan chain lengths and standard scan chain lengths might result. This can reduce the effective amount of test compression.

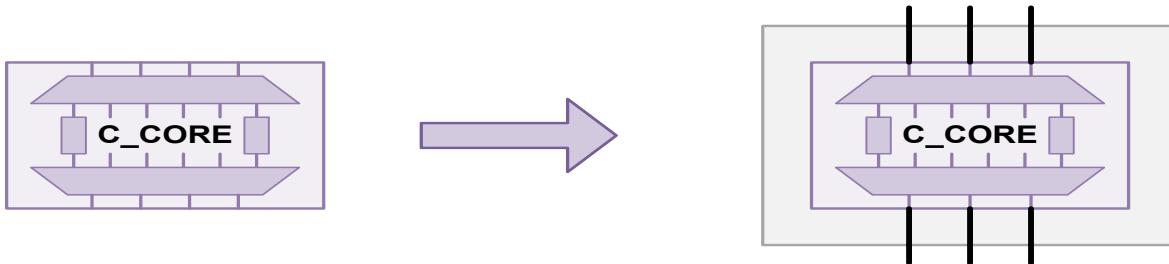
You can use the Hybrid flow to reduce this chain length imbalance. The Hybrid flow is a combination of the compressed scan HSS flow and the HASS flow. The Hybrid flow operates as follows:

- For compressed scan cores, the tool promotes core-level scan connections to top-level connections, as in the compressed scan HSS flow.
- For uncompressed logic, the tool applies top-level scan compression, as in the HASS flow.

The Hybrid flow supports the same uncompressed logic types as the HASS flow. For a list of these logic types, see [HASS Integration of Additional Uncompressed Scan Logic on page 717](#).

[Figure 301](#) shows an example of the Hybrid integration flow.

Figure 301 Hybrid Integration of a Compressed Scan Core



Performing Top-Level Hybrid Integration

To enable the Hybrid flow at the top level, use the following commands:

```
dc_shell> set_dft_configuration -scan_compression enable  
dc_shell> set_scan_compression_configuration -hybrid true
```

The first command enables scan compression, and the second command enables the Hybrid flow.

To configure Hybrid integration, you must supply the following information:

- Specify the total top-level scan chain count with the `set_scan_configuration -chain_count` command.
- Specify the number of compressed scan chains for the new top-level codec with the `set_scan_compression_configuration -chain_count` command, or the maximum compressed scan chain length with the `set_scan_compression_configuration -max_length` command.

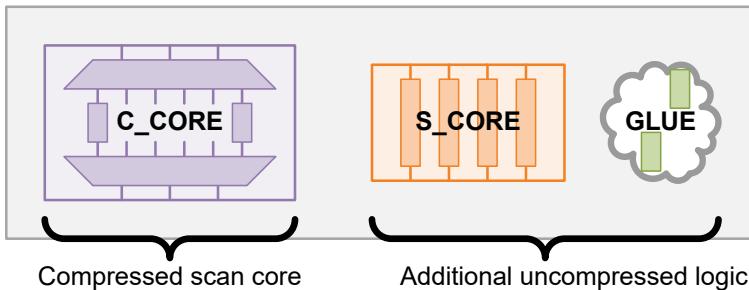
Note:

This value does not include the compressed chains in any existing compressed scan cores.

Just as with the HASS flow, the scan chain connections of compressed scan cores are promoted to top-level scan chain connections. When you specify the total top-level chain count with the `set_scan_configuration -chain_count` command, this value includes these promoted compressed scan core connections. However, the compressed chain count specified with the `set_scan_compression_configuration -chain_count` command applies only to the logic included in top-level compressed scan insertion.

Figure 302 shows a top-level design that contains a compressed scan core named C_CORE, a standard scan core named S_CORE, and some top-level logic named GLUE.

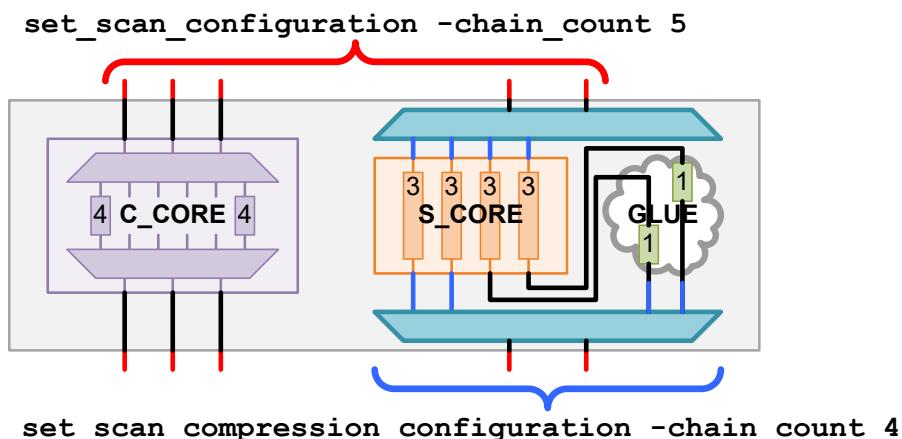
Figure 302 Top Level Before Applying Hybrid Integration



[Figure 303](#) shows the resulting Hybrid top-level design operating in compressed scan mode. Note the following features:

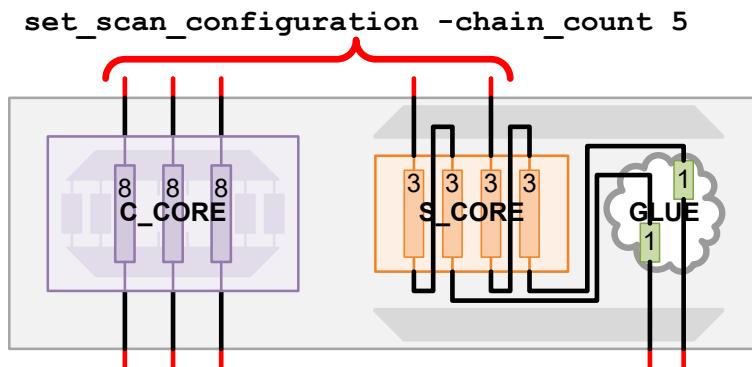
- The tool determines the number of inputs and outputs on the new top-level codec by taking the top-level chain count and subtracting the codec connections of the existing compressed scan cores. In [Figure 303](#), C_CORE uses three of the five total top-level scan chain connections. The remaining two chain connections determine the width of the new top-level codec.
- The tool architects four compressed scan chains in the uncompressed logic, then compresses these chains with the new top-level codec.

Figure 303 Compressed Scan Mode Operation After Hybrid Integration



[Figure 304](#) shows the HASS top-level integration results operating in standard scan mode. The TestMAX DFT tool architects two standard scan chains in the uncompressed logic, and connects them to the two available top-level scan pins.

Figure 304 Standard Scan Mode Operation After Hybrid Integration



Performing Top-Level Hybrid Integration with Partitions

By default, Hybrid integration creates a single codec for all scan logic except the existing compressed scan cores. You can use the DFT partition feature to create multiple codecs at the top level. This can help reduce routing congestion. For more information about defining DFT partitions to create multiple codecs, see [Chapter 17, Using DFTMAX Compression.](#)

You can use the `define_dft_partition` command to define an additional partition and specify the cells and designs to be placed in that partition. All cores and logic not explicitly assigned to a user-defined partition remain in the default partition, named `default_partition`.

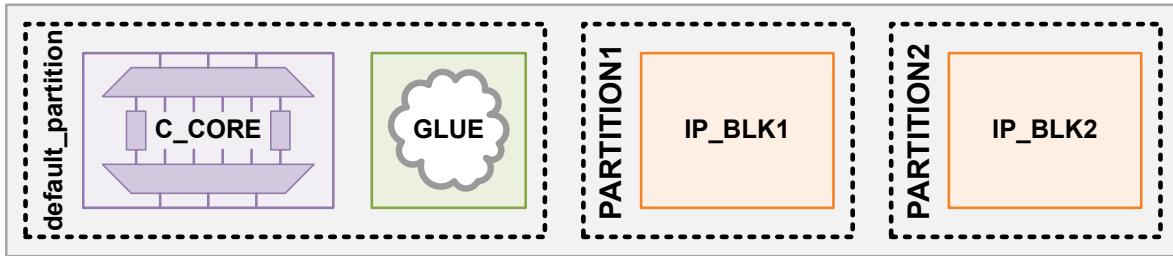
You can use the partition feature to perform compressed scan insertion on cores which have not yet been scan-inserted. This allows you to defer compressed scan insertion of a core to the top level integration run, where the scan compression configuration can be adjusted and rerun as needed.

Consider the following scenario:

- Compressed scan core C_CORE already has compressed scan inserted, and only requires integration at the top level.
- Blocks IP_BLK1 and IP_BLK2 have been synthesized with a test-ready compile, but they do not yet have scan chains. Each of these blocks should have its own codec inserted within its hierarchy.
- Some top-level glue logic exists, contained in the GLUE block. This glue logic should have its own codec inserted at the top level.

The required partitions can be defined as shown in [Figure 305](#).

Figure 305 Hybrid Partition Definitions Before Top-Level Integration



[Example 109](#) shows the commands used for partition creation and scan configuration. The `set_dft_location` command is used to place the codecs inside IP_BLK1 and IP_BLK2.

Example 109 Defining Partitions in a Hybrid Flow

```
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -hybrid true

define_dft_partition PARTITION1 -include {IP_BLK1}
define_dft_partition PARTITION2 -include {IP_BLK2}

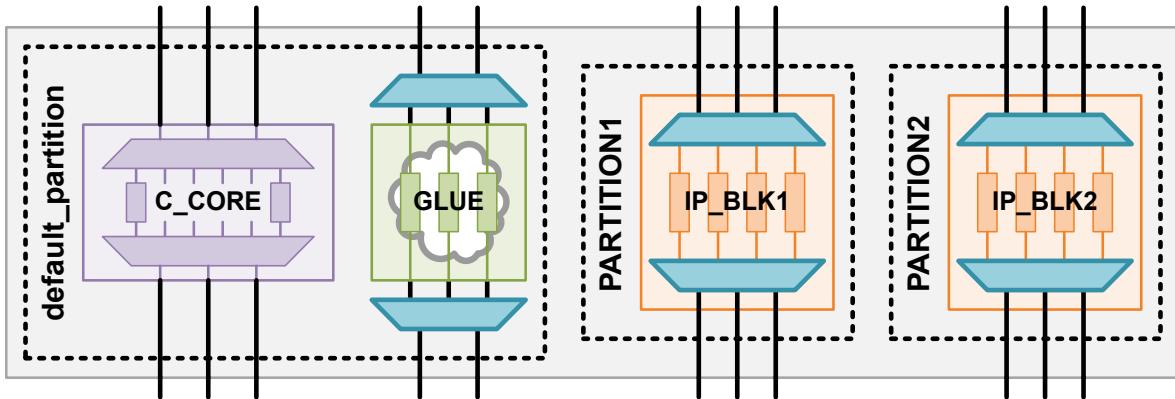
current_dft_partition PARTITION1
set_scan_configuration -chain_count 3
set_scan_compression_configuration -chain_count 4
set_dft_location -include CODEC IP_BLK1

current_dft_partition PARTITION2
set_scan_configuration -chain_count 3
set_scan_compression_configuration -chain_count 4
set_dft_location -include CODEC IP_BLK2

current_dft_partition default_partition
set_scan_configuration -chain_count 5
set_scan_compression_configuration -chain_count 3
```

[Figure 306](#) shows the resulting codec configurations after the `insert_dft` command is used.

Figure 306 Hybrid Partition Definitions After Top-Level Integration



The normal Hybrid scan architecture rules apply inside each partition. For example:

- A top-level chain count of five is specified for the default partition. Since C_CORE already has three scan chain connections, two scan chain connections are allocated to the codec inserted in the GLUE block.
- A compressed chain count of three is specified for the default partition. Three compressed chains are created inside the GLUE block during compressed scan insertion.
- Top-level chain counts and compressed chain counts applied to other partitions only apply to the codec insertion for logic within those partitions.

See Also

- [Top-Down Flat Compressed Scan Flow With DFT Partitions on page 668](#) for more information about defining DFT partitions

Using Multiple Test Modes in Hierarchical Flows

In hierarchical scan compression flows with multiple test modes, DFT cores have test modes that must be incorporated into top-level test modes during core integration. This process is explained in the following topics:

- [Default Core-Level Test Mode Assignment](#)
- [User-Defined Core-Level Test Mode Scheduling](#)

See Also

- [Multiple Test Modes on page 357](#) for more information about user-defined test modes

Default Core-Level Test Mode Assignment

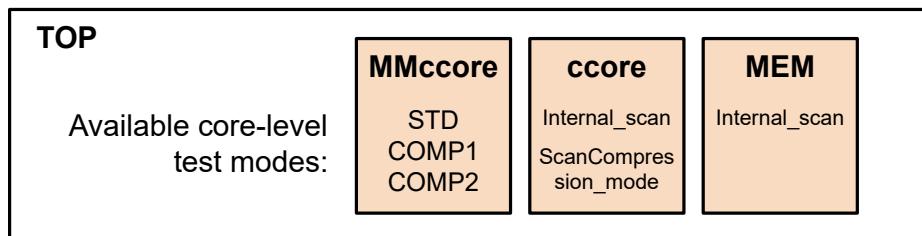
By default, the TestMAX DFT tool creates as many top-level test modes as needed to accommodate all of the core-level test modes present during core integration.

The relationship between core-level and top-level test modes is determined by test mode name. In scan compression flows, the following additional rule applies:

- A compressed scan core with default test mode names (Internal_scan and ScanCompression_mode) is always active. It is assigned to
 - Internal_scan mode in top-level test modes defined with the `scan` usage
 - ScanCompression_mode mode in top-level test modes defined with the `scan_compression` usage

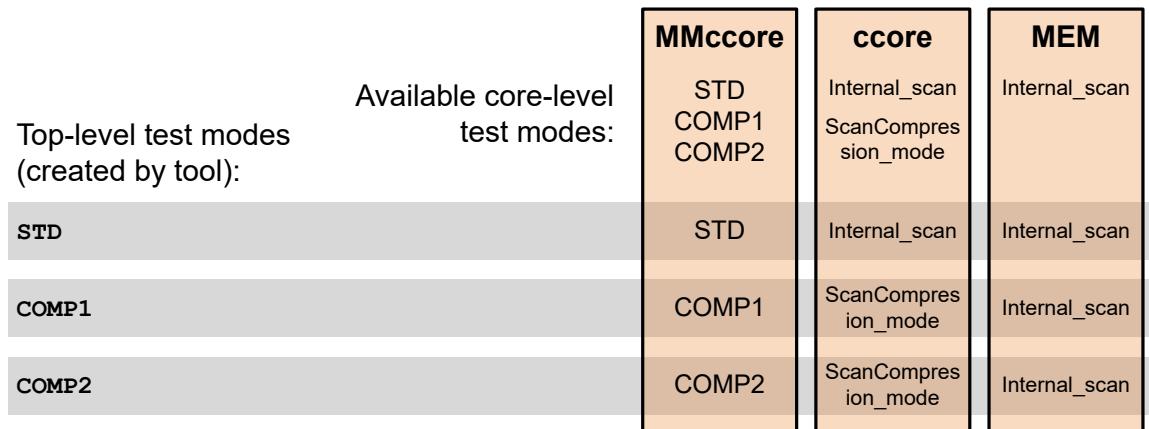
[Figure 307](#) shows an example with three compressed cores instantiated in a top-level design. The example includes a compressed scan core with multiple user-defined test modes, a compressed scan core with the default standard and compressed scan modes, and a scannable memory with a single Internal_scan test mode.

Figure 307 Three Cores With Different Test Modes Instantiated in a Top-Level Design



For this example, [Figure 308](#) shows the top-level test modes created by the tool during core integration. Each column represents a core, each row represents a top-level test mode, and the intersections of the columns and rows show the core-level test mode used for that top-level test mode.

Figure 308 Top-Level Test Modes With Default Test-Mode Assignments



After DFT insertion, the `list_test_modes` command reports the core-level test modes used in each of the top-level test modes. For the previous example, the `list_test_modes` command reports the core-level test modes as shown in [Example 110](#).

Example 110 Top-Level Test Mode Report for Default Core-Level Test-Mode Assignment

```

Control signal value - Integration Test Mode
Core Instance - Test Mode
-----
Name: STD
Type: InternalTest
Focus:
Core ccore in Internal_scan mode
Core MMccore in STD mode
Core mem in Internal_scan mode

Name: COMP1
Type: InternalTest
Focus:
Core ccore in ScanCompression_mode mode
Core MMccore in COMP1 mode
Core mem in Internal_scan mode

Name: COMP2
Type: InternalTest
Focus:
Core ccore in ScanCompression_mode mode
Core MMccore in COMP2 mode
Core mem in Internal_scan mode

Name: Mission_mode
Type: Normal

```

User-Defined Core-Level Test Mode Scheduling

At the top level, you can override the default name-based association of core-level test modes. This is known as *test mode scheduling*. To do this, use the `-target` option of the `define_test_mode` command:

```
define_test_mode test_mode_name
    -target {core1:mode1 [core2:mode2 ...] [current_design_name]}
```

The `-target` option specifies a list of core and test-mode pairs to use for the top-level test mode being defined; each pair consists of a core instance name and a core test-mode name separated by a colon (:). In compressed scan flows, the list can also contain the name of the current design to specify that the top-level logic should be active and tested.

When you use the `-target` option in a compressed scan flow, the following rules apply:

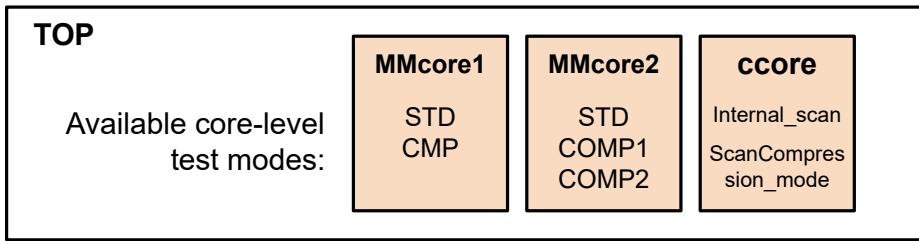
- All test modes must be defined with the `define_test_mode` command; no test modes are automatically created.
- All test mode definitions must use the `-target` option.
- Targeted cores (included in the target list) are placed in their targeted mode.
- If a core is targeted in some test modes but not others, it is inactive in the test modes where it is not targeted. This is known as *sparse targeting*. (To completely exclude a core from all top-level test modes, use the `-exclude_elements` option of the `set_scan_configuration` command.)
- Untargeted cores (not included in any target list) are tested in top-level modes where the top-level logic is tested:
 - In top-level standard scan modes, they are placed in standard scan mode.
 - In top-level compressed scan modes, they are placed in compressed scan mode (for compressed scan cores) or standard scan mode (for standard scan cores).
 - They are placed in the first available such mode defined inside the core's test model.
- The top-level logic, which is all scannable logic outside DFT cores, is only active and tested when targeted.

Note:

The `-target` option has some limitations when used in compressed scan core integration modes. See [HASS and Hybrid Flow Limitations on page 733](#).

[Figure 309](#) shows an example with three compressed cores instantiated in a top-level design. The example includes a compressed scan core with the default test modes and two compressed scan cores with multiple user-defined test modes.

Figure 309 Three Cores With Different Test Modes Instantiated in a Top-Level Design



The `-target` option allows the core-level modes to be scheduled to resolve the difference in user-defined test mode names, as shown in [Example 111](#).

Example 111 Specifying User-Defined Test Mode Assignments

```
# top-level test mode definitions
define_test_mode STD -usage scan \
    -target {MMcore1:STD top}
define_test_mode COMP1 -usage scan_compression \
    -target {MMcore1:COMP top}
define_test_mode COMP2 -usage scan_compression \
    -target {MMcore1:COMP top}
```

For this example, [Figure 310](#) shows the top-level test modes created by the tool during core integration. Each column represents a core, each row represents a top-level test mode, and the intersections of the columns and rows show the core-level test mode used for that top-level test mode. In addition, the “top” column shows when the top-level logic is active and tested. Blue columns indicate logic scheduled by the `-target` option.

Figure 310 Top-Level Test Modes With User-Defined Test-Mode Scheduling

Top-level test modes (created by tool):	Available core-level test modes:			
	MMcore1	top	MMcore2	ccore
define_test_mode STD -usage scan \ -target {MMcore1:STD top}	STD CMP	STD (Active and tested)	STD	Internal_scan
define_test_mode COMP1 \ -usage scan_compression \ -target {MMcore1:COMP top}	CMP	CMP (Active and tested)	COMP1	ScanCompression_mode
define_test_mode COMP2 \ -usage scan_compression \ -target {MMcore1:COMP top}	CMP	CMP (Active and tested)	COMP1	ScanCompression_mode

After DFT insertion, the `list_test_modes` command reports the core-level test modes as shown in [Example 112](#).

Example 112 Top-Level Test Mode Report for User-Defined Test-Mode Scheduling

```

Control signal value - Integration Test Mode
Core Instance - Test Mode
-----
Name: STD
Type: InternalTest
Focus:
Core ccore in Internal_scan mode
Core MMcore1 in STD mode
Core MMcore2 in STD mode

Name: COMP1
Type: InternalTest
Focus:
Core ccore in ScanCompression_mode mode
Core MMcore1 in CMP mode
Core MMcore2 in COMP1 mode

Name: COMP2
Type: InternalTest
Focus:
Core ccore in ScanCompression_mode mode
Core MMcore1 in CMP mode
Core MMcore2 in COMP1 mode

Name: Mission_mode
Type: Normal

```

You can use sparse targeting to target a core and/or the top-level logic in some modes but not others. Sparse targeting is typically used in core wrapping flows where cores can be placed into inward-facing or outward-facing test modes. For more information, see [Scheduling Wrapped Cores on page 505](#). Sparse targeting should be used carefully with unwrapped cores because inactive logic can drive X values into active logic, and the outputs of active logic cannot be captured by inactive logic.

Top-Level Integration Script Examples

This topic provides the following script examples for the HASS and Hybrid flows:

- [Typical HASS Flow Script](#)
- [Typical Hybrid Flow Script](#)
- [Hybrid Flow Script With Multiple Test Modes](#)

Typical HASS Flow Script

HASS integration takes place at the top level. [Example 113](#) shows a typical top-level script.

Example 113 Top-Level Script for HASS Flow

```
read_verilog TOP.v
read_test_model sub1.ctlddc
read_test_model sub2.ctlddc
current_design TOP
link

set_dft_configuration -scan_compression enable
set_scan_compression_configuration -integration_only true
dft_drc
preview_dft
insert_dft
write_test_protocol -test_mode ScanCompression_mode -output comp.spf
write_test_protocol -test_mode Internal_scan -output scan.spf
```

Note:

Post-DFT DRC in scan compression mode is not supported at the top level.

Typical Hybrid Flow Script

[Example 114](#) shows a typical script for top-level integration for the Hybrid flow.

Example 114 Script for Top-Level Integration in the Hybrid Flow

```
read_verilog my_top_test_ready.v
read_test_model ddc/core1.ctlddc
read_test_model ddc/core2.ctlddc

current_design my_top
link

set_dft_configuration -scan_compression enable
set_scan_compression_configuration -hybrid true

set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port CLK
set_dft_signal -view existing_dft -type constant \
    -active_state 1 -port my_test_mode_port
set_dft_insertion_configuration \
    -synthesis_optimization none -preserve_design_name true

create_test_protocol
dft_drc
preview_dft -show all
```

```

insert_dft
current_test_mode Internal_scan
dft_drc -verbose

remove_design core1
remove_design core2

change_names -rules verilog -hierarchy

write -format verilog -hierarchy -output vg/top_scan.v
write -format ddc -hierarchy -output ddc/top_scan.ddc
write_test_protocol -test_mode ScanCompression_mode \
    -output stil/top_moxie.stil -names verilog
write_test_protocol -test_mode Internal_scan \
    -output stil/scan.stil -names verilog

```

Hybrid Flow Script With Multiple Test Modes

[Example 115](#) shows a typical script for the top-level integration with multiple test modes using the Hybrid flow.

Example 115 Top-Level Integration With Multiple Test Modes in the Hybrid Flow

```

read_verilog my_top_test_ready.v
read_test_model ddc/core1.ctlddc
read_test_model ddc/core2.ctlddc
current_design my_top
link

## Define the pins for compression/base_mode using "test_mode all".
## These modes are my_comp and my_scan1
for {set i 1} {$i <= 16 } { incr i 1 } {
    create_port -direction in test_si[$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 TestMode signals to be used
set_dft_signal -view spec -type TestMode \
    -port [list i_trdy_de i_trdy_ddi_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 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}

```

```

# Configure DFTMAX compression
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -base_mode my_base1 \
    -minimum_compression 10 \
    -test_mode scan_compression1 \
    -xtolerance high -hybrid true

# Configure the basic scan modes
# 8 chains for core1 xtol, 8 chains for core2, and 16 for top level
set_scan_configuration -chain_count 16 -test_mode my_base1
# 1 chain for burn_in mode
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
    -test_mode burn_in
set_dft_signal -view existing_dft -type TestClock -timing {45 55} \
    -port CLK
set_dft_insertion_configuration -synthesis_optimization none

# Create the test protocol
create_test_protocol

# Preview DFT insertion
preview_dft -show all

# Run the pre-DFT DRC
dft_drc

# Insert DFT logic
insert_dft
current_test_mode my_base1
dft_drc -verbose
change_names -rules verilog -hierarchy
remove_design core1
remove_design core2
write -format verilog -hierarchy -output vg/top_scan.v
write -format ddc -hierarchy -output ddc/top_scan.ddc
write_test_protocol -test_mode scan_compression1 \
    -output stil/ scan_compression1.stil -names verilog
write_test_protocol -test_mode my_base1 \
    -output stil/ my_base1.stil -names verilog
write_test_protocol -test_mode burn_in \
    -output stil/ burn_in.stil -names verilog

```

HASS and Hybrid Flow Limitations

Note the following limitations of the HASS and Hybrid flows:

- Post-DFT DRC of test modes that contain active compressed scan cores is not supported.
- Block-level patterns cannot be ported at the top level.

- When you use the `set_scan_configuration -chain_count` command at the top level with a chain count insufficient to satisfy all core scan pin connections, you see the following warning issued by the `preview_dft` command:

```
Warning: Cells with 33 new incompatible clock domains
have not been assigned to scan chains. Cannot honor
-chain_count specification of 2. (TEST-355)
```

- When you use the `-target` option of the `define_test_mode` command,
 - In the HASS core integration flow, you must enable HASS integration with the `-integration enable` option of the `set_dft_configuration` command, not the `-integration_only true` option of the `set_scan_compression_configuration` command.
 - In the Hybrid core integration flow, a top-level codec is inserted in a test mode only when you target the top-level logic by including the name of the current design in the target list. You cannot insert a codec for targeted cores without also compressing the top-level logic, which includes any untargeted standard scan cores and any wrapped cores in outward-facing mode.
- When you use DFT partitions,
 - In the Hybrid flow, only one partition can contain both cores and top-level logic. The remaining partitions can contain cores or top-level logic, but not both.
- When you integrate cores that contain external chains,
 - In the HASS flow, the external chains are not concatenated with other scan logic; instead, they use dedicated top-level scan I/O connections. You can use the `set_scan_path` command to manually concatenate them with other scan logic.
 - In the Hybrid flow, you must use the `set_scan_path` command to define how the external chains are incorporated into scan chains. Otherwise, incorrect codec logic can result.

19

Managing X Values in Scan Compression

A significant number of X sources in any compression architecture can degrade fault coverage, especially with high scan compression ratios. Today's complex designs often contain many such X sources: logic constrained by certain timing exceptions, memories and IP cores without test modes or models, combinational feedback loops, and nonscan flip-flops. This chapter describes features provided by DFTMAX compression to analyze and efficiently mask X values in the design.

This chapter includes the following topics:

- [High X-Tolerance Scan Compression](#)
 - [Static-X Analysis](#)
 - [Architecting X Chains](#)
-

High X-Tolerance Scan Compression

DFTMAX compression has a default tolerance for some Xs, but it also provides an option to implement a full tolerance of Xs. This topic describes *high X-tolerance* scan compression, a technology that provides low-impact, 100 percent X-tolerance for designs that use scan compression in the presence of many X sources.

This topic covers the following:

- [The High X-Tolerance Architecture](#)
 - [Enabling High X-Tolerance](#)
 - [Scan-In and Scan-Out Requirements](#)
 - [Limitations](#)
-

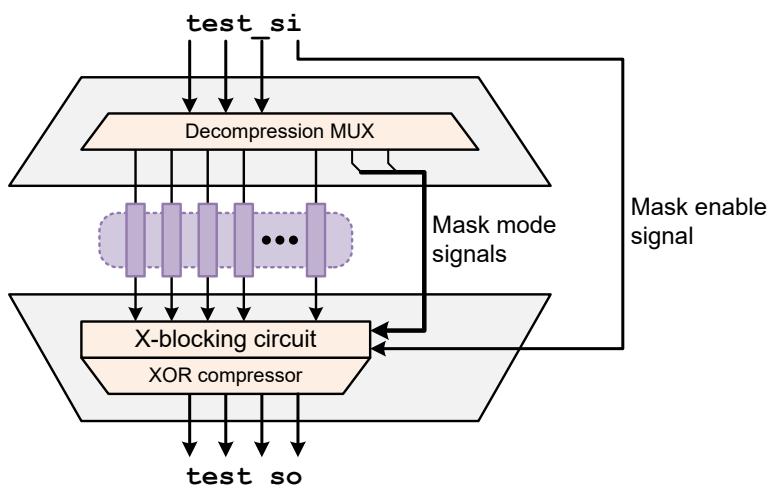
The High X-Tolerance Architecture

DFTMAX compression provides scan compression using only combinational circuitry. This approach achieves moderate to high compression while minimizing the additional cost for DFT implementation. Furthermore, scan compression can be applied to a wide variety of

designs, including designs with a large number of X values. The source of these Xs are either static (logic-induced) or dynamic (constraint-induced).

The high X-tolerance scan compression solution provided by DFTMAX compression meets the challenge of coverage loss by implementing new logic that selectively masks circuit response on a per-shift basis — a technique that provides 100 percent X-tolerance without introducing sequential circuitry. Note that this solution does not require any additional pins to perform X-masking. This architecture is shown in [Figure 311](#).

Figure 311 High X-Tolerance DFTMAX Compression Architecture



The high X-tolerance architecture provides the following observe modes:

- A full (unmasked) observe mode, which is equivalent to the default X-tolerance mode
- Additional X-tolerance (masked) observe modes, which can mask X values from selected chains before they reach the XOR compressor

An existing ScanDataIn port provides the mask enable signal. A mask enable value of zero selects the full observe mode. A mask enable value of one plus combinations of the mask mode signals selects additional X-tolerance observe modes for unload. Information about the X-tolerance observe modes is contained in the SPF in the CompressorStructures section in the Compressor `my_design_U_compressor_ScanCompression_mode` ModeControl definitions.

The high X-tolerance architecture introduces a combinational path between the scan input and scan output ports. The path travels from the scan input ports, through the decompression MUX mask signal generation logic, through the X-blocking and XOR compactor circuits, then through the scan output ports. This path can potentially contain long routes as well as combinational logic. To help meet timing requirements, you can use the pipelined scan data feature. For more information, see [Pipelined Scan Data on page 755](#).

Enabling High X-Tolerance

By default, the DFTMAX scan compression logic provides some tolerance of Xs. No option is needed to obtain this default X-tolerance capability.

For designs with large numbers of X values, you can enable the high X-tolerance feature with the following command:

```
dc_shell> set_scan_compression_configuration -xtolerance high
```

The `preview_dft` and `insert_dft` commands report information about the scan compression codecs in the design. The output from these commands includes an information message to confirm that the high X-tolerance feature is enabled:

```
Architecting Load Decompressor (version 5.8)
Number of inputs/chains/internal modes = 8/20/4
Architecting Unload compressor (version 5.8)
Number of outputs/chains = 5/20
Information: Compressor will have 100% x-tolerance
```

Scan-In and Scan-Out Requirements

The high X-tolerance feature imposes a limit on the number of compressed scan chains you can build with a given number of scan-in and scan-out pins. [Table 52](#) shows the maximum number of compressed scan chains that can be built for a given set of scan-in and scan-out pins.

Table 52 High X-Tolerance Compressed Scan Chain Limits

Number of scan-in and scan-out pins	Maximum number of chains without OCC controller	Maximum number of chains with OCC controller ¹¹
2	4	
3	12	6
4	32	16
5	80	40
6	192	96
7	448	224
8	1024	512
9	2304	1152

11. This column assumes that a single decompressor input is dedicated to OCC clock chains. Additional dedicated clock chain decompressor inputs will further reduce the limit.

Number of scan-in and scan-out pins	Maximum number of chains without OCC controller	Maximum number of chains with OCC controller ¹¹
10	5120	2560
11	11264	5632
12	24576	12288
13	32000	26624
14 and higher	32000	32000

If the on-chip clocking (OCC) feature is used with compressed clock chains, the dedicated decompressor scan input lowers the limit. For more information about compressed clock chains, see [Scan Compression and OCC Controllers on page 687](#).

Table 53 shows the maximum compressed scan chain count when high X-tolerance is used with some asymmetrical low-pin-count configurations:

Table 53

Asymmetrical scan-in, scan-out pin configuration	Maximum number of chains without OCC controller	Maximum number of chains with OCC controller
2 scan-ins, 1 scan-out	2	
3 scan-ins, 1 scan-out	4	
3 scan-ins, 2 scan-outs	8	4
4 scan-ins, 3 scan-outs	24	12
5 scan-ins, 4 scan-outs	64	32
6 scan-ins, 5 scan-outs	160	80

If the specified compressed scan chain count cannot be satisfied, the tool issues an error message that contains information about how many compressed scan chains were requested and how many compressed scan chains can be built for the current scan pin configuration. [Example 116](#) shows the error message issued when 20 compressed scan chains are requested, but only 12 scan chains can be built.

11. This column assumes that a single decompressor input is dedicated to OCC clock chains. Additional dedicated clock chain decompressor inputs will further reduce the limit.

Example 116 High X-Tolerance Error Message for Insufficient Scan-In Pins

```
Error: Architecting of Load/Unload compressor failed with the given set
of parameters. (TEST-1722)
      Number of internal chains architected: 20
      Number of available compression channels: 12
      Number of load compressor inputs: 3
      Number of unload compressor outputs: 3
```

You can also use the TestMAX ATPG `analyze_compressors` command to determine if a codec can be built for a given set of parameters. For more information, see TestMAX ATPG and TestMAX Diagnosis Online Help.

Limitations

Note the following limitations of the high X-tolerance feature:

- All codecs must have the same X-tolerance type when the following features are used:
 - Shared codec I/O connections
 - Serialized compressed scan
- You cannot use end-of-cycle measures with high X-tolerance codecs.

Static-X Analysis

Some flip-flops capture X values more often than others because they are located in the fanout of logic constructs that generate unknown values. A flip-flop that frequently captures an X value during capture is called a *static-X cell*.

DFTMAX compression provides a static-X analysis feature that analyzes a design and reports static-X cells. This analysis feature can be used in both standard scan and compressed scan flows. When enabled, it reports static-X cell information during pre-DFT DRC. By itself, static-X analysis does not affect subsequent scan chain architecture.

Static-X analysis is enabled by using the following command:

```
dc_shell> set_dft_drc_configuration -static_x_analysis enable
```

When enabled, static-X analysis directs pre-DFT DRC to carry out an X-probability analysis of the sequential cells by invoking combinational simulation within the tool and then defining those sequential cells with high X-capture probability as static-X cells. The analysis is performed using the following procedure:

1. Normal test DRC analysis is performed to determine the set of scannable cells.
2. Constrained primary inputs are set to their constrained values, clocks are set to their inactive values, and scan-enable signals are set to their inactive (capture) values.

3. Constant-value state elements are set to their constant values.
4. Other primary inputs and nonconstant scannable cells are set to random binary values. All other state elements, such as nonscan cells, are set to X.
5. 1024 random patterns are simulated to determine the frequency that the data input of a scannable cell is at X.
6. A scannable cell whose data input is X with a frequency exceeding a predetermined threshold is recorded along with its frequency of capturing an X value. The predetermined threshold is 25 percent and cannot be changed.

After identifying the static-X cells, the `dft_drc` command reports them as D39 violations in the following format:

```
Warning: Probability of capture X (X probability <%>) exceeds threshold
for scan-cell DFF %s. (D39-x)
```

The `dft_drc` command also sets the `test_dft_xcellViolation` attribute on all identified static-X cells. You can use this attribute to obtain the cells for further script-based analysis:

```
dc_shell> set static_x_cells \
           [get_cells -hierarchical * -filter {test_dft_xcellViolation == true}]
```

When debugging static-X cells, remember that a static-X cell captures the frequent X values, but the source of these frequent X values will likely exist in the fanin logic to the cell. The following design constructs can introduce X values into the design logic:

- Black boxes
- CTL models
- Combinational feedback loops
- Uncontrolled internal buses
- Uncontrolled bidirectional ports
- Nonscan cells

Note:

Timing exceptions are not considered as a source of X values during DFTMAX static-X analysis. Instead, they are considered as dynamic path-specific sources of X values in the TestMAX ATPG tool.

Architecting X Chains

DFTMAX compression provides a static-X chain feature that identifies scan cells that frequently capture X values, and then groups them exclusively into special scan chains, called *X chains*. By grouping X-capturing cells into dedicated scan chains, incidental X masking of any chain is reduced or eliminated.

The static-X chain feature deals with the pattern inflation that is caused by the occurrence of static-X cells in compression mode. This method of handling the X cells achieves better test data volume reduction (TDVR), and improves ATPG quality of results. X-chain information is communicated seamlessly in the DFTMAX to TestMAX ATPG flow through the STIL protocol file (SPF). The SCANDEF file generated by the TestMAX DFT tool groups the X cells into separate SCANDEF partitions so that a place-and-route tool can preserve those groups.

This topic covers the following:

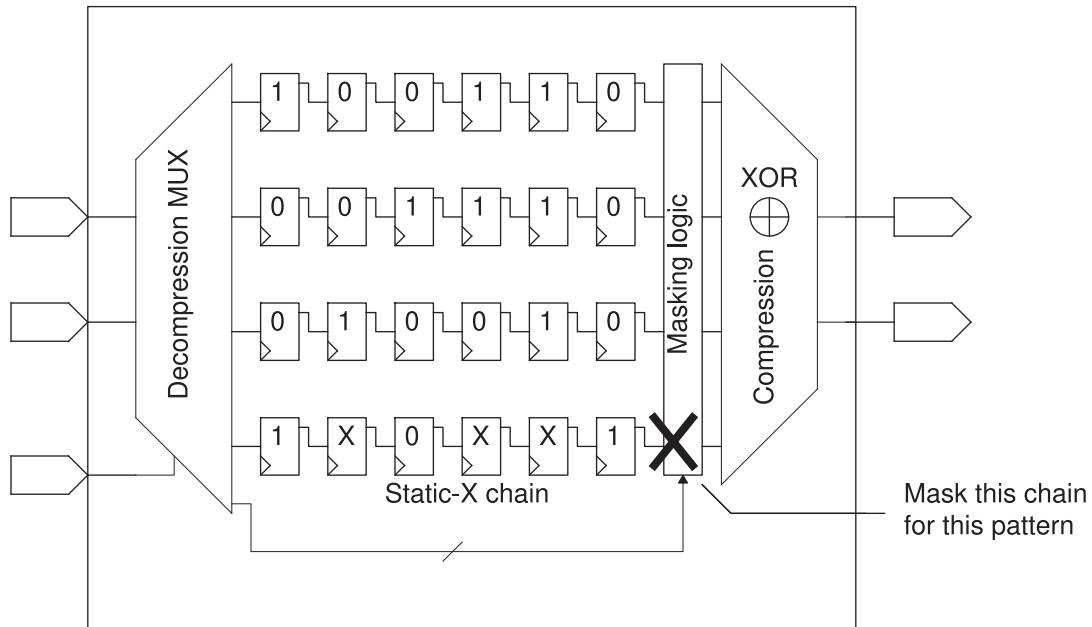
- [The X-Chain Architecture](#)
- [Enabling X Chains](#)
- [Manually Specifying X-Chain Cells](#)
- [Using the set_scan_path Command With X Chains](#)
- [Using AutoFix With X Chains](#)
- [Using X Chains in Hierarchical Flows](#)
- [Using the test_simulation_library Variable](#)
- [Representing X Chains in SCANDEF Files](#)
- [Passing X-Chain Information to TestMAX ATPG](#)
- [Error and Warning Summaries](#)
- [X-Chain Usage Guidance](#)

The X-Chain Architecture

The static-X analysis feature identifies and reports scan cells that frequently capture X values, called *static-X cells*. However, by itself, the static-X analysis feature does not affect scan chain architecture; it is only an analysis feature.

The X-chain feature builds on the static-X analysis feature. It groups these identified static-X cells exclusively into special scan chains, called *X chains*. This allows pattern generation to efficiently mask static-X cells for most test patterns, leaving the other chains unmasked, as shown in [Figure 312](#).

Figure 312 X Values in Static-X Chains



As a result, fewer chains are masked by the high X-tolerance masking logic, resulting in better fault coverage and a lower volume of test data.

Enabling X Chains

To isolate the identified static-X cells and architect X chains that contain only these cells, use the following commands:

```
dc_shell> set_dft_drc_configuration -static_x_analysis enable
dc_shell> set_scan_compression_configuration \
           -xtolerance high \
           -static_x_chain_isolation true
```

The X-chains feature has the following requirements:

- Static-X analysis must be enabled, as the static-X cell attributes are used to determine which cells are placed in the X chains. For more information, see [Static-X Analysis on page 739](#).
- The X-chain feature only operates in compressed scan modes, and it requires that high X-tolerance is enabled. For more information, see [High X-Tolerance Scan Compression on page 735](#).

The `preview_dft` command reports X chains with an additional “(X chain)” label, as shown in the following example:

```
*****
Current mode: ScanCompression_mode
*****
Number of chains: 320
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks

Scan chain '1' contains 65 cells
    Active in modes: ScanCompression_mode

Scan chain '2' contains 50 cells (X chain)
    Active in modes: ScanCompression_mode

Scan chain '3' contains 65 cells
    Active in modes: ScanCompression_mode
```

Note:

Use the `preview_dft` command to identify X chains. The `report_scan_path` command does not identify X chains.

The tool performs scan chain balancing with the additional constraint that static-X cells cannot be mixed with non-static-X cells in the same scan chain. If the number of static-X cells exceeds 20 percent of the total number of scan cells, the tool issues the following error message:

```
Error: Too many static-X cells in design. Cannot isolate cells as
separate X-chains. (TEST-1090)
```

Note that assigning the static-X cells to X chains applies an additional constraint to the physical implementation tool. The static-X cells could be distributed across the chip, so connecting them together into one or more scan chains can result in long wires that contribute to routing congestion.

Manually Specifying X-Chain Cells

The X-chain feature uses the `test_dft_xcellViolation` cell attribute, set by static-X analysis, to determine the scan cells placed in dedicated X chains. After pre-DFT DRC completes, you can manually set or remove this attribute on scan cells to modify the set of static-X cells. The `insert_dft` command then uses the modified set of static-X cells to construct the X chains.

To specify cells as static-X cells, use the following command:

```
dc_shell> set_attribute -type boolean [get_cells cell_list] \
    test_dft_xcellViolation true
```

To remove the static-X attribute from cells, use the following command:

```
dc_shell> remove_attribute [get_cells cell_list] \
    test_dft_xcellViolation
```

Note:

The `set_attribute` and `remove_attribute` commands must be used after pre-DFT DRC is run with the `dft_drc` command, but before scan is inserted with the `insert_dft` command.

Use this capability to include additional scan cells in the X chains. For example, certain scan cells might frequently capture dynamic X values during pattern generation in TestMAX ATPG. If you know the timing exceptions and are able to translate them to capturing scan cell names, you can mark these scan cells as static-X cells with the `set_attribute` command.

The `test_dft_xcellViolation` attribute is only honored for leaf scan cells. It is ignored for hierarchical cells (including CTL-modeled cells).

Using the `set_scan_path` Command With X Chains

If both static-X cells (identified during pre-DFT DRC) and non-static-X cells are defined in a common `set_scan_path` command applied to a scan compression mode, the resulting scan path is not an X chain even though it contains static-X cells. When this happens, the tool issues the following warning message:

Warning: Chain %s has both X and non-X cells. (TEST-1079)

However, if the defined scan path consists entirely of static-X cells, the scan path becomes an X-chain.

This behavior applies only to `set_scan_path` commands applied to a scan compression mode. If a `set_scan_path` specification is applied to a standard scan mode, it does not affect X-chain scan architecture.

Using AutoFix With X Chains

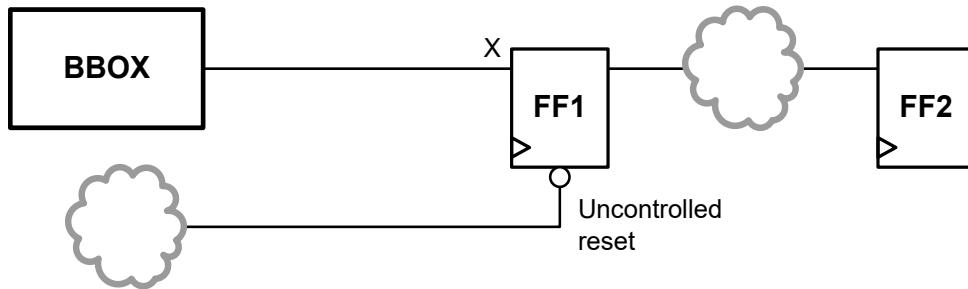
If static-X analysis reports a large number of static-X cells, you can use AutoFix to fix X-capture problems.

When using AutoFix with the X-chains feature, pre-DFT DRC performs static-X analysis before applying AutoFix. If the features are used together, they can potentially interact in a way that affects the static-X analysis and the resulting X chains.

Consider the circuit shown in [Figure 313](#). Flip-flop FF1 always captures an X value. During static-X analysis, FF1 is treated as a nonscan cell due to its uncontrollable reset and is therefore not marked as a scannable static-X cell. During AutoFix processing, the

uncontrollable reset becomes controllable, and FF1 becomes scannable. It is placed into a regular scan chain despite always capturing an X value.

Figure 313 Using AutoFix With X Chains



Sequential cells with D1, D2, and D3 violations (uncontrollable clock, set, and reset signals) are susceptible to this interaction if they frequently capture X values and are subsequently made scannable by AutoFix.

To avoid this interaction, use the following two-pass flow:

1. Apply AutoFix with scan insertion disabled, so that only the AutoFix logic is inserted:

```

set_dft_configuration \
    -scan_disable \
    -fix_clock_enable -fix_set_enable -fix_reset_enable
set_automfix_configuration ...
preview_dft
insert_dft
  
```

2. Disable AutoFix, reenable scan insertion, and continue with normal scan insertion:

```

set_dft_configuration \
    -fix_clock_disable -fix_set_disable -fix_reset_disable \
    -scan_enable -scan_compression_enable
set_scan_configuration ...
set_scan_compression_configuration ...
  
```

An alternative solution is to use the single-pass flow and manually mark the affected X-capturing sequential cells with the `test_dft_xcellViolation` attribute. However, this solution requires knowledge of the cells that are affected by the interaction.

See Also

- [Using AutoFix on page 333](#) for more information about using AutoFix to fix design testability issues

Using X Chains in Hierarchical Flows

This topic provides information on using X chains in hierarchical flows. The behaviors described in this section result from using core test language (CTL) models to represent core blocks during hierarchical flows.

Note:

When a DFT run includes a previously scan-inserted core read from a full-netlist .ddc file, the tool still uses a CTL model representation during DFT operations.

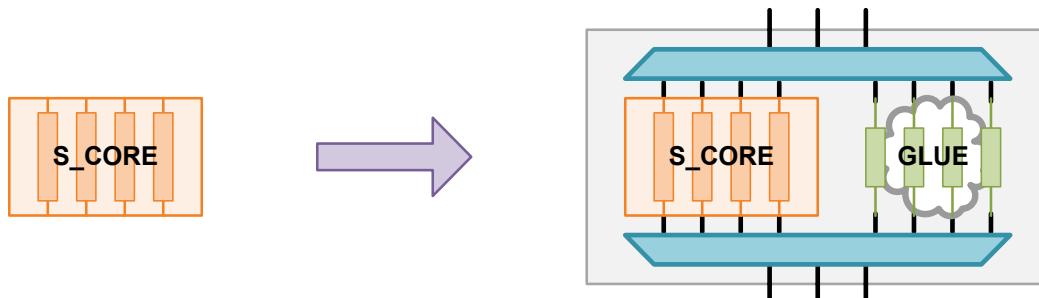
This topic covers the following:

- [Static-X Cells in the HASS Flow](#)
- [Hierarchical Blocks and X Sources](#)

Static-X Cells in the HASS Flow

This topic pertains to the HASS flow, in which standard scan is inserted at the core level, and then compressed scan is inserted at the top level using these core-level standard scan chain segments. [Figure 314](#) shows an example of this flow.

Figure 314 Applying the HASS Flow to a Standard Scan Core



Static-X analysis can be performed in both standard scan and compressed scan flows. However, X chains can only be created in compressed scan flows. When creating a standard scan core that will subsequently be scan-compressed in a HASS flow, X analysis can be used to determine if there are any static-X cells in the core. However, X chains cannot be used to consolidate the static-X cells into X chains at the core level.

When using this flow, you should use static-X analysis to ensure that the standard scan core does not contain any static-X cells. If it does, consider using AutoFix to resolve the X sources. For more information, see [Using AutoFix on page 333](#). Note that AutoFix might not be able to resolve all X sources.

At the top level, the core is modeled using CTL model information during hierarchical compressed scan insertion. Any static-X cells that exist in the core-level scan chain

segments are not visible to top-level DRC analysis, and they are incorporated into regular codec scan chains.

The `dft_drc -verbose` command reports the core-level standard scan chain segments as possible D39 violations, with one violation reported for each segment:

```
-----
Begin Pre-DFT violations...

Warning: Probability of capture X (100) exceeds threshold for scancell
CORE Usub. (D39-1)
Warning: Probability of capture X (100) exceeds threshold for scancell
CORE Usub. (D39-2)
Warning: Probability of capture X (100) exceeds threshold for scancell
CORE Usub. (D39-3)
Warning: Probability of capture X (100) exceeds threshold for scancell
CORE Usub. (D39-4)

Pre-DFT violations completed...
-----

-----
DRC Report

Total violations: 4
-----

4 PRE-DFT VIOLATIONS
    4 Static X scan cell violations (D39)

Warning: Violations occurred during test design rule checking. (TEST-124)

-----
Sequential Cell Report
    0 out of 1333 sequential cells have violations
-----

SEQUENTIAL CELLS WITHOUT VIOLATIONS
    * 1329 cells are valid scan cells
        Z_reg[30]
        Z_reg[29]
        Z_reg[28]
        ...
        ...

CORE SEGMENTS WITHOUT VIOLATIONS
    * 4 core segments are valid scan segments
        Usub/1
        Usub/2
        Usub/3
        Usub/4
```

The CTL model scan chain segments are reported as D39 violations to report that the segments could potentially contain static-X cells. Although the scan chain segments are reported as possible static-X sources, they are not incorporated into the top-level X chains.

Pre-DFT DRC also issues the following warning message:

Warning: Static X-cell analysis may be inaccurate on design containing cells with CTL models. (TEST-610)

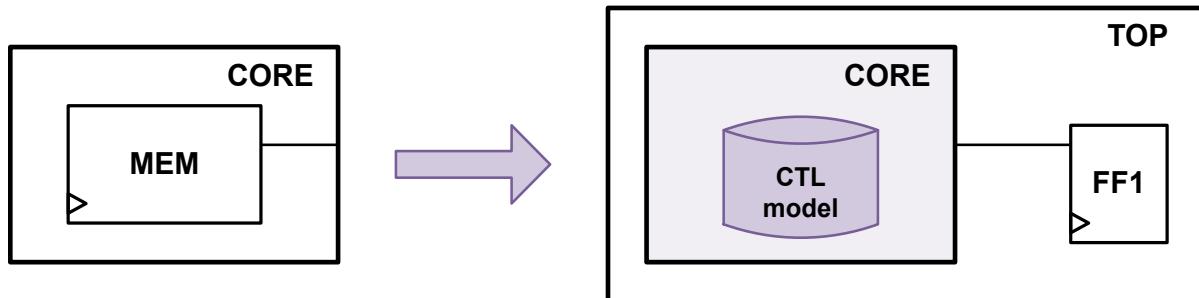
For compressed scan cores used in a hierarchical adaptive scan synthesis (HASS) flow, there is no problem with static-X cells, if they are present, because the static-X cells were already isolated into X chains when the models were created.

Hierarchical Blocks and X Sources

When a compressed scan insertion run includes a previously scan-inserted core, the core is represented using CTL model information. This is true even when the core is represented as a full netlist read from a .ddc file. This CTL model contains information about the DFT logic in the core, but does not include functional information about the core outputs.

In some cases, it might be possible for X sources to drive core outputs. Consider the example shown in [Figure 315](#), where a memory cell drives a core output.

Figure 315 Modeling Cores With X Sources Using CTL Models



During core-level DRC, no scannable cells capture an X value from the memory, and therefore no static-X cells are reported by static-X analysis. During top-level DRC, a scannable cell now captures the memory output. However, the core is modeled using CTL model information, which does not model the functional core outputs. As a result, static-X analysis does not detect that flip-flop FF1 frequently captures an X value, and it is placed into a regular codec scan chain.

If you know the list of affected top-level flip-flops, apply the `test_dft_xcellViolation` attribute to place them into X chains. For more information, see [Manually Specifying X-Chain Cells on page 743](#).

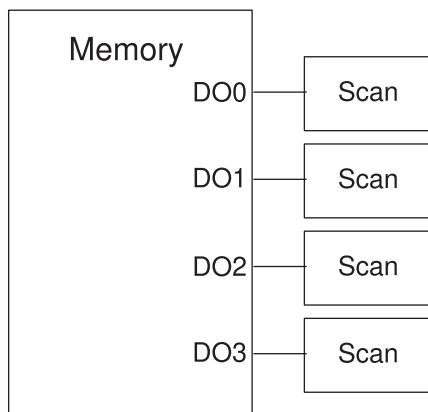
Using the `test_simulation_library` Variable

In some cases, the synthesis libraries used by the tool might model a cell as a black box, while the simulation libraries used by the TestMAX ATPG tool might provide functional information for the same cell. The `test_simulation_library` variable enables DFTMAX DRC to use the TestMAX ATPG simulation libraries.

When using the X-chains flow, it is desirable to replace most types of black-box cells with their simulation models. This provides a more accurate assessment of X behaviors during static-X analysis. You can verify the resulting X chains in the TestMAX ATPG tool by looking for the M469, M470, and M471 messages in conjunction with the `analyze_compressors -xchain_analysis` command. These messages report on the consistency of X-capture frequency between regular scan chains and X chains.

However, you should not configure memory cell models with the `test_simulation_library` variable when using the X-chains flow. [Figure 316](#) shows an example where flip-flops capture the values from memory outputs.

Figure 316 Memory Outputs Driving Scan Cells



If the simulation model is configured for the memory, pre-DFT DRC reports the information shown in [Example 117](#).

Example 117 Pre-DFT DRC Report for Memory Models

```
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...
...Loading simulation libraries...
```

```

...checking vector rules...
...checking pre-dft rules...

-----
Begin Modeling violations...
Warning: Cell U_RAM (MEM) is unknown (black box) because functionality
for output pin Q[0] is bad or incomplete. (TEST-451)
Information: Cells with this violation : U_RAM. (TEST-283)

Modeling violations completed...
-----

-----
Begin Pre-DFT violations...
Warning: Clock input clk of DFF U_RAM cannot capture data. (D17-1)
Warning: Probability of capture X (100) exceeds threshold for scancell
        DFF U_RAM. (D39-1)

Pre-DFT violations completed...
-----
```

The D39 violation is issued for the memory model itself. However, no D39 violations are issued for the sequential cells that capture the memory outputs. The simulation model used by DFTMAX DRC is treated like a Verilog netlist with instantiated sequential cells. These sequential cells are considered as valid scannable cells within the `dft_drc` command, which results in hiding the X-generation effect of the memory outputs. As a result, the sequential cells connected directly or indirectly to the memory outputs are not treated as static-X cells and are therefore not placed into X chains.

Subsequently, in the TestMAX ATPG tool, memories are X-generators during much of the ATPG process, and those sequential cells capture Xs, although the actual capture details might depend on the particulars of the ATPG engine and target faults. Therefore, it is better to put those cells into X chains. To do this, the memory models should not be configured with the `test_simulation_library` variable. Instead, the normal black-box synthesis memory models should be used during DFTMAX pre-DFT DRC.

Representing X Chains in SCANDEF Files

The TestMAX DFT tool puts DFTMAX X-chain information in the SCANDEF file to instruct the physical implementation tool to preserve the X chains during optimization. Static-X cells are written into a SCANDEF partition whose label name starts with `X_`. This allows the place-and-route tool to preserve those groups, as shown in the following SCANDEF fragment:

```

- 1424
+ START M12/U4497 Y
+ FLOATING Q12/PCB_8192x64c16s0_bit_reg_17_ ( IN SI ) ( OUT Q )
Q12/PCB_8192x64c16s0_bit_reg_49_ ( IN SI ) ( OUT Q )
```

```

Q12/PCB_8192x64c16s0_bit_reg_50_ ( IN SI ) ( OUT Q )
Q12/PCB_8192x64c16s0_bit_reg_25_ ( IN SI ) ( OUT Q )
Q12/PCB_8192x64c16s0_bit_reg_28_ ( IN SI ) ( OUT Q )
Q12/PCB_8192x64c16s0_bit_reg_56_ ( IN SI ) ( OUT Q )
+ PARTITION X_clk_core_45_45
+ STOP Q12/PCB_8192x64c16s0_bit_reg_21_ SI ;

```

Passing X-Chain Information to TestMAX ATPG

To generate the test patterns and masking controls, the ATPG tool needs to have access to information about the structure of the X chains. The TestMAX DFT tool puts this information into the SPF for use by the ATPG tool.

The following is an SPF example with X chains:

```

Compressor "top_U_compressor_ScanCompression_mode" {
    ModeGroup mode_group;
    UnloadGroup unload_group;
    UnloadModeGroup unload_mode_group0 unload_mode_group1
        unload_mode_group2;
    CoreGroup core_group;
    UnloadModeEnable enable_group;
    Modes 193;
    Mode 0 {
        ModeControls {
            "test_si17" = 0;
        }
        Connection "3" 0 1 2 3;
        Connection "4" 4;
        Connection "5" 5;
        Connection "6" 6;
    }
}
```

Compressed chains that are part of the `CoreGroup` but are not connected in Mode 0 are implicitly defined as X chains. In the following example, compressed chains “1” and “2” are X chains that consist of static-X cells exclusively. Mode 0 is called the full observe XOR mode. The two X chains are observed by a direct-observability mode, in which each X-chain can be observed at a different, single output with no other compressed chains XORed.

```

Mode 58 {
    ModeControls {
        "test_si17" = 1;
        "test_si15" = 0;
        "test_si16" = 0;
        "test_si1" = 1;
        "test_si2" = 1;
        "test_si3" = 1;
        "test_si4" = 0;
        "test_si5" = 0;
        "test_si6" = 1;
    }
}
```

```
Connection "1" 3;
Connection "2" 11;
Connection "28" 6;
Connection "53" 14;
Connection "78" 15;
Connection "103" 10;
Connection "127" 0;
Connection "177" 2;
Connection "226" 16;
Connection "1012" 12;
Connection "1013" 9;
Connection "1014" 1;
Connection "1015" 8;
Connection "1016" 5;
Connection "1018" 13;
Connection "1019" 4;
Connection "1020" 7;
```

Error and Warning Summaries

The following error and warning messages exist for this feature:

- TEST-1090 (Error) Too many X cells in design. Cannot isolate static X cells as separate X chains.

Description:

You receive this message if you have specified X-chain isolation in a Scan Compression mode and more than 20 percent of valid scan cells have static-X (D39) violations.

- TEST-1088 (Warning) Static X chain isolation is ignored in %s as high xtolerance is not enabled.

Description:

You receive this message if you have specified X-chain isolation without high X-tolerance in a Scan Compression mode, using the `set_scan_compression_configuration` command. Static-X chain isolation will be ignored for this mode.

- TEST-610 (Warning) Static X cell analysis may be inaccurate on design containing cells with CTL models.

Description:

This message indicates that the current design has cells with CTL models, which means the static-X cell analysis might be inaccurate.

- TEST-1079 (Warning) Chain %s has both X and non-X cells.

Description:

You receive this message if you have specified a `set_scan_path` command that mixes X and non-X cells within the same scan chain in a scan compression mode. The resulting mixed chain will not be considered as an X-chain.

X-Chain Usage Guidance

The X-chains feature is intended for designs that have static-X values. This feature might not improve results for designs with dynamic X values when compared to high X-tolerance without X chains. There is no automated way to determine which type of X value is propagated within a design. Additional test data volume reduction can be achieved with the X-chain feature if the following conditions are met:

- A considerable number of memories or hard macros are used in TestMAX ATPG.
- A large number of R14 violations are issued by DRC in the TestMAX ATPG tool.
- Many scan cells are analyzed as X-scan cells due to capturing X values from tieX cells, as noted when executing the TestMAX ATPG `set_simulation -analyze_x_sources` and `run_simulation` commands.

20

Advanced DFTMAX Compression

This chapter describes advanced features that can be used while inserting compressed scan circuitry into your design. These features are used to customize DFT insertion, to improve the frequency of the scan testing logic, and to reduce the pattern count for pin-limited designs.

This chapter includes the following topics:

- [Specifying a Location for Codec Logic Insertion](#)
- [Pipelined Scan Data](#)
- [Sharing Codec Scan I/O Pins](#)
- [Implicit Scan Chains](#)

Specifying a Location for Codec Logic Insertion

By default, the tool inserts the scan compression codec at the top level of the current design. However, you can use the `set_dft_location` command to specify an alternate insertion location:

```
dc_shell> set_dft_location -include {CODEC} instance_name
```

The specified instance name must be a hierarchical cell. It cannot be a library cell, black box, or black-box CTL model.

If the specified hierarchical cell does not exist, the `insert_dft` command creates it during DFT insertion. For more information, see [Creating New DFT Logic Blocks on page 284](#).

Note:

Compressed scan reconfiguration MUXs and test-mode decode logic are not placed in the specified location.

When a top-down multiple partition flow is used, this feature can be used to place each partition's codec logic at a specified location. For example,

Example 118 Specifying Codec Logic Insertion Locations for Multiple Partitions

```
set_dft_location core ;# place all non-codec logic in core

define_dft_partition P1 -include BLK1
define_dft_partition P2 -include BLK2

current_dft_partition P1
set_scan_configuration -chain_count 4
set_scan_compression_configuration -chain_count 10
set_dft_location -include {CODEC} core/BLK1

current_dft_partition P2
set_scan_configuration -chain_count 3
set_scan_compression_configuration -chain_count 8
set_dft_location -include {CODEC} core/BLK2
```

For compatibility, the tool also supports an older, deprecated method for specifying the codec insertion location:

```
dc_shell> set_scan_compression_configuration -location instance_name
```

The specified instance name must already exist. This method takes precedence over the `set_dft_location` command.

See Also

- [Specifying a Location for DFT Logic Insertion on page 280](#) for more information about specifying the insertion location for other types of DFT logic
- [Per-Partition Scan Compression Configuration Commands on page 672](#) for more information about DFT specifications that can be specified per-partition

Pipelined Scan Data

Pipelined scan data is a feature provided by the TestMAX DFT tool to resolve delay problems associated with long routes in compressed scan chain logic.

This topic covers the following:

- [Introduction to Pipelined Scan Data](#)
- [Using Pipelined Scan Data](#)
- [Using Pipelined Scan Data With Scan Compression](#)
- [Pipelined Scan Data Specifications](#)
- [Pipelined Scan Data Test Protocol Format](#)

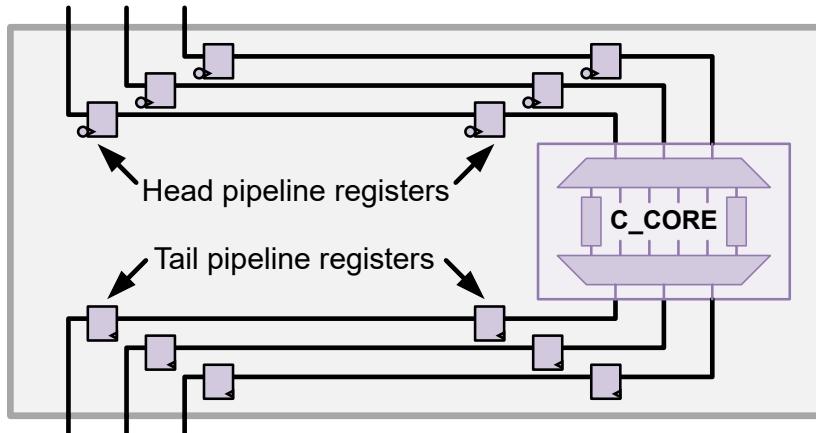
- [Pipelined Scan Data Limitations](#)
- [Hierarchical Flows With Pipelined Scan Data](#)

Introduction to Pipelined Scan Data

In typical scan flows, long wires between the scan chain input and the first flip-flop and between the last flip-flop and the scan chain output can cause delay problems. Scan compression logic and higher scan frequencies further amplify the problem. These delays are reduced by placing pipeline registers at the beginning and end of the scan chains. They divide the long routes between the scan chain terminals into smaller wires between the registers and therefore help reduce the path delay.

[Figure 317](#) shows an example of a compressed scan design with pipeline registers. The head pipeline registers are placed between the scan inputs and the decompressor, and the tail pipeline registers are placed between the compressor and the scan outputs.

Figure 317 Pipeline Registers in a Compressed Scan Design



The tool can automate the insertion of the head and tail pipeline registers around the codec, or you can provide user-defined pipeline registers at the scan inputs and outputs that the tool connects to the codec. Test DRC verifies the correct operation of the pipeline registers and updates the test protocol that TestMAX ATPG uses for pattern generation.

Using Pipelined Scan Data

The pipelined scan data feature provides two methods of specifying pipeline register insertion:

- Pipeline registers can be automatically inserted and connected by the tool during the `insert_dft` command.
- User-defined pipeline registers can be provided in the design logic. The tool makes the needed scan path connections to these existing pipeline registers during the `insert_dft` command.

These pipeline register insertion methods are explained in the following topics:

- [Enabling Pipelined Scan Data](#)
- [Automatically Inserting Head and Tail Pipeline Registers](#)
- [Specifying User-Defined Head and Tail Pipeline Registers](#)

Enabling Pipelined Scan Data

For both the automatically inserted and user-defined pipeline register flows, use the `set_dft_configuration` command to enable the pipelined scan data feature in the compressed scan flow:

```
set_dft_configuration -pipeline_scan_data enable
```

By default, pipeline registers are not inserted.

The type of insertion method is determined by the commands you use to configure the pipeline registers. You must choose either automatic or user-defined pipeline register insertion, as these methods are mutually exclusive.

Automatically Inserting Head and Tail Pipeline Registers

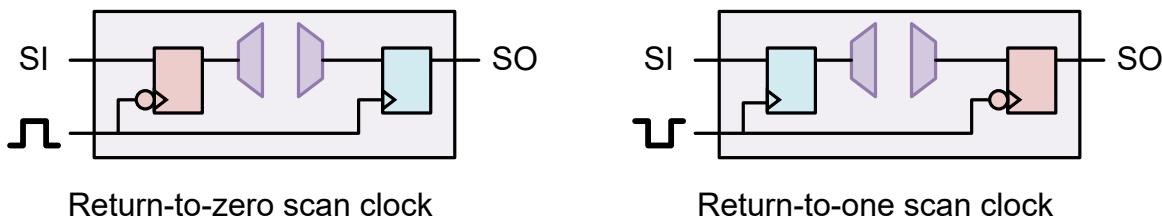
The `set_pipeline_scan_data_configuration` command is used to configure the automatic pipeline register insertion. Specify the number of head and tail pipeline stages using the `-head_pipeline_stages` and `-tail_pipeline_stages` options:

```
set_pipeline_scan_data_configuration \
    -head_pipeline_stages integer \
    -tail_pipeline_stages integer \
    -head_scan_flop true
```

The `-head_scan_flop` option causes the tool to create scan-replaced head pipeline registers that hold their state during scan capture. For more information, see [Avoiding X Capture in Head Pipeline Registers on page 764](#).

DFT insertion uses a D flip-flop from the target library for the automatically inserted pipeline registers. To minimize the need for lock-up latches at the compressed scan chains, the head pipeline registers are clocked on the trailing clock edge, and the tail pipeline registers are clocked on the leading clock edge. Rising-edge-triggered or falling-edge-triggered flip-flops are used depending on whether the scan clock uses a return-to-zero or return-to-one waveform. [Figure 318](#) shows the flip-flops used for both types of clock waveforms.

Figure 318 Pipeline Registers in a Compressed Scan Design



Note:

If you enable retiming registers on the scan input side by using the `set_scan_configuration -add_test_retimings_flops` command with the `begin_only` or `begin_and_end` option values, the head pipeline registers are clocked on the leading clock edge instead.

The newly inserted pipeline registers have names of the form

```
SNPS_PipeHead_SI_pin_name_stage
SNPS_PipeTail_SO_pin_name_stage
```

where `SI_pin_name` is the scan in port, `SO_pin_name` is the scan out port, and `stage` is the stage depth.

By default, a new test clock port named `SNPS_PipeClk` is created for the pipeline registers. If you want to use an existing design clock instead, use the `-head_pipeline_clock` and `-tail_pipeline_clock` options to specify the clock:

```
set_pipeline_scan_data_configuration \
    -head_pipeline_clock clock_name \
    -tail_pipeline_clock clock_name \
    -head_pipeline_stages integer \
    -tail_pipeline_stages integer \
    -head_scan_flop true
```

Use the `report_pipeline_scan_data_configuration` command to report the current automatic insertion configuration, and the `reset_pipeline_scan_data_configuration` command to reset the automatic insertion configuration.

The automatically inserted pipeline register flow is only available for compressed scan designs created with the tool.

Specifying User-Defined Head and Tail Pipeline Registers

DFT Compiler can connect user-provided pipeline registers to the compressed scan logic. You can use this feature to pre-place pipeline registers at strategic locations in tight floorplans. DFT Compiler automatically makes the needed scan path connections during the `insert_dft` command.

You can use the `set_scan_path` command to specify the scan data path connections to these pipeline registers:

```
set_scan_path chain_name \
    -scan_data_in port_name \
    -scan_data_out port_name \
    -pipeline_head_registers instance_list \
    -pipeline_tail_registers instance_list \
    -view spec
```

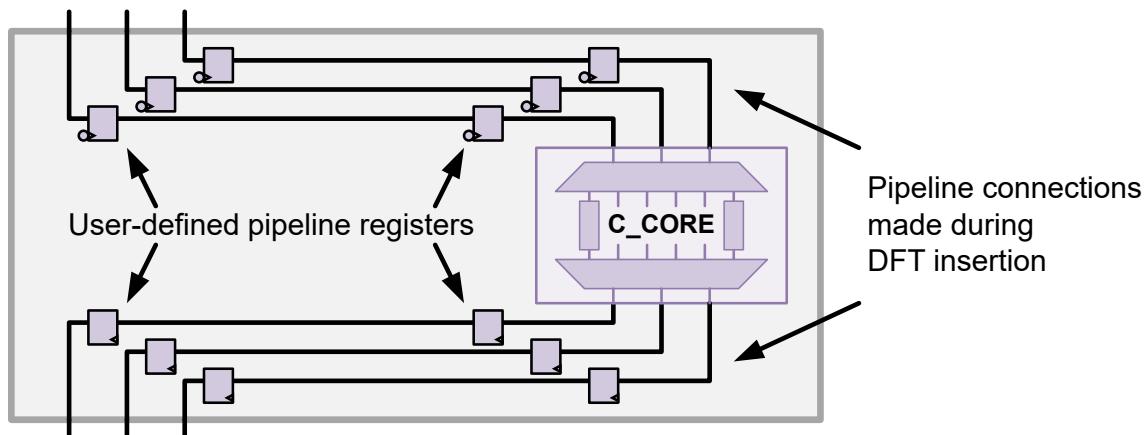
Note:

Do not specify a test mode for the `set_scan_path` command when defining pipeline connections. The tool automatically propagates the specification to all test modes during scan architecture.

Each `set_scan_path` command associates a set of existing head and/or tail pipeline registers with a scan-in and scan-out port. A full scan path chain definition with scan elements is not needed. You can provide a list of multiple pipeline registers to implement multiple pipeline stages. The head and tail pipeline depths can be different.

For user-defined pipeline registers, you are responsible for the connections to the scan data ports and between the pipeline stages, and for proper conditioning for the scan ports and clocks to the pipeline registers. [Figure 319](#) demonstrates the required connections for two head pipeline registers and two tail pipeline registers per scan chain.

Figure 319 User-Defined Pipeline Registers and Connections



During DFT insertion, DFT Compiler makes connections from the pipeline registers to the scan decompression MUX and XOR compressor logic, but does not check the validity of the paths between the scan inputs and head registers and between the tail registers and the scan outputs. After DFT insertion, the `dft_drc` command determines whether the scan chains are shifting properly.

You can apply the `set_size_only` command to user-defined pipeline registers before the first `compile -scan` command so that the registers are not removed by logic optimization. Although the `size_only` property allows the `compile -scan` command to scan-replace the pipeline registers with their scan equivalents, the `insert_dft` command will unscan the pipeline registers before making the pipeline connections.

When user-defined pipeline registers are specified with the `set_scan_path` command, DFT Compiler ignores any options relating to automatically inserted pipeline registers specified with the `set_pipeline_scan_data_configuration` command.

Observe these additional requirements when implementing user-defined pipeline registers:

- Design the pipeline structures before DFT insertion so that the registers can be referenced in the `set_scan_path` command.
- Ensure that the specified chain count and the number of scan chains are the same.
- Specify the head and tail pipeline registers with a full hierarchical name.
- Specify the corresponding scan input and output ports for each external chain.
- Specify the pipeline registers in scan order, from input pin to scan chain, and from scan chain to output pin.
- All head pipeline registers must be triggered by the same clock. All tail pipeline registers must be triggered by the same clock. The clock can be dedicated or shared with other scan flip-flops.
- DFT Compiler accepts any clocking scheme that ensures correct scan shift. However, a good rule to follow is to have all the head pipeline registers triggered by the latest edge of the shift clocks and to have the tail registers triggered by the earliest edge of the shift clocks.
- All head pipelines must have the same depth, and all tail pipelines must have the same depth.
- Design your head pipeline registers to retain their values or propagate a constant value during the capture cycles for optimal ATPG results. This prevents unknown values from propagating to the unload data. Master-slave pipeline registers require particular

care. For more information, see [Avoiding X Capture in Head Pipeline Registers on page 764](#).

- The tail registers are assumed to have unknown values at the beginning of unload and do not need to maintain the state.

The user-defined pipeline register flow is available in both the DFT Compiler and TestMAX DFT tools.

Using Pipelined Scan Data With Scan Compression

The following topics describe considerations that apply to pipelined scan data in compressed scan flows:

- [Configuring Pipelined Scan Data in a Compressed Scan Flow](#)
- [Avoiding X Capture in Head Pipeline Registers](#)
- [Adding Pipeline Stages at the Compressor Inputs](#)

Configuring Pipelined Scan Data in a Compressed Scan Flow

The compressed scan flow with pipelining is similar to the regular scan flow. It follows the typical methodology of specify, preview, and insert. The tool wires the scan chain elements, and if necessary, inserts synchronization logic and generates appropriate test protocol files to be used in automatic test pattern generation (ATPG).

The following steps demonstrate a compressed scan flow with pipelining. This command sequence example applies to an unmapped design.

1. Read the design.

```
dc_shell> read_file -format verilog rtl.v
dc_shell> current_design top
dc_shell> link
```

2. Choose a scan style for your design.

```
dc_shell> set_scan_configuration \
           -style multiplexed_flip_flop
```

3. Perform a test-ready compile.

```
dc_shell> compile -scan
```

4. Specify scan clocks and other DFT signals.

```
dc_shell> for {set i 0} {$i < 3i} {incr i} {
           set_dft_signal -view spec \
```

```

        -type ScanDataIn -port SI_$i \
        -test_mode all

        set_dft_signal -view spec \
        -type ScanDataOut -port SO_$i \
        -test_mode all
    }

dc_shell> set_dft_signal -view spec \
        -type Reset -port resetn -active_state 0

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

```

5. Enable compressed scan and pipelined scan data.

```

dc_shell> set_dft_configuration \
        -pipeline_scan_data_enable \
        -scan_compression enable

dc_shell> set_scan_compression_configuration \
        -xtolerance default

```

6. Specify the scan architecture.

```
dc_shell> set_scan_configuration \
        -chain_count 32 -clock_mixing mix_clocks
```

7. Enable automatic pipeline register insertion, or specify user-defined pipeline register scan path connections.

- a. For automatically inserted pipeline registers:

```

dc_shell> set_pipeline_scan_data_configuration \
        -head_pipeline_clock CLK \
        -tail_pipeline_clock CLK \
        -head_pipeline_stages 1 \
        -tail_pipeline_stages 2 \
        -head_scan_flop true

dc_shell> set_scan_path chain0 -view spec \
        -scan_data_in SI_0 \
        -scan_data_out SO_0

        . . .

dc_shell> set_scan_path chain31 -view spec \
        -scan_data_in SI_31 \
        -scan_data_out SO_31

```

The `-head_scan_flop` option is used to prevent the head pipeline registers from capturing during scan capture.

- For user-defined pipeline register connections:

```
dc_shell> set_scan_path chain0 -view spec \
           -pipeline_head_registers \
           {head_pipe_0_reg} \
           -pipeline_tail_registers \
           {tail_stage1_pipe_0_reg tail_stage2_pipe_0_reg} \
           -scan_data_in test_SI_0 \
           -scan_data_out test_SO_0

           . . .

dc_shell> set_scan_path chain31 -view spec \
           -pipeline_head_registers \
           {head_pipe_31_reg} \
           -pipeline_tail_registers \
           {tail_stage1_pipe_31_reg tail_stage2_pipe_31_reg} \
           -scan_data_in test_SI_31 \
           -scan_data_out test_SO_31
```

- Generate a test protocol and check for design violations by running the test design rule checking at the gate level.

```
dc_shell> create_test_protocol
dc_shell> dft_drc
```

- Preview the scan structures.

```
dc_shell> preview_dft
```

- Build the scan structures into your design.

```
dc_shell> set_dft_insertion_configuration \
           -synthesis_optimization none \
           -preserve_design_name true

dc_shell> insert_dft
```

- Write out the pipeline-inserted netlist and the test protocol files.

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

dc_shell> report_scan_configuration

dc_shell> change_names -rules verilog -hierarchy

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

```
dc_shell> write -format verilog -hierarchy -output design.v
dc_shell> write_test_protocol \
           -output ScanCompression.spf \
           -test_mode ScanCompression_mode
dc_shell> write_test_protocol -output Scan.spf \
           -test_mode Internal_scan
```

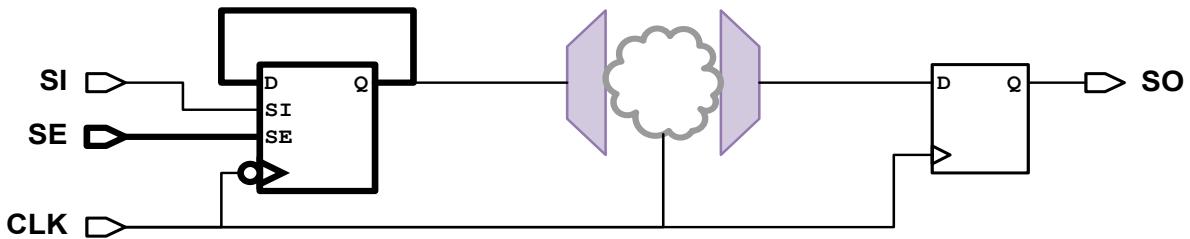
Avoiding X Capture in Head Pipeline Registers

When pipelined scan data is used in a compressed scan flow, the head pipeline registers should hold their state or capture a constant value during the capture cycle. This prevents unknown X values from being captured in the head pipeline registers, which would then get propagated through the decompression MUX and into the compressed scan chains.

The following methods can be used to capture known values in the head pipeline registers during scan capture:

- Capture the current register output state during scan capture, as shown in [Figure 320](#).

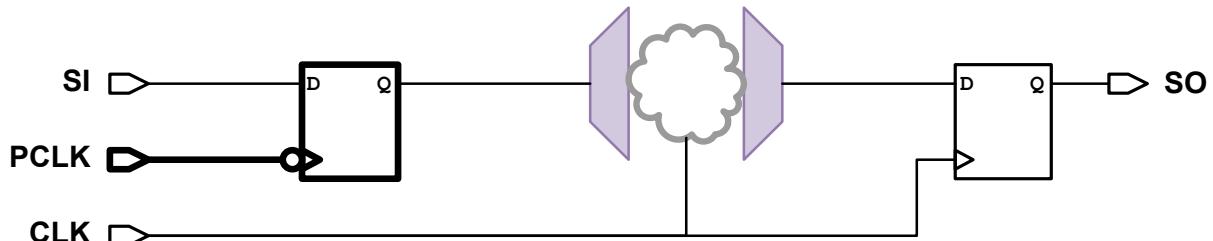
Figure 320 Head Pipeline Register With State-Holding Scan Flip-Flop



If you are using the automatic pipeline insertion flow, specify the `-head_scan_flop true` option of the `set_pipeline_scan_data_configuration` command. The tool will use scan head pipeline registers, and tie each register's output to its functional data input so that the state is held during scan capture.

- Use a dedicated head pipeline register clock, as shown in [Figure 321](#).

Figure 321 Head Pipeline Register With Dedicated Clock

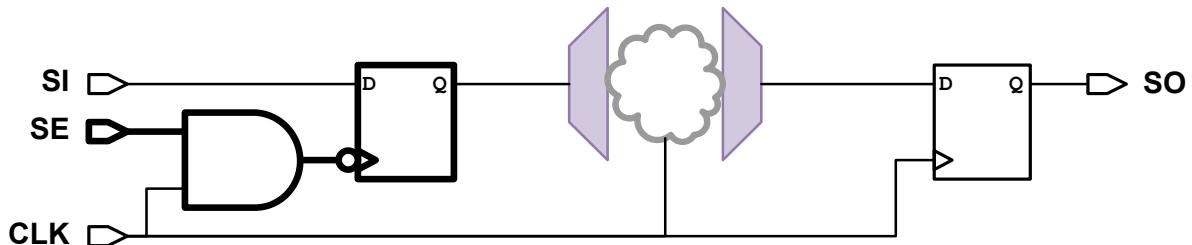


If you are using the automatic pipeline insertion flow, you can specify a dedicated head pipeline register clock using the `-head_pipeline_clock` option of the `set_pipeline_scan_data_configuration` command.

If the clock is independently controllable from the top level, you should add a constraint in the TestMAX ATPG tool to suppress the dedicated head pipeline register clock during scan capture.

- Use a gated head pipeline register clock, as shown in [Figure 322](#).

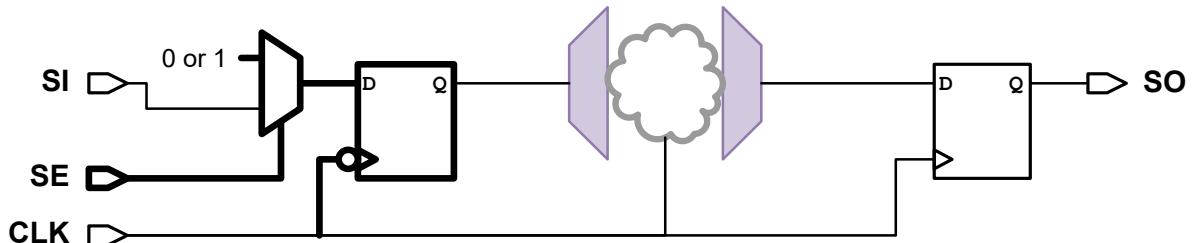
Figure 322 Head Pipeline Register With Gated Clock



There is no option to implement a gated clock in the automatic pipeline register insertion flow.

- Capture a constant value during scan capture, as shown in [Figure 323](#).

Figure 323 Head Pipeline Registers With Constant Value Capture



There is no option to implement a constant value capture in the automatic pipeline register insertion flow.

Post-DFT DRC verifies whether the head pipeline registers hold their values during capture and issues an R-18 violation message if the check is unsuccessful.

Master-Slave Pipeline Registers

When master-slave pipeline registers are used, take care to ensure that the slave register is a valid dependent slave of the master. To do this, ensure that the slave always captures new data from the master, so that the master flip-flop holds its state but the slave flip-flop does not.

For example, when the pipeline clock has a return-to-zero waveform, the master flip-flop is the rising edge triggered flip-flop, and the slave flip-flop is a falling edge triggered flip-flop immediately following it. At the end of every shift cycle, the master and slave flip-flops have exactly the same data.

Adding Pipeline Stages at the Compressor Inputs

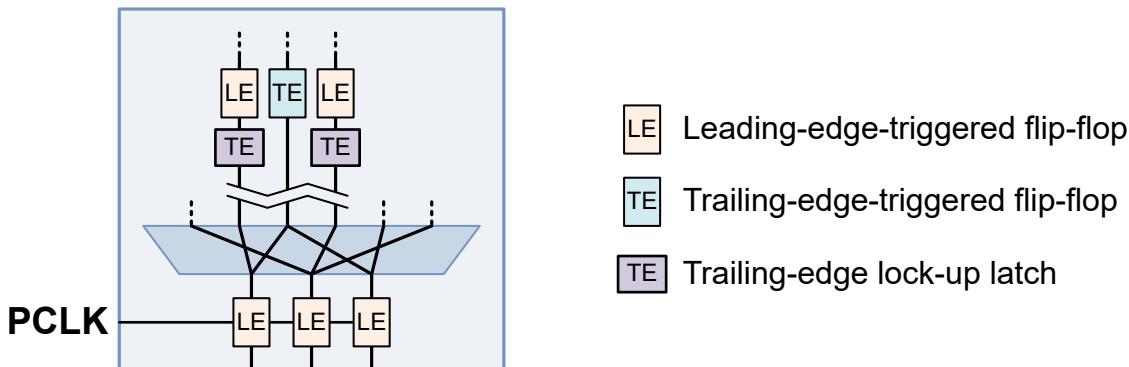
When tail scan data pipelining is used in a compressed scan flow, the compressor logic drives the tail pipeline registers. The tail pipeline registers are clocked on the leading pipeline clock edge.

If the last scan element of a compressed scan chain is clocked on the trailing clock edge, only a partial clock cycle is available for the compressor XOR logic. This occurs when

- The last scan element is clocked by the trailing edge of any clock.
- The last scan element is clocked by the leading edge of a clock other than the pipeline clock, requiring a lock-up latch to hold the data until the trailing edge.

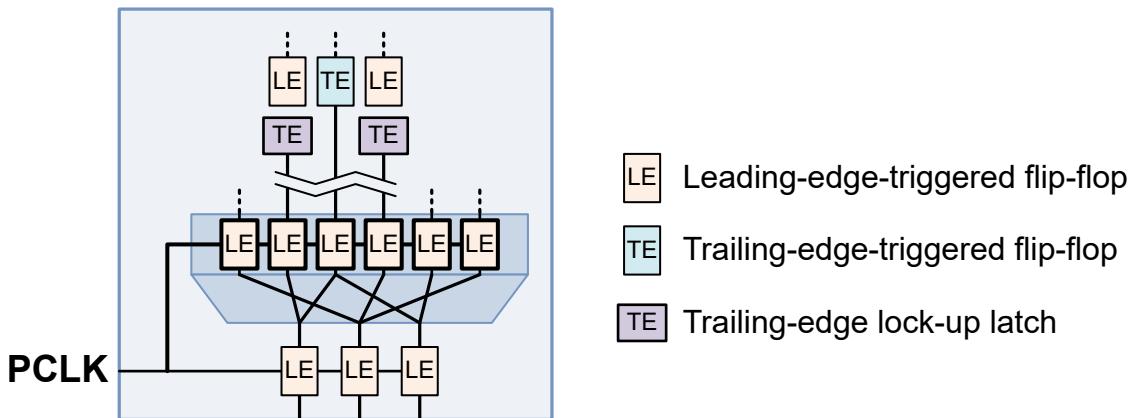
In addition, any long routing from the last scan element to the compressor also subtracts from the usable clock period. These scenarios are shown in [Figure 324](#).

Figure 324 Partial-Cycle Paths Through the Compressor Logic



To remedy this, you can add a stage of pipeline registers that are clocked by the same clock and edge as the tail pipeline registers to all compressor inputs, as shown in [Figure 325](#).

Figure 325 Full-Cycle Paths Through the Compressor Logic



The compressor input pipeline registers push any partial-cycle paths and long routes to the input side of these pipeline registers so that a full clock cycle is available for the compressor XOR logic. The pipeline registers are added inside the compressor design so that long routes remain on the input side when the compressor block is placed far away in layout. Pipeline registers are added at all compressor inputs to handle long routes when the last scan element is driven by the same clock as the tail pipeline registers.

To enable this feature, use the following option:

```
dc_shell> set_scan_compression_configuration -compressor_pipeline true
```

You can confirm that the pipeline registers are added by looking at the codec information in the `preview_dft` or `insert_dft` report. For example,

```
Architecting Load Decompressor (version 5.8)
Number of inputs/chains/internal modes = 6/30/3
Architecting Pipelined Unload compressor (version 5.8)
Number of outputs/chains = 6/30
```

This feature requires that tail scan data pipelining registers also be used. The compressor pipeline registers do not count against the tail scan data pipeline register depth.

Note the following limitations:

- Streaming compressed scan is not supported.
- Serialized compressed scan is not supported.
- End-of-chain retiming flip-flops can be used with pipelined compressor inputs, but the combination of these features adds extra retiming flip-flops and lockup latches that increase complexity without adding timing margin.

- Post-DFT DRC reports these pipeline register cells as nonscan cells. However, DRC in the TestMAX ATPG tool reports and uses them as scan cells.
- In some cases, post-DFT DRC reports an S19 violation on one of the compressor pipeline registers. This warning can be safely ignored.

Pipelined Scan Data Specifications

This topic covers the following:

- [Scan Architecture](#)
- [Scan Register Synchronization](#)

Scan Architecture

While implementing pipelined scan data logic, DFT Compiler takes the following aspects into consideration:

- Pipeline registers are automatically excluded from scan replacement. This exclusion takes precedence over any other scan membership specifications, such as the `set_scan_path` command and the `-include` and `-exclude` options of the `set_scan_configuration` command.
- In compressed scan modes, lock-up latches are inserted (as needed) at the beginning and/or end of the compressed chains. There is no optimization to minimize them by consolidating them outside the compression logic.
- Inversions in the pipelined scan data path are supported.
- DFT Compiler ignores pipeline registers during synthesis of compressed scan chains.

Scan Register Synchronization

DFT Compiler has the following requirements for scan synchronization:

- DFT Compiler checks whether synchronization between head pipeline registers and scan chains can be done. It also checks for synchronization between scan chains and tail pipeline registers. If there is a discrepancy, an error message is displayed and scan architecting or insertion is prevented.
- The clock signal triggering the lock-up element at the beginning of the chain is the inversion of the clock signal triggering the first flip-flop of the same chain. Similarly, the clock signal triggering the lock-up element at the end of the chain is the inversion of the clock signal triggering the last flip-flop of the chain.

Pipelined Scan Data Test Protocol Format

The test protocol file generated for the internal scan mode is the same as in a normal scan flow. The test protocol file generated for the scan compression mode contains information about the pipeline registers. The test protocol file has the following additional information:

- The number of head pipeline stages is indicated by the `LoadPipelineStages` keyword.
- The number of tail pipeline stages is indicated by the `UnloadPipelineStages` keyword.

For example,

```
CompressorStructures {
    LoadPipelineStages 3;
    UnloadPipelineStages 2;
    Compressor des2_U_decompressor {
        ModeGroup mode_group;
        LoadGroup load_group;
        CoreGroup core_group;
        Modes 3;
        ...
    }
    Compressor des2_U_compressor {
        UnloadGroup unload_group;
        CoreGroup core_group;
        Mode {{....}}
    }
}
```

Pipelined Scan Data Limitations

These are the requirements and limitations for implementing pipeline registers in a compressed scan flow:

- For maximum observability, the head pipeline flip-flops must hold state during the capture cycle to avoid capturing unknown X values.
- Compressed scan does not support unbalanced pipelining across chains. The number of head pipeline stages does not need to match the number of tail pipeline stages, but all decompressor inputs and all compressor outputs—including any associated with compressed clock chains—must have the same pipeline depth.
- If you are using external (uncompressed) chains, such as external clock chains or other user-defined external chains, their pipeline depths must match other scan chains. For more information, see [Excluding Scan Chains From Scan Compression on page 686](#).
- Scan-enable pipelining is independent of compressor scan data pipelining. Scan-enable signals can have any number of pipeline stages; however, the logic must be such that the `load_unload` and `capture` operations can be independently verified by DRC. For example, `test_setup` and/or `load_unload` preamble must set the design in

shift mode, so that when the scan-enable signal is at the nonshift value, the flip-flops are able to capture the system data.

- Any combinational logic between the scan ports and the pipeline registers must be sensitized to a known state.
- In designs with OCC controllers, an ATE clock cannot be used to directly clock head or tail pipelined scan data registers:
 - For DFT-inserted OCC controllers, when you specify the ATE clock as the pipeline register clock, the tool uses an OCC-controlled clock associated with the ATE clock. For details, see [SolvNet article 2685005, “How Are OCC Clocks Chosen for Pipelined Scan Data Registers?”](#)
 - For user-defined OCC controllers, if the ATE clock is manually connected to the pipeline registers, no DRC violations are reported, but incorrect ATPG patterns might be generated.
- Multiple test-mode operation with user-defined pipeline registers is not supported. However, multiple test-mode operation with automatically inserted pipeline registers is supported.
- User-defined pipeline registers and automatically inserted pipeline registers are mutually exclusive.

Hierarchical Flows With Pipelined Scan Data

In hierarchical flows, you can enable pipelined scan data at the chip level. When compressed scan cores already contain pipeline stages, the TestMAX DFT tool incrementally adds top-level pipeline stages as needed to meet the top-level pipeline depth target. This is known as *incremental pipelining*.

Consider a core created with a pipeline depth of one:

```
set_dft_configuration -pipeline_scan_data enable

set_pipeline_scan_data_configuration \
  -head_pipeline_stages 1 \
  -tail_pipeline_stages 1 \
  -head_pipeline_clock clk \
  -tail_pipeline_clock clk
```

When this core is integrated at the chip level, the chip-level pipeline depth is set to two:

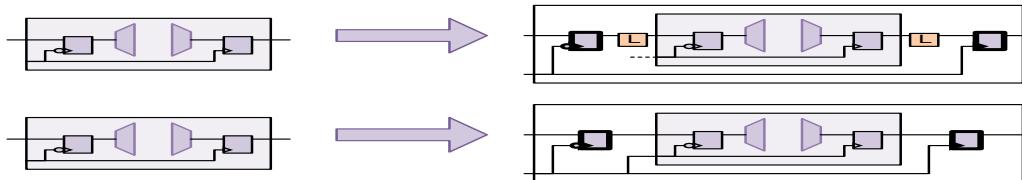
```
set_dft_configuration -pipeline_scan_data enable

set_pipeline_scan_data_configuration \
  -head_pipeline_stages 2 \
  -tail_pipeline_stages 2 \
```

```
-head_pipeline_clock clk \
-tail_pipeline_clock clk
```

[Figure 326](#) shows how the tool adds an additional pipeline stage to meet the chip-level target. The bolded flip-flops are added at the chip level during integration.

Figure 326 Hierarchical Pipelined Scan Data Example



At the core level, the pipeline registers can be driven by a shared functional clock or a dedicated pipeline clock. At the chip level, you typically connect this core-level clock pin to the desired chip-level clock signal. However, in the automatically inserted pipeline register flow, if you have a dedicated core-level pipeline clock pin that is unconnected at the chip level, the tool automatically connects it to the chip-level pipeline clock.

General Rules

The following general rules apply to all integration flows where cores are integrated at the chip level with pipelined scan data enabled:

- Pipelined scan data must be enabled when integrating pipelined cores, even if no additional pipeline stages are added at the top level.
- During chip-level integration, the tool adds pipeline stages as needed to meet the top-level pipeline depth specification.
- The only requirement for pipeline clock configuration is that the resulting pipeline scan path must meet scan-shift timing requirements (which the tool verifies before DFT insertion). Given this,
 - The head and tail pipeline clocks inside a core can differ.
 - The pipeline clock configuration can differ across cores.
 - Each higher integration level can use a different incremental pipeline clock.

Lockup latches are automatically inserted as needed.

- Unconnected core-level pipeline clock pins (whether DFT-created or user-defined) are automatically connected to the chip-level pipeline clock signal (whether DFT-created or user-defined).

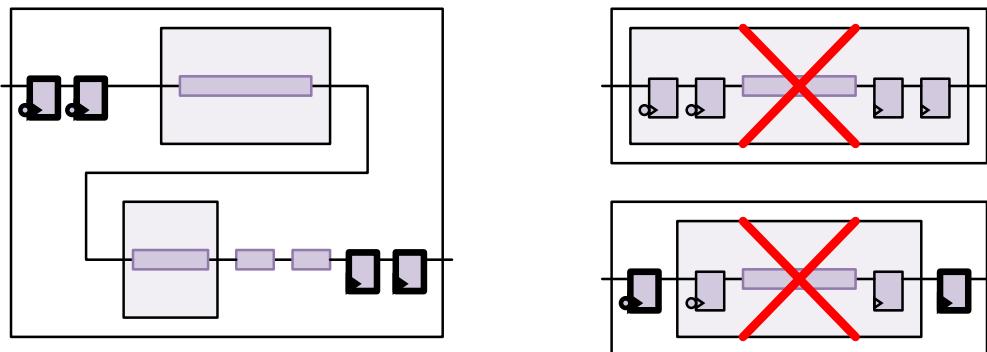
Pipelined Scan Data in the Standard Scan HSS Flow

In the standard scan HSS flow, note the following:

- DFT Compiler supports only unpipelined standard scan cores in hierarchical flows. Pipelined standard scan cores are not supported.
- The scan chains in standard scan cores can be used as scan segments that are mixed with other scan cells or scan segments.

[Figure 327](#) shows these properties.

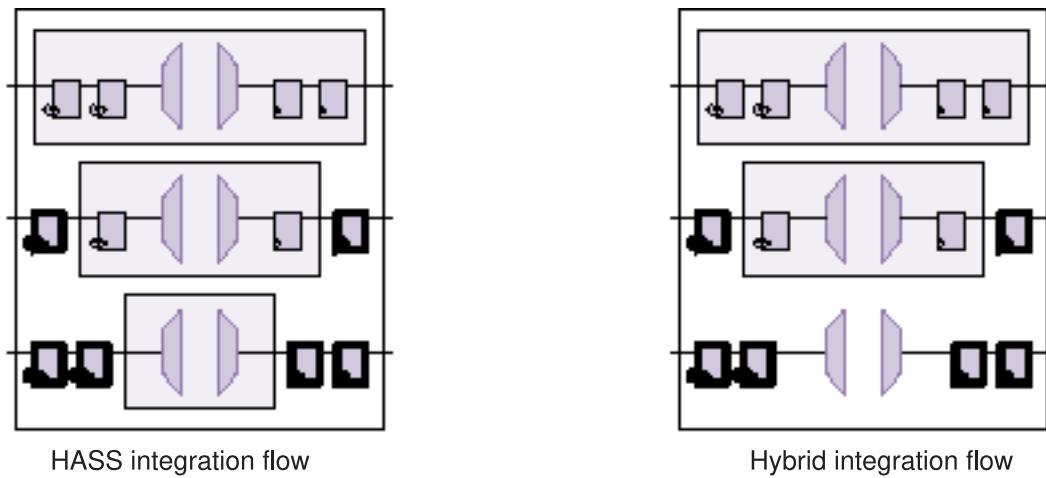
Figure 327 Integration Properties for the Pipelined Standard Scan HSS Flow



Pipelined Scan Data in the HASS and Hybrid Flows

[Figure 328](#) shows incremental pipelining in the HASS and Hybrid core integration flows. In the Hybrid flow, the tool also adds pipeline stages around the top-level codec to meet the top-level pipeline depth.

Figure 328 Incremental Pipelining in the HASS and Hybrid Flows



See Also

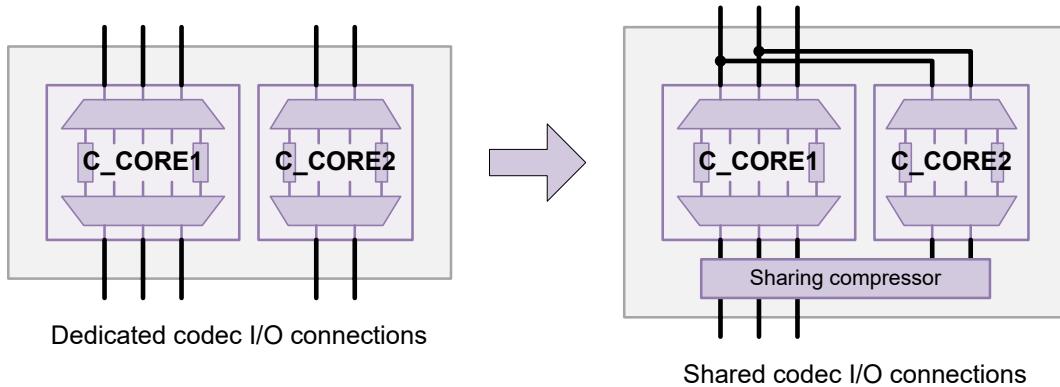
- [The HASS Flow on page 714](#) for more information about the HASS flow
- [The Hybrid Flow on page 720](#) for more information about the Hybrid flow

Sharing Codec Scan I/O Pins

DFTMAX compression provides design testability with reduced scan pin count requirements. However, the tool normally requires each codec to have dedicated scan-in and scan-out pin connections. For large designs with many blocks and many separate codecs, the scan pin requirements can still be challenging for pin-limited designs.

The tool allows multiple codecs to share the same scan-in and scan-out pins or ports in the HASS and Hybrid flows, as shown in [Figure 329](#). This is known as *codec I/O sharing*.

Figure 329 Dedicated Codec I/O Connections and Shared Codec I/O Connections



Codec I/O sharing provides the following features:

- I/Os can be shared across nonidentical cores and codecs of different widths.
- When codec inputs have dissimilar widths or when an increased number of scan inputs is provided, the codec input connections are allocated evenly across the scan inputs.
- Identical cores can use optimized shared I/O connections for improved testability.
- You can combine shared codec inputs with dedicated codec outputs.
- You can create multiple shared codec I/O groups.

For a list of the limitations of the shared codec I/O feature, see [Shared Codec I/O Limitations on page 815](#).

This topic covers the following:

- [Specifying the I/O Sharing Configuration](#)
- [Determining the Fully Shared I/O Configuration](#)
- [Codec I/O Sharing in the HASS Flow](#)
- [Codec I/O Sharing in the Hybrid Flow](#)
- [Codec I/O Sharing in the Top-Down Flat Flow](#)
- [Codec I/O Sharing With OCC Controllers](#)
- [Codec I/O Sharing With Identical Cores](#)
- [Codec I/O Sharing With Shared Codec Controls](#)
- [Codec I/O Sharing Groups](#)
- [Codec I/O Sharing and Standard Scan Chains](#)

- [Codec I/O Sharing and Pipelined Scan Data](#)
- [Integrating Cores That Contain Shared Codec I/O Connections](#)
- [Shared Codec I/O Limitations](#)

Specifying the I/O Sharing Configuration

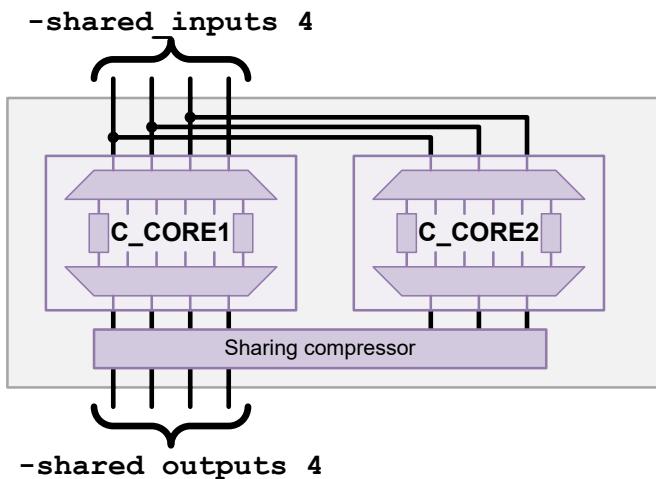
To share codec scan I/O pins, use the `-shared_inputs` and `-shared_outputs` options of the `set_scan_compression_configuration` command:

```
set_scan_compression_configuration
  -shared_inputs M
  -shared_outputs N
```

The value *M* specifies the size of the set of shared scan-in pins used for all codec input connections. The value *N* specifies the size of the set of shared scan-out pins used for all codec output connections. These values pertain only to the scan I/O signals needed for codec connections; scan I/O signals for external chains in compressed scan mode and for scan chains in standard scan mode should not be included in these values.

[Figure 330](#) shows two compressed scan cores with connections that are fully shared using the `set_scan_compression_configuration -shared_inputs 4 -shared_outputs 4` command. The first codec has four I/O pins and the second codec has three I/O pins. A minimum shared set of four top-level scan I/O pins is required to satisfy the connections of the wider codec. All shared scan inputs are tied together, and the scan outputs are combined with an output sharing compressor block.

Figure 330 Compressed Scan Mode Operation for Two Codecs With Fully Shared I/O



You can use the `-shared_inputs` option to set the number of shared scan inputs to any value from fully shared to unshared, inclusive. Similarly, you can use the

`-shared_outputs` option to set the number of shared scan outputs to any value from fully shared to unshared, inclusive. Values between the fully shared and unshared values are called *partially shared* values. For more information about computing the fully shared configuration, see [Determining the Fully Shared I/O Configuration on page 777](#).

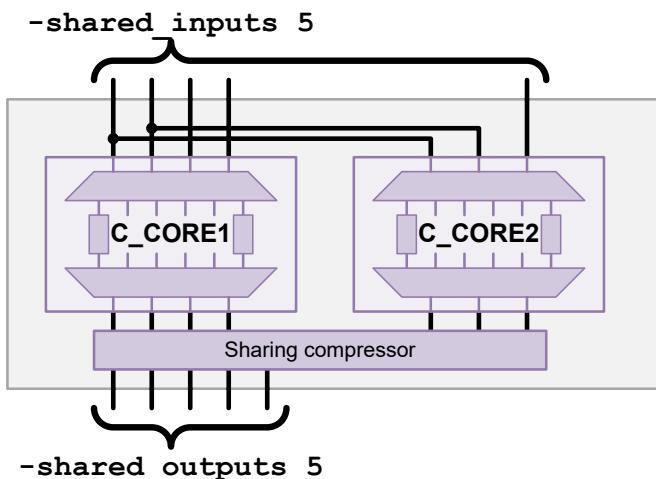
Note:

If you specify the fully unshared (dedicated) outputs value, you must use the flow described in [Specifying Shared Codec Inputs With Dedicated Codec Outputs on page 792](#).

To enable the codec I/O sharing feature, you must specify shared inputs with the `-shared_inputs` option. The `-shared_outputs` option by itself cannot enable codec I/O sharing, and you cannot share outputs without also sharing inputs. If you specify the `-shared_inputs` option without the `-shared_outputs` option, the codec outputs are automatically fully shared.

As you increase the number of shared scan inputs, the controllability of the compressed scan chains increases. As you increase the number of shared outputs, the observability of the compressed scan chains increases. [Figure 331](#) shows the same two codecs with partially shared inputs and outputs.

Figure 331 Two Codecs With Partially Shared Inputs and Outputs



If you provide too few scan inputs with the `-shared_inputs` option, the tool issues the following warning message and proceeds with the fully shared input value:

Warning: Your request for 3 shared codec scan inputs in partition default_partition cannot be met; 4 shared scan inputs will be used instead. (TEST-1420)

If you provide too few scan outputs with the `-shared_outputs` option, a similar warning message and adjustment occur:

Warning: Your request for 3 shared codec scan outputs in partition default_partition cannot be met; 4 shared scan outputs will be used instead. (TEST-1421)

DFT insertion places the output sharing compressor at the top level by default. To insert the sharing compressor inside a specific hierarchical block, specify the location using the `set_dft_location -include XOR_SELECT` command. For more information, see [Specifying a Location for DFT Logic Insertion on page 280](#).

Determining the Fully Shared I/O Configuration

The configuration with the minimum number of shared inputs and outputs for a design is called the *fully shared* configuration. It is described in more detail in the following topics:

- [Determining Shared Input Pin Types](#)
- [Adding High X-Tolerance Block-Select Pins](#)
- [Automatically Computing the Fully Shared Configuration](#)
- [Manually Computing the Fully Shared Configuration](#)

Note:

For simplicity, shared codec I/O figures outside this section do not separate input pins into different categories.

Determining Shared Input Pin Types

When you share the scan input connections of compressed scan codecs, their scan-in pins are categorized for sharing, as described in the following topics:

- [Scan-In Pins That Drive Compressed OCC Clock Chains](#)
- [Load Mode Scan-In Pins](#)
- [High X-Tolerance Enable Scan-In Pins](#)
- [Regular Scan-In Data Pins](#)

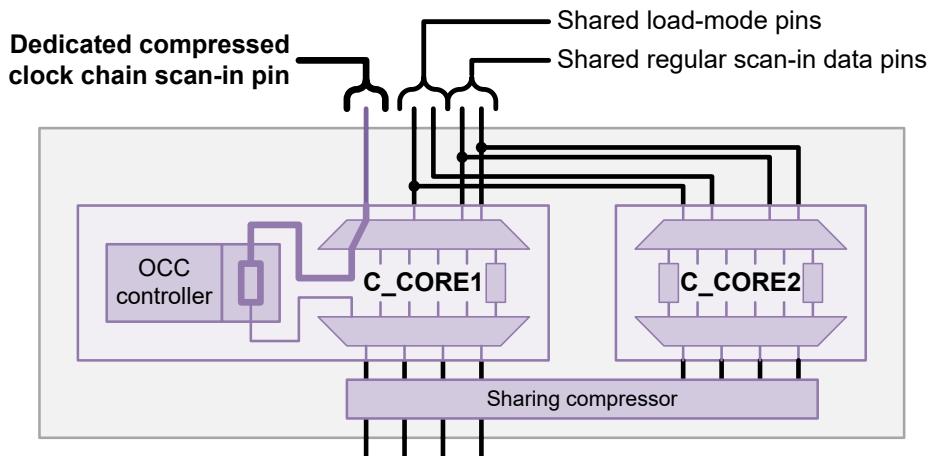
Scan-In Pins That Drive Compressed OCC Clock Chains

By default, when you insert scan compression in an OCC controller flow, the clock chain is compressed. DFTMAX compression dedicates a decompressor scan input path to the clock chain as shown in [Figure 284](#). This allows the clock chain values to be controlled without imposing constraints on other scan cells.

When you use codec I/O sharing, codec inputs that drive compressed clock chains cannot be shared. However, you must still include these inputs in the value provided to the `-shared_inputs` option. [Figure 332](#) shows two compressed scan cores, where one core,

C_CORE1, has a compressed clock chain driven by a dedicated codec scan input. The fully shared input value in this example is five.

Figure 332 Dedicated Scan Inputs for Compressed Clock Chains



If you are using the Hybrid flow with a top-level OCC controller and the top-level codec compresses the clock chain of the top-level OCC controller, you must include a scan input for the top-level codec.

If you are using external clock chains, do not include them in the value provided to the `-shared_inputs` option; they are excluded from scan compression.

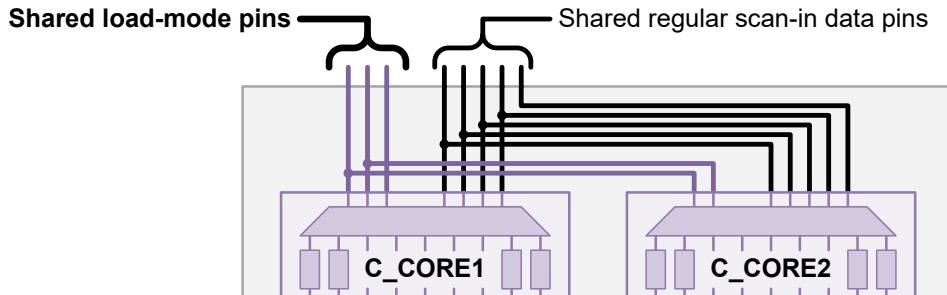
See Also

- [Codec I/O Sharing With OCC Controllers on page 787](#) For more information about using compressed clock chains and external clock chains with shared-I/O connections.

Load Mode Scan-In Pins

In a DFTMAX codec, some of the scan-in data pins are designated as load-mode pins. When you use codec I/O sharing, load-mode pins can be shared, but only with other load-mode pins. [Figure 333](#) shows the fully shared configuration for two cores with different load-mode pin counts.

Figure 333 Two Codecs With Different Load-Mode Pin Counts



See Also

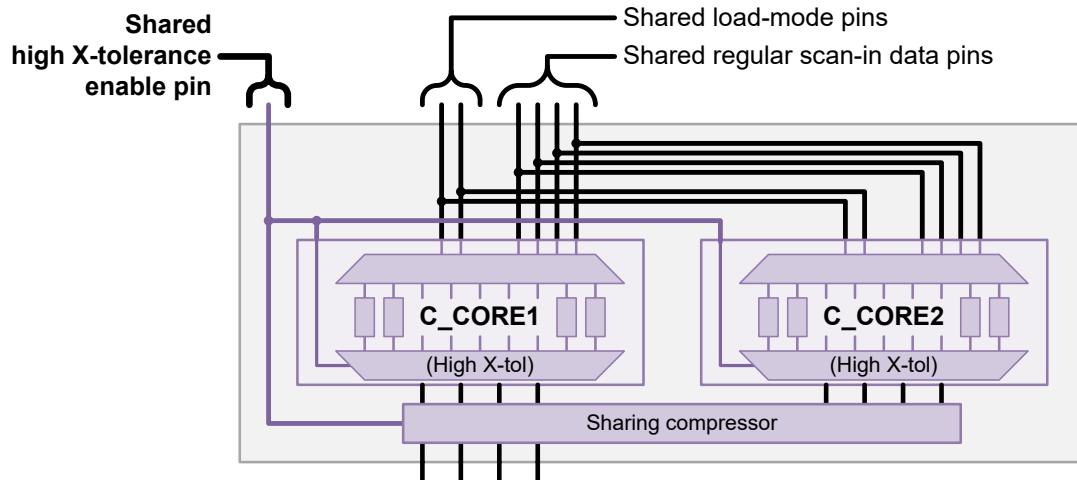
- [Decompressor Operation on page 657](#) for more information about load-mode pins

High X-Tolerance Enable Scan-In Pins

High X-tolerance enable scan-in pins enable high X-tolerance masking. Each high X-tolerance core or codec has only one of these pins. An output sharing compressor used with high X-tolerance cores or codecs also has one of these pins.

When you use codec I/O sharing, these high X-tolerance enable scan-in pins can be shared, but only with other high X-tolerance enable scan-in pins. [Figure 334](#) shows the fully shared configuration for two high X-tolerance cores.

Figure 334 Two Codecs With Shared High X-Tolerance Pins



Because codec I/O sharing requires that all codecs have the same X-tolerance type, these enable pins do not impose any complexity on the fully shared input computation. If high X-tolerance is used, all codecs have a single high X-tolerance enable pin, which is shared as other scan-in data pins are shared.

Note:

In this section, only the high X-tolerance enable connection to the output sharing compressor is shown for clarity. In other areas of the shared codec I/O documentation, it is omitted as the focus is on the block-select connections to the sharing compressor.

See Also

- [The High X-Tolerance Architecture on page 735](#) for more information about high X-tolerance enable pins

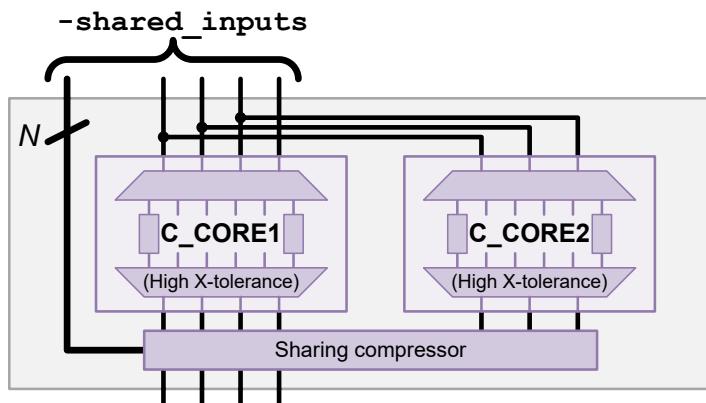
Regular Scan-In Data Pins

Regular scan-in data pins are all remaining pins that do not fall into the other three categories. When you use codec I/O sharing, regular scan-in data pins can be shared, but only with other regular scan-in data pins.

Adding High X-Tolerance Block-Select Pins

When you share the scan-out pins of high X-tolerance codecs, you must allow for additional scan-in pins to provide block-select X-masking signals for the output sharing compressor. [Figure 335](#) shows the additional block-select signal connections.

Figure 335 Adding Shared Inputs for High X-Tolerance Block-Select Signals



The number of additional block-select scan-in pins N is equal to \log_2 of the number of shared codecs, rounded up to the next integer value. This number must be included in the value provided to the `-shared_inputs` option. [Table 54](#) shows the number of additional scan-in pins required as a function of the number of shared codecs.

Table 54

Number of shared codecs	Required number of additional scan data inputs
2	1
3 to 4	2
5 to 8	3
9 to 16	4
17 to 32	5
33 to 64	6

The `preview_dft` command at the integration level reports the block-select signal connections that will be created. [Example 119](#) shows the preview report for two shared-I/O codecs.

Example 119 Shared Codec I/O Block Select Signals in preview_dft Report

ScanDataIn Ports:

```
(si) shows Regular ScanDataIn signal
(lm) shows Load Mode signal
(sel) shows Block Select signal
(xtol) shows Xtol Enable signal

test_si1 (drives C_CORE1/test_si1) (si)
test_si2 (drives C_CORE1/test_si2) (lm)
test_si3 (drives C_CORE1/test_si3) (lm)
test_si4 (drives C_CORE1/test_si4) (xtol)

test_si1 (drives C_CORE2/test_si1) (si)
test_si2 (drives C_CORE2/test_si2) (lm)
test_si4 (drives C_CORE4/test_si3) (xtol)

test_si5 (drives U_sharing_compressor/bsel[0]) (sel)
test_si4 (drives U_sharing_compressor/xtol_enable) (xtol)
```

If you are using dedicated (unshared) outputs by setting the `-shared_outputs` option to the fully unshared value, no shared codec I/O block-select signals are needed because there is no output sharing compressor.

Automatically Computing the Fully Shared Configuration

To automatically compute the fully shared configuration, use the `preview_dft` command. Specify a value of 1 for the `-shared_inputs` and `-shared_outputs` options, run the

`preview_dft` command, and obtain the values reported in the TEST-1420 and TEST-1421 warning messages.

For example,

```
dc_shell> set_scan_compression_configuration \
           -shared_inputs 1 -shared_outputs 1
dc_shell> preview_dft
...
Warning: Your request for 1 shared codec scan inputs in partition
default_partition cannot be met; 4 shared scan inputs will be used
instead. (TEST-1420)
Warning: Your request for 1 shared codec scan outputs in partition
default_partition cannot be met; 4 shared scan outputs will be used
instead. (TEST-1421)
```

This approach is useful when codec information is not readily available for cores or codecs, such as when you do not have the CTL models available in an integration flow or you are using the top-down flat insertion flow and the codecs do not yet exist.

Manually Computing the Fully Shared Configuration

This topic describes how to manually compute the fully shared configuration. You can use this approach if the preview report takes too long to generate or if you are designing core-level scan architectures to meet a particular top-level sharing goal.

Note:

If you are using the Hybrid integration mode, include the top-level codec in these computations.

To manually compute the fully shared input configuration value, which is the minimum value that can be specified with the `-shared_inputs` option, do the following:

1. If all codecs have the same load-mode pin count, compute the maximum scan-in width of all codecs to be shared, then go to step 3.
If you do not know this information, use the automatic computation method described in [Automatically Computing the Fully Shared Configuration on page 781](#).
2. If some codecs have different load-mode pin counts, do the following:
 - a. For each codec to be shared, separate the scan-in pins into load-mode pins and non-load-mode pins.
The non-load-mode pins are the remaining scan-in pins that are compressed clock chain scan-in pins, high X-tolerance enable pins, or regular scan-in data pins.
 - b. Compute the maximum load-mode pin count value across all codecs.

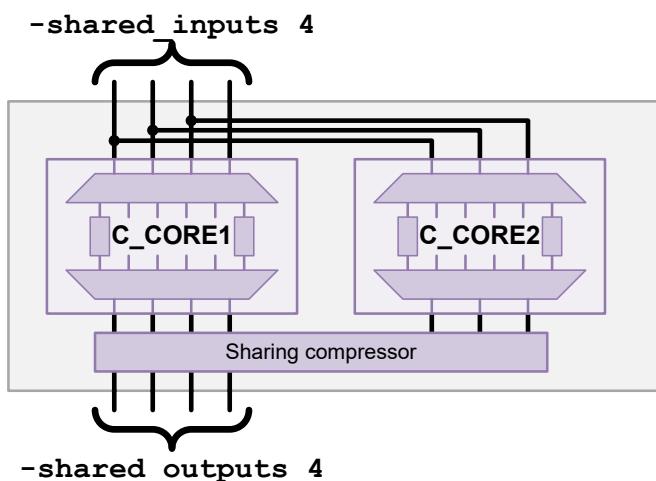
- c. Compute the maximum non-load-mode pin count value across all codecs.
- d. Add together the maximum values from steps 2b and 2c, then go to step 3.
- 3. If compressed clock chains are used, add the number of compressed clock chain scan-in pins because these pins cannot be shared.
- 4. If high X-tolerance is used, add N additional scan inputs for block-select signals, where N is equal to \log_2 of the number of shared codecs, rounded up to the next integer value. For more information, see [Adding High X-Tolerance Block-Select Pins on page 780](#).

To manually compute the fully shared output configuration value, which is the minimum value that can be specified with the `-shared_outputs` option, compute the maximum scan-out width of all codecs to be shared.

Codec I/O Sharing in the HASS Flow

In the HASS flow, one or more compressed scan cores are integrated at the top level. [Figure 336](#) shows two cores with fully shared codec I/O connections. The first codec has four I/O pins and the second codec has three I/O pins; the values for the `-shared_inputs` and `-shared_outputs` options of the `set_scan_compression_configuration` command are determined by the wider codec.

Figure 336 HASS Flow With Full Codec I/O Sharing

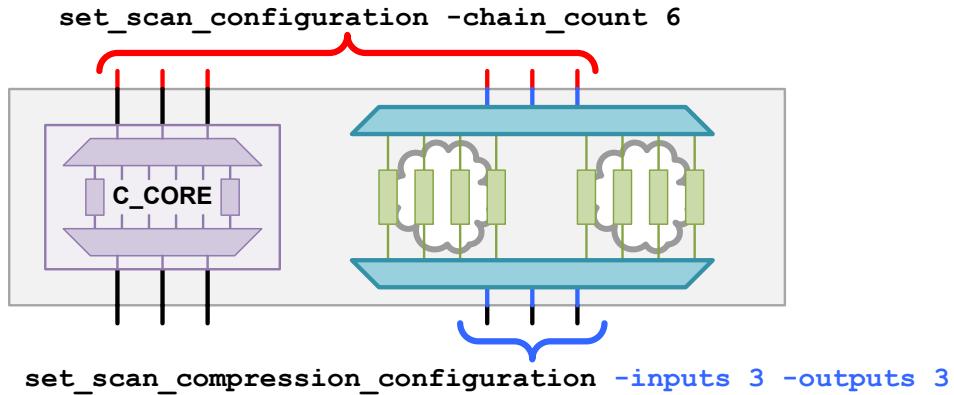


Codec I/O Sharing in the Hybrid Flow

In the Hybrid flow, a new codec is inserted at the top level along with existing block-level codecs. When the codec I/O sharing feature is not used, only the remaining scan data

pins not used by the existing block-level codecs are used for the new top-level codec. [Figure 337](#) shows a top-level design where the new top-level codec uses the remaining three available scan data I/O connections.

Figure 337 Hybrid Flow With Codec I/O Sharing Disabled



When codec I/O sharing is performed, the new top-level codec can use some or all of the scan input pins used by the existing block-level codecs. [Figure 338](#) shows an example of partial scan-in sharing and [Figure 339](#) shows an example of full scan-in sharing.

Figure 338 Hybrid Flow With Some Codec Input Sharing

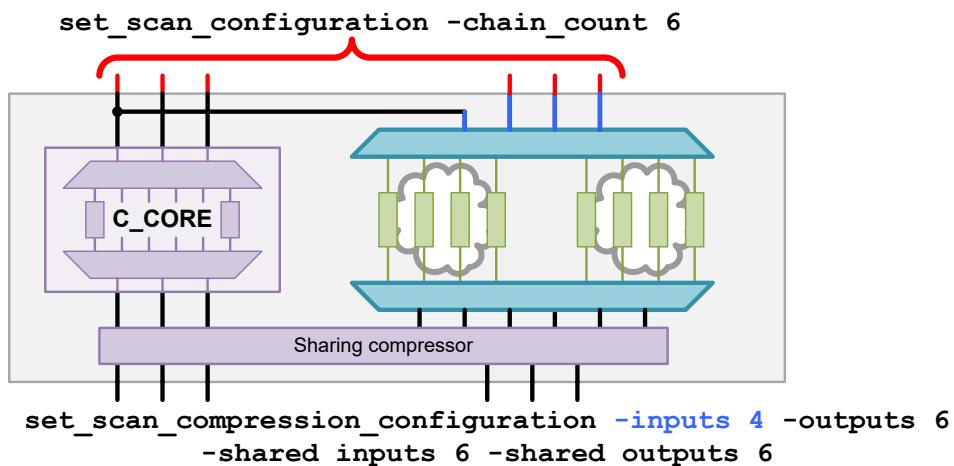
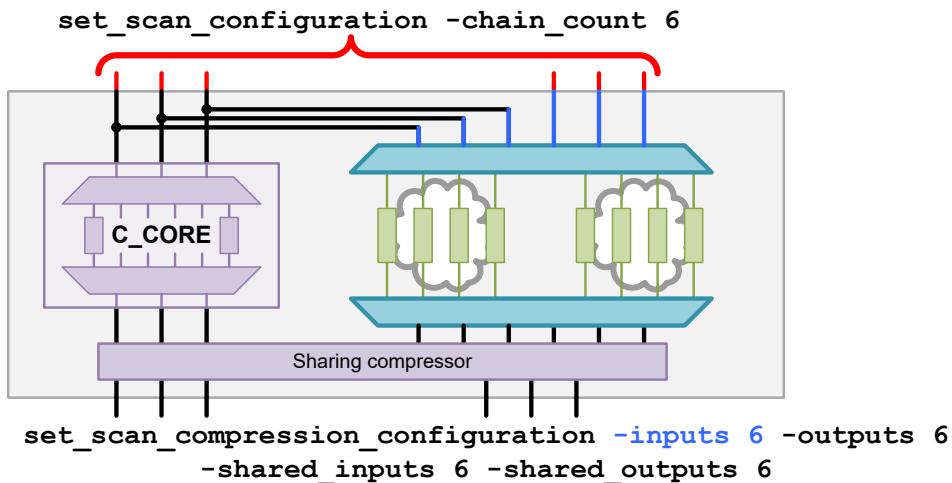


Figure 339 Hybrid Flow With Full Codec Input Sharing



The I/O sharing feature provides a degree of freedom in the construction of the new top-level codec. Because the top-level codec characteristics can no longer be derived from other scan configuration information in the Hybrid flow, you must explicitly configure the top-level codec characteristics using the `-inputs` and `-outputs` options of the `set_scan_compression_configuration` command.

Codec I/O Sharing in the Top-Down Flat Flow

In a normal top-down flat flow, multiple codecs are created by defining multiple DFT partitions. Each partition specifies the scan logic to be scan-compressed with a codec. However, each DFT partition is an independent scan context with its own scan configuration. This prevents the codec I/O sharing feature from being applied as follows:

- Scan I/O ports can only belong to one partition at a time; they cannot be shared across partitions.
- There is no single partition to attach the codec I/O sharing specification.

To remedy this, the top-down flat flow uses *subpartitions* to enable codec I/O sharing. A subpartition is defined with the `define_dft_partition` command, and specifies the scan logic to be scan-compressed with a codec. These subpartitions are then grouped together into top-level partitions, which can receive codec I/O sharing and scan port specifications.

This flow is configured as follows:

1. Define the subpartitions containing the logic to be scan-compressed with a codec.
2. Define a top-level partition containing the subpartitions that should share their codec I/O connections.

3. Configure codec characteristics within each subpartition.
4. Configure the shared codec I/O characteristics within each top-level partition.

The following example defines two codecs with shared I/O connections in a top-down flat flow:

```
# globally enable scan compression
set_dft_configuration -scan_compression enable

# define subpartitions that define codecs
define_dft_partition SUB_P1 -include {BLK1}
define_dft_partition SUB_P2 -include {BLK2}

# define top-level partition that groups subpartition codecs together
define_dft_partition PARTITION -include {SUB_P1 SUB_P2} ;# subpartitions

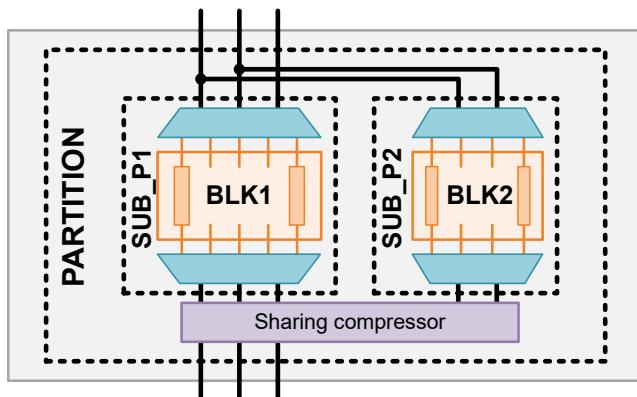
# apply subpartition codec characteristics
current_dft_partition SUB_P1
set_scan_configuration -chain_count 2
set_scan_compression_configuration -chain_count 5 -inputs 3 -outputs 3

current_dft_partition SUB_P2
set_scan_configuration -chain_count 1
set_scan_compression_configuration -chain_count 4 -inputs 2 -outputs 2

# apply top-level codec I/O sharing characteristics
current_dft_partition PARTITION
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3
```

[Figure 340](#) shows the top-down flat DFT insertion results for these commands.

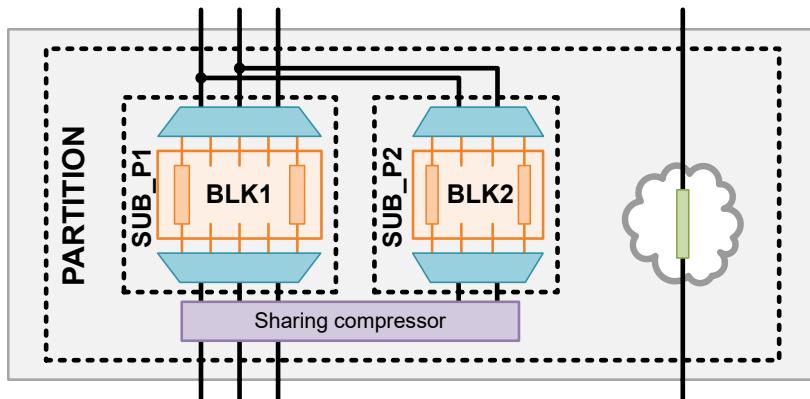
Figure 340 Top-Down Flat Flow Codec I/O Sharing



When you use the shared codec I/O feature in a top-down flat flow, all scan logic to be compressed must be placed into subpartitions. If scan cells in a partition exist outside a

subpartition, the tool places them into an external chain that is not compressed inside that partition. See [Figure 341](#).

Figure 341 Partition-Level External Chains in Top-Down Flat Flow



Note:

Subpartition definitions are only supported in a shared codec I/O flow.

An enclosing top-level partition is required because this is the single-group case of the functionality described in [Defining Sharing Groups in the Top-Down Flat Flow on page 802](#).

See Also

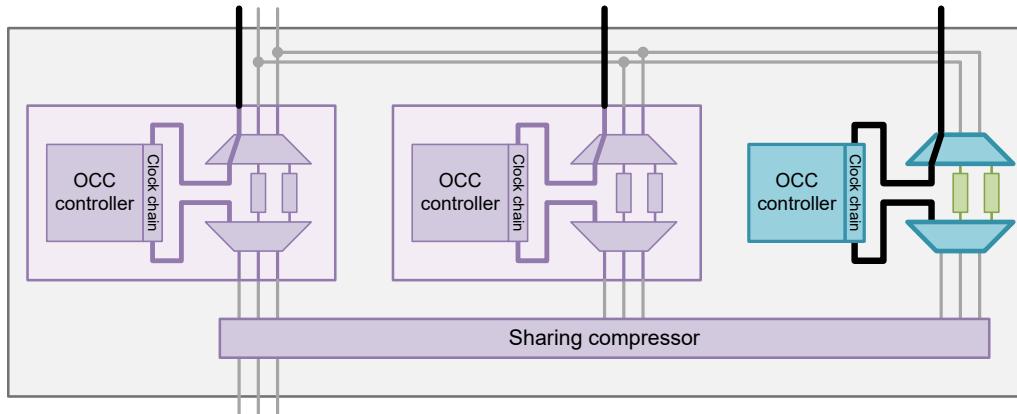
- [Top-Down Flat Compressed Scan Flow With DFT Partitions on page 668](#) for more information about defining DFT partitions

Codec I/O Sharing With OCC Controllers

In a DFTMAX flow, when you insert scan compression in a design with OCC controllers, the clock chain is placed between the decompressor and compressor by default, as described in [Using Compressed Clock Chains on page 688](#).

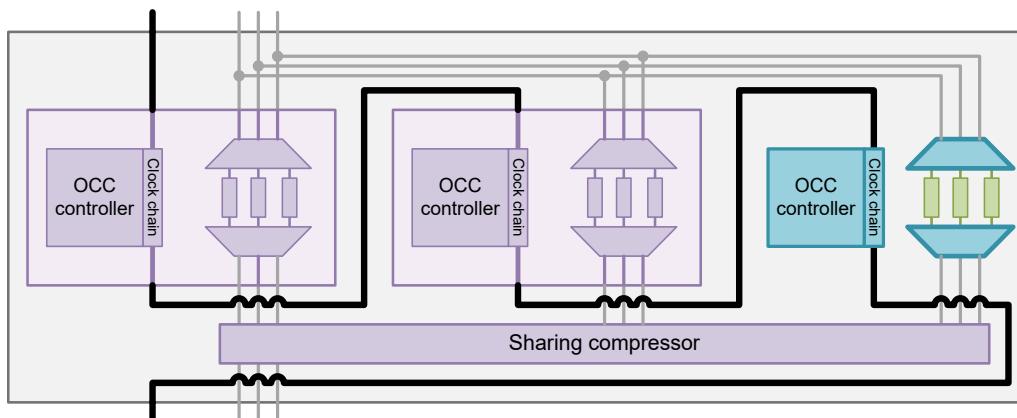
However, compressed clock chain inputs cannot be shared. This results in a dedicated scan-in connection for each dedicated clock chain codec input in the design, as shown in [Figure 342](#).

Figure 342 Shared Codec I/O With Compressed Clock Chains



To avoid these unsharable clock chain inputs across shared-I/O cores, you can use external clock chains, which can be concatenated into a single top-level clock chain across the cores as shown in [Figure 343](#).

Figure 343 Shared Codec I/O With External Clock Chains



Although external clock chains require dedicated scan I/Os in each core, they can be concatenated into a single top-level clock chain, reducing the total scan I/O requirements at the top level.

Note:

If you are using DFT partitions, all clock chains to be concatenated must belong to the same partition. See [SolvNet article 2675107, “Concatenating OCC Clock Chains From Multiple DFT Partitions.”](#)

For details on defining external clock chains, see [Defining External Clock Chains on page 689](#).

Codec I/O Sharing With Identical Cores

Codec I/O sharing does not require that cores be identical. However, if you are integrating multiple identical instances of a core, DFTMAX compression can take advantage of their identicity to optimize their scan-in and scan-out connections for improved testability.

Codec I/O sharing with identical cores is described in the following topics:

- [Identical Core Connections](#)
- [Specifying Identical Cores](#)
- [Using Scrambled Output Connections](#)
- [Specifying Shared Codec Inputs With Dedicated Codec Outputs](#)

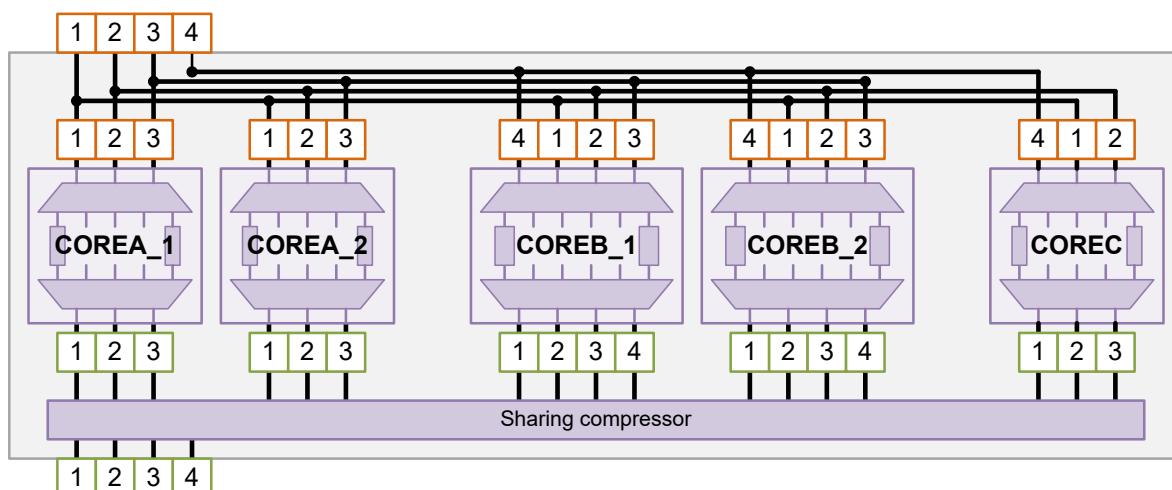
Identical Core Connections

Identical cores have the same functional and scan logic structures. Because of this, DFTMAX compression optimizes shared codec I/O connections of identical cores as follows:

- Identical scan input connections are used so that each pattern sensitizes the same faults across all cores.
- Identical scan output connections are used to reduce the impact of X values on pattern count.

You can have multiple groups of identical cores, and you can mix identical cores with nonidentical (unique) cores. [Figure 344](#) shows the codec scan data connections for two instances of COREA, two instances of COREB, and a single instance of COREC.

Figure 344 Codec Scan Data Connections for Groups of Identical Cores

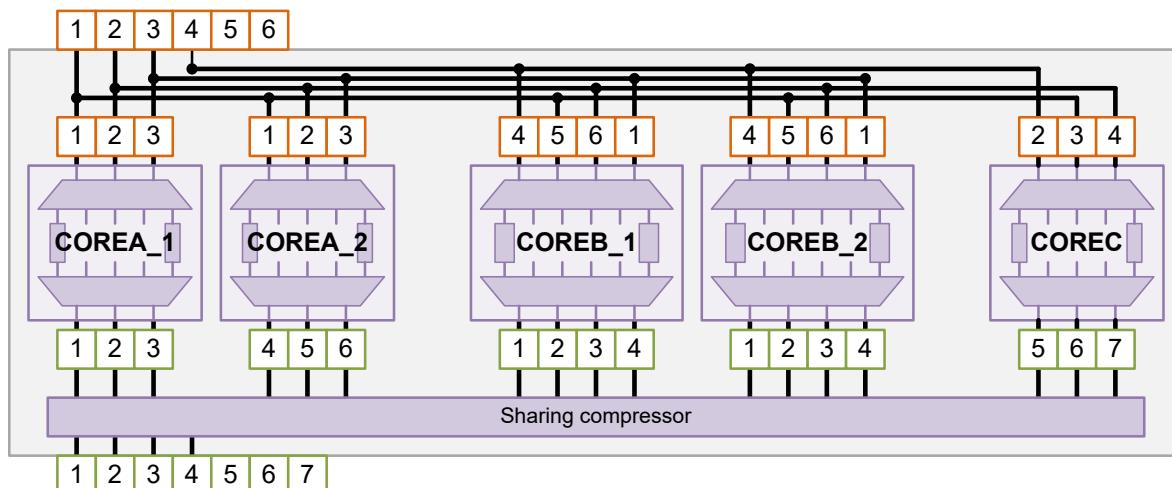


The tool uses as many shared inputs and outputs as possible. For each identical core group,

- The core inputs use all available shared scan inputs in sequence.
- The core outputs start at the first scan output, using the largest number of shared outputs that is a multiple of the core output width.

[Figure 345](#) shows the previous example modified to use a larger number of shared inputs and outputs.

Figure 345 Codec Scan Data Connections With More Shared Inputs and Outputs



Specifying Identical Cores

The tool does not identify identical core instances by default; you must specify them with the `-identical_cores` option of the `set_scan_compression_configuration` command. Wildcards are supported. For example,

```
dc_shell> set_scan_compression_configuration \
           -shared_inputs 4 -shared_outputs 4 \
           -identical_cores {COREA_* COREB_*}
```

If there are multiple groups of identical cores, the tool analyzes them to determine the identicity grouping. Cores are considered identical when the following codec parameters match:

- Decompressor input width
- Compressor output width
- X-tolerance configuration
- Compressed scan chain count

The `preview_dft` and `insert_dft` commands print information messages that show the identical core groups:

```
Information: Detected group of identical cores: COREA_1 COREA_2
(TEST-1450)
Information: Detected group of identical cores: COREB_1 COREB_2
(TEST-1450)
```

You can use these messages to confirm that identical cores are identified as expected.

When you run TestMAX ATPG, use the `-shared_io_analysis` option of the `set_atpg` command. This option performs an analysis to identify all identical circuit networks in the identical cores, which results in improved pattern generation.

Using Scrambled Output Connections

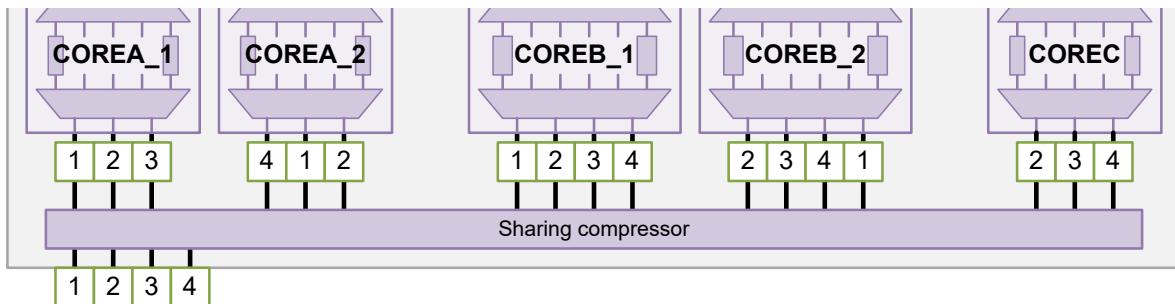
If your design generates few X values, you can use nonidentical (scrambled) scan output connections for the identical cores to potentially improve the diagnosability of the design.

To do this, set the `-scramble_identical_outputs` option to `true`:

```
dc_shell> set_scan_compression_configuration \
    -shared_inputs 4 -shared_outputs 4 \
    -identical_cores {COREA_* COREB_*} \
    -scramble_identical_outputs true
```

Scrambled output connections, shown in [Figure 346](#), provide more uniqueness in the output signatures of each identical core.

Figure 346 Using Scrambled Output Connections for Identical Cores



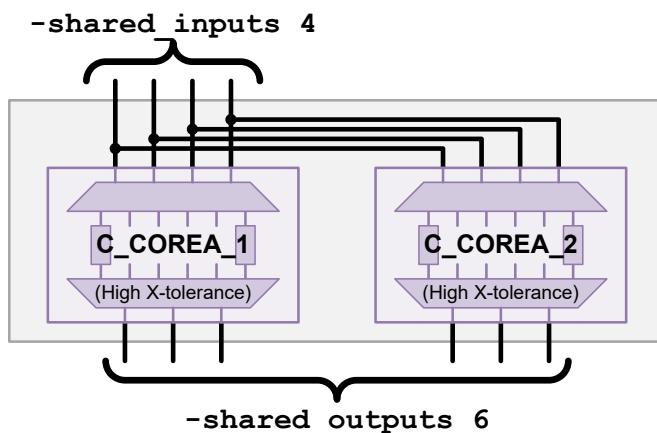
For each identical core group, the core outputs start at the first scan output, using all available shared scan outputs in sequence. When the shared output width is a multiple of an identical core's output width, the sequence advances as needed to avoid identical connections as much as possible.

Specifying Shared Codec Inputs With Dedicated Codec Outputs

You might want to use identical shared codec input connections for identical cores while still using dedicated codec output connections. This configuration reduces the scan-in pin requirements while providing full output observability with no sharing compressor required.

[Figure 347](#) shows an example of this sharing configuration.

Figure 347 Example of Shared Codec Inputs With Dedicated Codec Outputs



To use dedicated codec outputs, the following requirements must be met:

- All cores are identical in a single group (according to the criteria described in [Codec I/O Sharing With Identical Cores on page 789](#)).
- The value specified for the `-shared_inputs` option is the fully shared value (equal to the shared codec input width).
- The value specified for the `-shared_outputs` option is the fully unshared value (equal to the sum of all shared codec output widths).

When the tool detects that these requirements are met, it issues the following information message:

Information: You have asked for shared codec inputs and dedicated codec outputs in partition *partition_name*. (TEST-1446)

Note:

In this flow, the `-identical_cores` option is optional because all cores must be identical.

Because there is no sharing of codec outputs, no sharing compressor is required. This also means that no additional shared codec I/O block-select signals must be included in the `-shared_inputs` value when using high X-tolerance. For more information about block-select signals, see [Adding High X-Tolerance Block-Select Pins on page 780](#).

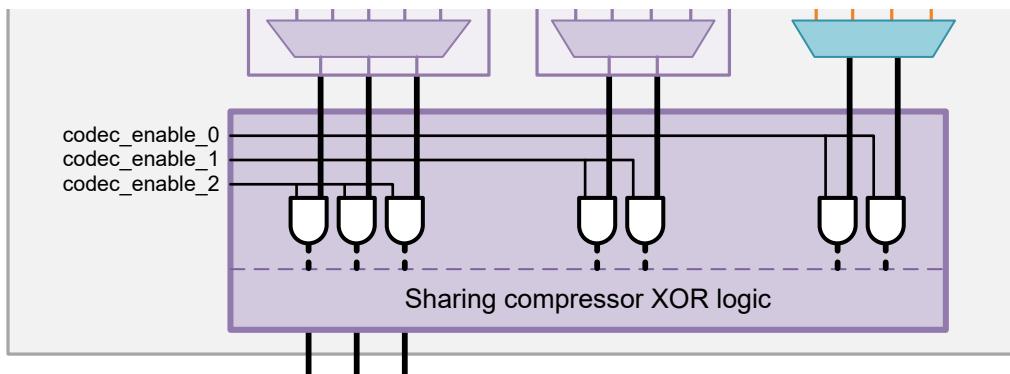
Codec I/O Sharing With Shared Codec Controls

In some cases, you might want to exclude one or more blocks from testing. For example,

- Test blocks individually to perform “core harvesting,” in which redundant blocks take the place of bad blocks to improve device yield
- Exclude a single block from testing that has known issues or defects

You can implement shared codec control logic to selectively enable observation of shared-I/O codecs. The tool adds AND-gating logic at the codec inputs of output sharing compressors, as shown in [Figure 348](#). After DFT insertion, you can write a test protocol that selectively disables any set of codecs, provided at least one codec is active in the design across all DFT partitions.

Figure 348 Output Sharing Compressor With Codec Control Logic



Note the following:

- Disabled codecs are omitted from the SPF. In the TestMAX ATPG tool, scan elements associated with disabled codecs can drive X values.
- To block external X values, test wrapped cores in their inward-facing (INTEST) mode.
- The codec controls logic blocks observation at disabled compressors. However, disabled blocks can still receive the clock, scan-in, and scan-enable signals and operate as constructed during gate-level simulation and manufacturing test.

Configuring Shared Codec Controls

To implement codec control logic, enable the following option:

```
dc_shell> set_scan_compression_configuration -shared_codec_controls true
```

When using DFT partitions, you can enable this option globally or on a per-partition basis.

The `preview_dft` command reports the codec enable pins to be implemented for each output sharing compressor:

```
dc_shell> preview_dft
...
*****
* Shared Codec Controls Report *****
U_sharing_compressor/codec_enable_0 controls
  sub1:sub_1_U_decompressor_ScanCompression_mode
U_sharing_compressor/codec_enable_1 controls
  sub2:sub_2_U_decompressor_ScanCompression_mode
U_sharing_compressor/codec_enable_2 controls
  top:top_default_partition_U_decompressor_ScanCompression_mode
*****
```

The `insert_dft` command makes the codec enable pin connections as follows:

- If you are using IEEE 1500 test-mode control, the tool automatically implements TMCDR register bits that drive the codec enable pins.
- If you are not using IEEE 1500 test-mode control, you can define internal hookup pins for the codec enable signals, as described in [Specifying User-Defined Codec Enable Signals on page 795](#).

If you do not define any codec enable signals, the tool connects the codec enable pins to logic 1, and you must modify the netlist to control the codec enable pins.

After DFT insertion, use the `write_test_protocol` command to write an SPF that disables one or more codecs by specifying a codec list with the `-disable_codecs` option. Reference each codec's decompressor using the `cell_name:decompressor_name` syntax. For example,

```
# write a protocol that tests all codecs
write_test_protocol \
  -test_mode ScanCompression_mode \
  -output COMP_enable_all_codecs.spf

# write a protocol that tests only the top-level codec
# by disabling all others
write_test_protocol \
  -test_mode ScanCompression_mode \
  -output COMP_disable_core_codecs.spf \
  -disable_codecs {sub1:sub_1_U_decompressor_ScanCompression_mode \
  sub2:sub_2_U_decompressor_ScanCompression_mode}
```

If you are using IEEE 1500 test-mode control, the `write_test_protocol` command writes a test protocol that automatically configures the TMCDR codec selection bits as needed.

If you are not using IEEE 1500 test-mode control, the tool assumes that you are modifying the test protocol and/or netlist to disable the specified codecs. The `write_test_protocol` command issues a reminder as follows:

Information: The codec control pins of the specified disabled codecs must be driven by logic 0. (TEST-1473)

The codec control information is stored in the .ddc file written for the current design. You can read the .ddc file back in to generate additional test protocols at a later time.

See Also

- [Shared Codec I/O Limitations on page 815](#) for limitations of shared codec controls
- [Integrating Shared I/O Cores That Contain Shared Codec Controls on page 814](#) for more information about integrating cores that have shared codec controls

Specifying User-Defined Codec Enable Signals

If you are using the shared codec controls feature, you can connect the codec enable signals to your own internal hookup pins by defining DFT signals with a type of `codec_enable`. For example,

```
dc_shell> set_dft_signal -view spec -type codec_enable \
    -hookup_pin MY_EN_reg[1]/Q
dc_shell> set_dft_signal -view spec -type codec_enable \
    -hookup_pin MY_EN_reg[0]/Q
```

The codec enable signals must be defined using the `-hookup_pin` option; they cannot be defined using the `-port` option. If you are using DFT partitions, the signals must be specified on a per-partition basis; you cannot define global signals across to be allocated all partitions.

If there are fewer signals than codecs being controlled in the group, the tool ignores the signals, issues a TEST-1482 warning, and connects the codec enable pins to logic 1.

By default, the tool chooses a codec for each codec enable signal. To specify which codec is controlled by each signal, use the `-codec` option to reference each codec's decompressor using the `cell_name:decompressor_name` syntax:

```
dc_shell> set_dft_signal -view spec -type codec_enable \
    -hookup_pin MY_EN_reg[1]/Q \
    -codec sub2:sub_2_U_decompressor_ScanCompression_mode
dc_shell> set_dft_signal -view spec -type codec_enable \
    -hookup_pin MY_EN_reg[0]/Q \
    -codec sub1:sub_1_U_decompressor_ScanCompression_mode
```

The `-codec` option can only be used together with the `-type codec_enable` option.

Codec I/O Sharing Groups

You can use DFT partitions to share I/O connections within multiple codec groups. You can use this flow to reduce routing congestion by only sharing the connections of codecs that are in close proximity to each other or to optimize the sharing arrangement for identical cores.

DFT insertion places each partition's sharing compressor at the top level by default. To insert a partition's sharing compressor inside a specific hierarchical block, specify the location using the `set_dft_location -include XOR_SELECT` command applied within that partition's configuration. For more information, see [Specifying a Location for DFT Logic Insertion on page 280](#).

This topic covers the following:

- [Defining Sharing Groups in the HASS Flow](#)
- [Defining Sharing Groups in the Hybrid Flow](#)
- [Defining Sharing Groups for Codecs in Partitioned Cores](#)
- [Defining Sharing Groups in the Top-Down Flat Flow](#)

Defining Sharing Groups in the HASS Flow

In the HASS flow, create and configure sharing groups as follows:

1. Define DFT partitions containing the compressed scan cores that are to share their codec I/O connections.
2. Configure the shared codec I/O characteristics within each partition.

The following example defines two DFT partitions, where each contains two compressed scan cores:

```
# globally enable scan compression and HASS integration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -integration_only true

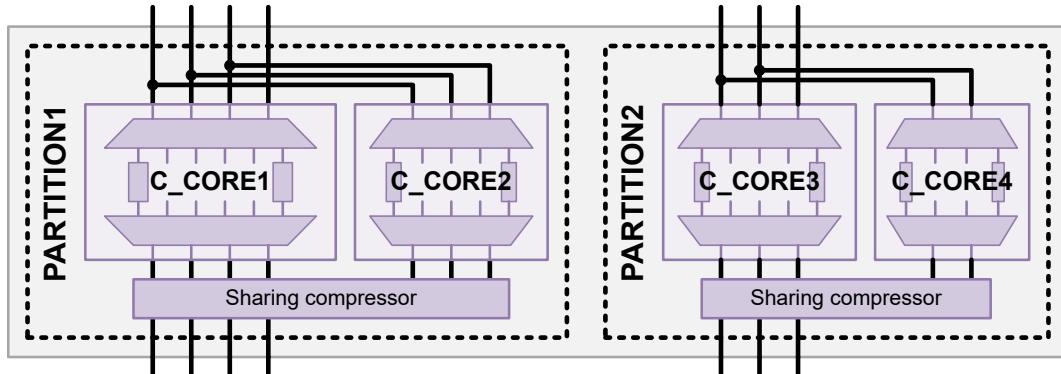
# define partitions that group cores
define_dft_partition PARTITION1 -include [list C_CORE1 C_CORE2]
define_dft_partition PARTITION2 -include [list C_CORE3 C_CORE4]

# apply codec I/O sharing characteristics
current_dft_partition PARTITION1
set_scan_configuration -chain_count 4
set_scan_compression_configuration -shared_inputs 4 -shared_outputs 4

current_dft_partition PARTITION2
set_scan_configuration -chain_count 3
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3
```

[Figure 349](#) shows the HASS integration results for these commands.

Figure 349 HASS Flow Codec I/O Sharing With Partitions



Defining Sharing Groups in the Hybrid Flow

In the Hybrid flow, there are one or more compressed scan cores along with additional top-level logic to be compressed. You can create one or more top-level codecs to compress this logic, and you can choose whether or not to share the connections of each top-level codec, as described in the following topics:

- [Using Shared Codecs for Top-Level Logic](#)
- [Using Dedicated Codecs for Top-Level Logic](#)

You can define multiple partitions to create multiple top-level codecs, including a mix of shared and dedicated top-level codecs.

Note:

A top-level codec is a codec that compresses top-level logic, which is logic that exists outside a compressed scan core. A top-level codec can be defined in a subpartition (for a shared top-level codec) or in a top-level partition (for a dedicated top-level codec).

Using Shared Codecs for Top-Level Logic

You can define subpartitions to specify the top-level scan logic to be scan-compressed with a shared codec. For more information about subpartitions, see [Codec I/O Sharing in the Top-Down Flat Flow on page 785](#).

In this flow, create and configure sharing groups as follows:

1. Define any subpartitions containing the logic to be scan-compressed with a top-level shared codec.
2. Define top-level partitions containing the compressed scan cores and/or subpartitions that are to share their codec I/O connections.

3. Configure the top-level logic codec characteristics within each subpartition.
4. Configure the shared codec I/O characteristics within each top-level partition.

The following example defines two top-level DFT partitions, each of which contains two shared-I/O codecs:

```
# globally enable scan compression and Hybrid integration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -hybrid true

# define subpartitions that define the new top-level codecs
define_dft_partition SUB_GLUE -include [list GLUE_cell_list]

# define top-level partitions that group subpartition codecs or cores
define_dft_partition PARTITION1 -include [list C_CORE1 SUB_GLUE]
define_dft_partition PARTITION2 -include [list C_CORE2 C_CORE3]

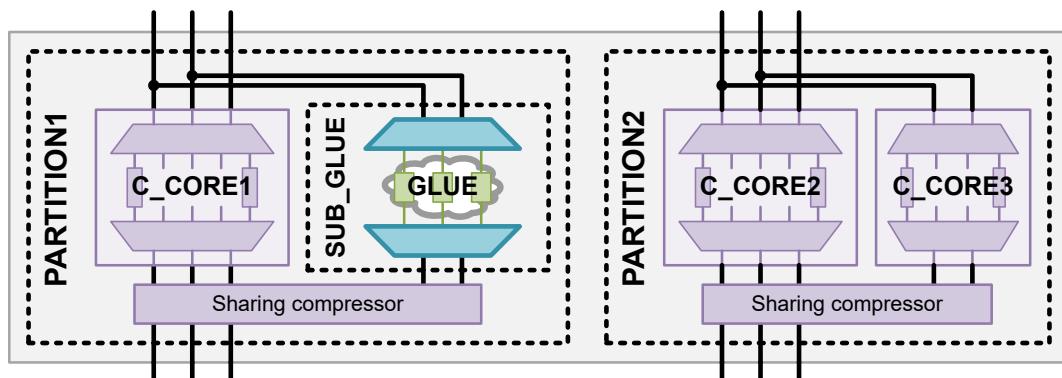
# apply top-level codec characteristics to subpartitions
current_dft_partition SUB_GLUE
set_scan_configuration -chain_count 2
set_scan_compression_configuration -chain_count 3 -inputs 2 -outputs 2

# apply codec I/O sharing characteristics
current_dft_partition PARTITION1
set_scan_configuration -chain_count 3
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3

current_dft_partition PARTITION2
set_scan_configuration -chain_count 3
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3
```

[Figure 350](#) shows the Hybrid integration results for these commands.

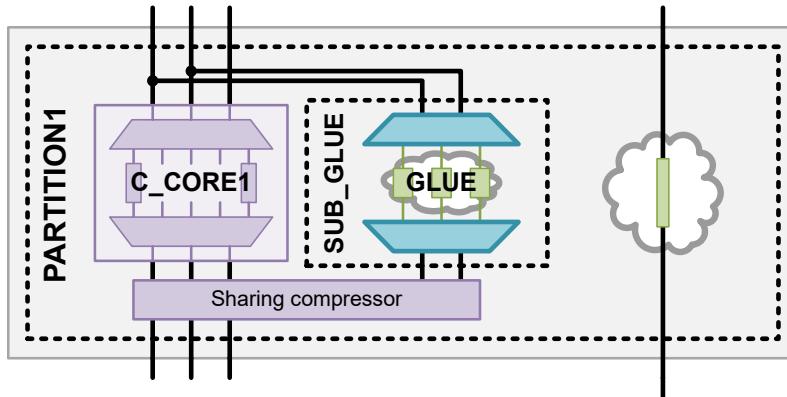
Figure 350 Hybrid Flow With Sharing Groups and a Shared Top-Level Codec



You can define one or more shared codec subpartitions within any top-level partition, with or without compressed scan cores. If some scan cells exist in a top-level partition but

outside a subpartition, the tool places them into an external chain that is not compressed inside that partition. See [Figure 351](#).

Figure 351 Partition-Level External Chains in Hybrid Flow



Additional requirements apply to the scan configuration when codec I/O sharing is used in the Hybrid flow with shared top-level codecs. For more information, see [Shared Codec I/O Limitations on page 815](#).

Using Dedicated Codecs for Top-Level Logic

You can define top-level partitions to specify the top-level scan logic to be scan-compressed with a dedicated codec.

In this flow, create and configure sharing groups as follows:

1. Define a top-level partition containing the logic to be scan-compressed with a dedicated top-level codec. You can also omit a partition definition and use the default partition.
2. Define additional top-level partitions containing the compressed scan cores and/or subpartitions that are to share their codec I/O connections.
3. Configure the top-level logic codec characteristics within its top-level partition.
4. Configure the shared codec I/O characteristics within each top-level partition where sharing occurs.

The following example defines two top-level DFT partitions, one of which contains two shared-I/O cores, and one of which contains a dedicated top-level codec:

```
# globally enable scan compression and Hybrid integration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -hybrid true

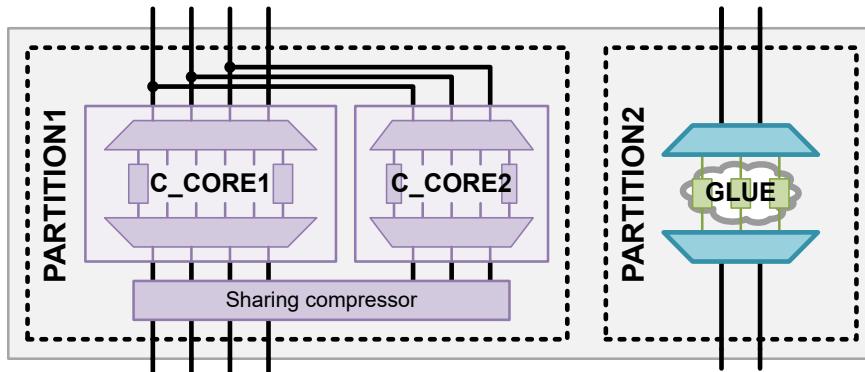
# define top-level partitions that group cores or dedicated codecs
define_dft_partition PARTITION1 -include [list C_CORE1 C_CORE2]
define_dft_partition PARTITION2 -include [list GLUE_cell_list]
```

```
# apply codec I/O sharing characteristics for cores
current_dft_partition PARTITION1
set_scan_configuration -chain_count 3
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3

# apply top-level dedicated codec characteristics
current_dft_partition PARTITION2
set_scan_configuration -chain_count 2
set_scan_compression_configuration -chain_count 3 -inputs 2 -outputs 2
```

[Figure 352](#) shows the Hybrid integration results for these commands.

Figure 352 Hybrid Flow With Sharing Groups and a Dedicated Top-Level Codec



Defining Sharing Groups for Codecs in Partitioned Cores

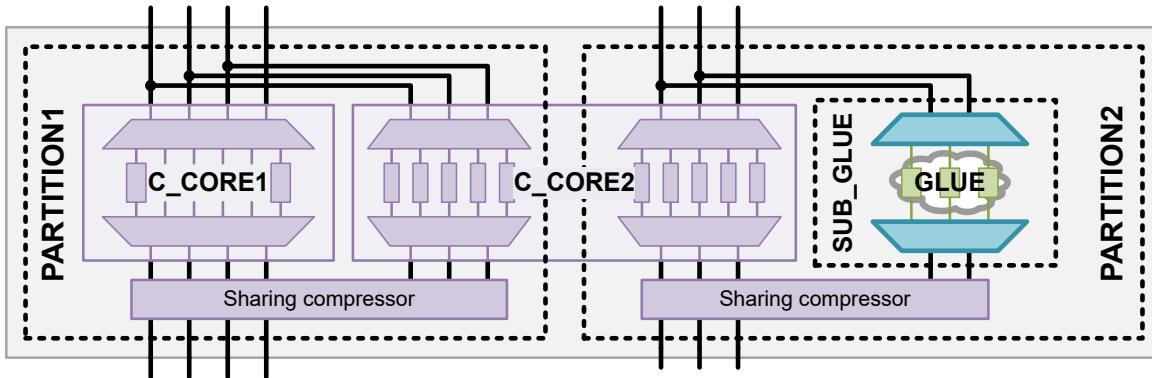
In the HASS and Hybrid flows, you can define sharing groups that reference individual codecs inside partitioned cores. To do this, reference the codec's decompressor using the `cell_name:decompressor_name` syntax in sharing group partition definitions. For example,

```
# define subpartitions that define the new top-level codecs
define_dft_partition SUB_GLUE -include [list GLUE_cell_list]

# define top-level partitions that group cores or dedicated codecs
define_dft_partition PARTITION1 -include [list \
    C_CORE1 \
    C_CORE2:core2_P1_U_decompressor_ScanCompression_mode]
define_dft_partition PARTITION2 -include [list \
    C_CORE2:core2_P2_U_decompressor_ScanCompression_mode \
    SUB_GLUE]
```

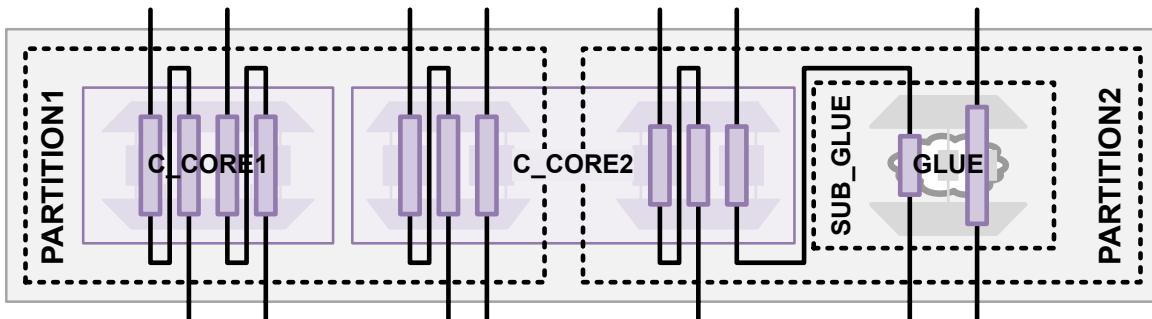
[Figure 353](#) shows the Hybrid integration results for these commands.

Figure 353 Hybrid Flow With Individual Codec References in Sharing Groups



When you include a core-level decompressor reference in a DFT partition definition, if core scan segments in other top-level test modes share the same core scan-in pins as the decompressor, they are also included in that DFT partition for those modes. [Figure 354](#) shows the standard scan results for the previous example, assuming that the standard scan and compressed scan modes of the cores share the same scan-in and scan-out pins.

Figure 354 Standard Scan Mode for Individual Codec References in Sharing Groups



If core scan segments in other modes do not share any referenced decompressor scan-in pins, they are included in the default partition.

You can obtain the decompressor names using one of the following methods:

- Use the `list_test_models -compressors` command at the top level before DFT insertion.
- Look in the `CompressorStructures` section of a core-level ASCII CTL model file.
- Look in the `CompressorStructures` section of a core-level STIL protocol file that is generated for the scan compression mode.

Note the following requirements and behaviors for referencing core codecs in DFT partition definitions:

- Codecs are referenced by decompressor name only.
- You can mix decompressor references with other object types in the same partition definition, subject to the usual restriction that the same underlying scan logic cannot exist in multiple partitions.
- Any unreferenced codecs are placed in the default partition.
- Decompressor references are supported only in shared codec I/O flows.

Defining Sharing Groups in the Top-Down Flat Flow

This flow is a simple extension of the normal top-down flat codec I/O sharing flow, except multiple top-level partitions are defined.

Create and configure sharing groups as follows:

1. Define any subpartitions containing the logic to be scan-compressed with a codec.
2. Define top-level partitions containing the subpartitions that are to share their codec I/O connections.
3. Configure the codec characteristics within each subpartition.
4. Configure the shared codec I/O characteristics within each top-level partition.

The following example defines two codec groups, each with its own shared I/O connections, in a top-down flat flow:

```
# globally enable scan compression
set_dft_configuration -scan_compression enable

# define subpartitions that define codecs
define_dft_partition SUB_P1 -include {BLK1}
define_dft_partition SUB_P2 -include {BLK2}
define_dft_partition SUB_P3 -include {BLK3}
define_dft_partition SUB_P4 -include {BLK4}

# define top-level partition that groups subpartition codecs
define_dft_partition PARTITION1 -include {SUB_P1 SUB_P2} ;# subpartitions
define_dft_partition PARTITION2 -include {SUB_P3 SUB_P4} ;# subpartitions

# apply subpartition codec characteristics
current_dft_partition SUB_P1
set_scan_configuration -chain_count 3
set_scan_compression_configuration -chain_count 5 -inputs 3 -outputs 3

current_dft_partition SUB_P2
set_scan_configuration -chain_count 2
```

```

set_scan_compression_configuration -chain_count 4 -inputs 2 -outputs 2
current_dft_partition SUB_P3
set_scan_configuration -chain_count 3
set_scan_compression_configuration -chain_count 5 -inputs 3 -outputs 3

current_dft_partition SUB_P4
set_scan_configuration -chain_count 2
set_scan_compression_configuration -chain_count 4 -inputs 2 -outputs 2

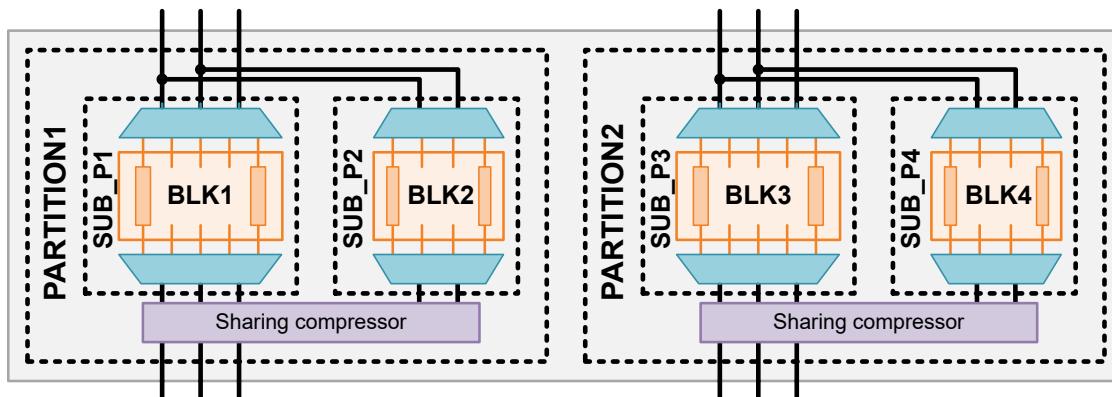
# apply top-level codec I/O sharing characteristics
current_dft_partition PARTITION1
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3

current_dft_partition PARTITION2
set_scan_compression_configuration -shared_inputs 3 -shared_outputs 3

```

Figure 355 shows the top-down flat DFT insertion results for these commands.

Figure 355 Top-Down Flat Flow Codec I/O Sharing With Multiple Top-Level Partitions

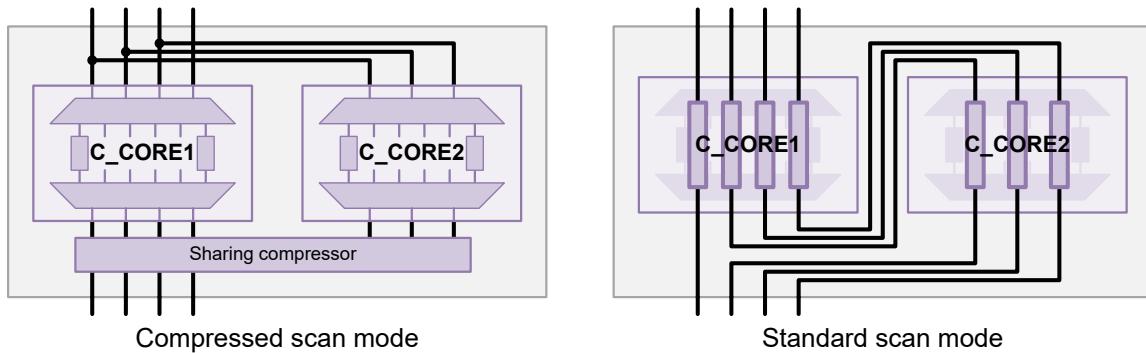


Codec I/O Sharing and Standard Scan Chains

When the top-level codec I/O connections are shared in compressed scan mode, the standard scan mode is also affected.

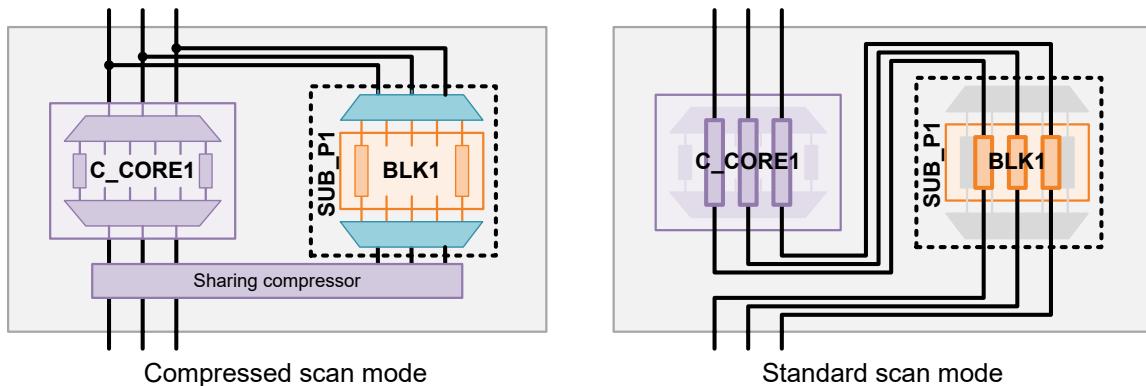
In the HASS integration flow, due to the reduced number of available scan I/O pins, the standard scan chains inside the compressed cores can no longer be promoted to dedicated top-level connections. To remedy this, standard scan chains in compressed scan cores become scan segments that can be concatenated, if needed, by top-level integration. **Figure 356** shows the compressed scan and standard scan chains for a design in the HASS integration flow.

Figure 356 Standard Scan Chains in the HASS Flow With Codec I/O Sharing



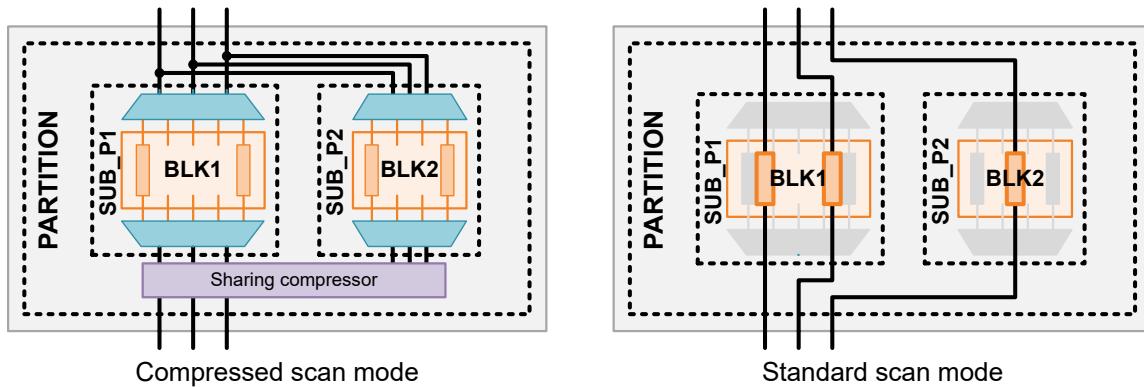
In the Hybrid flow with codec I/O sharing, core-level scan segments can be mixed with top-level scan cells to achieve optimal balancing. [Figure 357](#) shows the compressed scan and standard scan chains for a design in the Hybrid integration flow.

Figure 357 Standard Scan Chains in the Hybrid Flow With Codec I/O Sharing



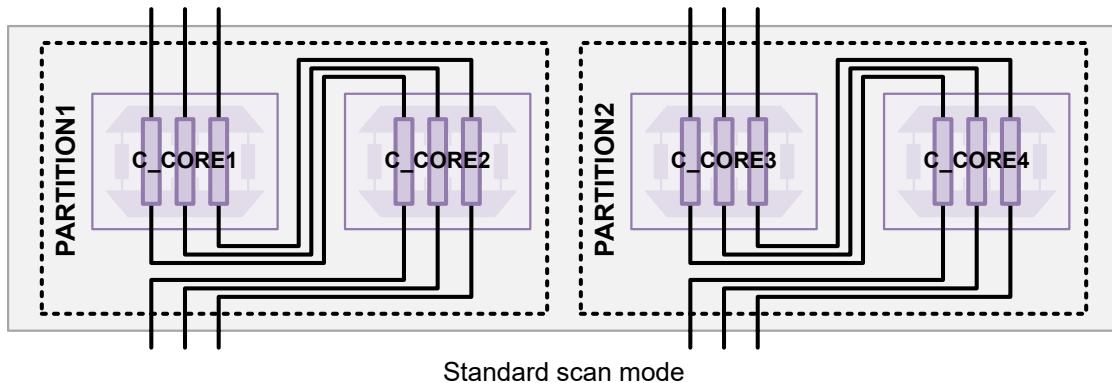
In top-down flat flows, the subpartition standard scan chains are promoted to top-level scan chains with no concatenation or rebalancing within the enclosing top-level partitions. You must apply the `set_scan_configuration -chain_count` command to the subpartitions to manage the standard scan chain counts. [Figure 358](#) shows the compressed scan and standard scan chains for a design in the top-down flat flow, where SUB_P1 has a specified chain count of 2 and SUB_P2 has a specified chain count of 1.

Figure 358 Standard Scan Chains in the Top-Down Flat Flow With Codec I/O Sharing



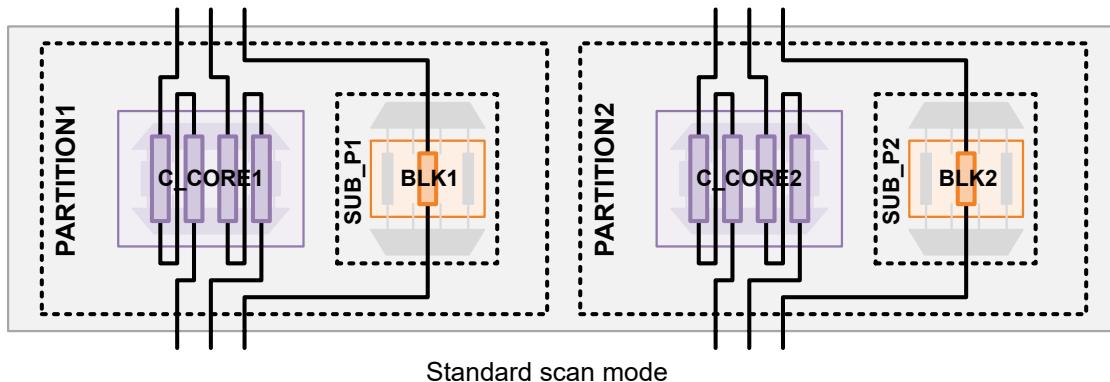
In the HASS flow with sharing groups, partition boundaries prevent core-level scan segment concatenation across partitions, but scan segments can still be concatenated and balanced within each partition. Apply the `set_scan_configuration -chain_count` command to each partition to specify the number of standard scan chains for that partition. [Figure 359](#) shows the standard scan chains for a design in the HASS integration flow with sharing groups.

Figure 359 Standard Scan Chains in the HASS Flow With Codec I/O Sharing Groups



In the Hybrid flow with sharing groups, core-level scan segments are concatenated and balanced within top-level partitions (as in the HASS flow with sharing groups), but subpartition scan chains are promoted to top-level scan chains (as in the top-down-flat flows). For subpartitions, apply the `set_scan_configuration -chain_count` command to specify the number of standard scan chains to create. For top-level partitions, apply the `set_scan_configuration -chain_count` command to specify the total scan chain count for all cores in the partition, excluding any subpartitions. [Figure 360](#) shows the standard scan chains for a design in the Hybrid integration flow with sharing groups.

Figure 360 Standard Scan Chains in the Hybrid Flow With Codec I/O Sharing Groups



For pipelined cores, if a leading-edge tail pipeline register in one core is concatenated to a trailing-edge head pipeline register in another core, the scan architect inserts a lockup latch and retiming flip-flop between the cores for correct scan shift operation (independent of the `-add_test_retimings_flops` option setting of the `set_scan_configuration` command). For details, see [SolvNet article 1656177, “Why Does insert_dft Add Extra Retiming Registers in a Shared Codec I/O Flow?”](#)

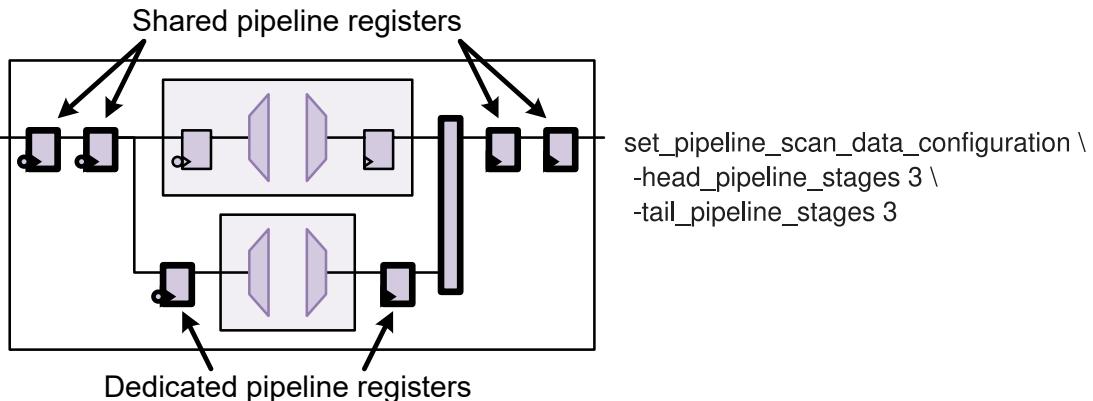
See Also

- [Chapter 18, Hierarchical Adaptive Scan Synthesis](#) for more information about specifying standard scan chain counts in the HASS and Hybrid flows
- [Top-Down Flat Compressed Scan Flow on page 663](#) for more information about applying chain count and scan I/O signal specifications to DFT partitions in scan compression flows

Codec I/O Sharing and Pipelined Scan Data

When codec I/O sharing is used with the pipelined scan data feature, the tool adds pipeline stages as needed to meet the total top-level pipeline depth, as shown in [Figure 361](#). Pipeline registers added along the shared scan data path are called *shared pipeline registers*, and pipeline registers added along the scan data path to a single shared codec are called *dedicated pipeline registers*.

Figure 361 Pipelined Scan Data in the Shared Codec I/O Flow

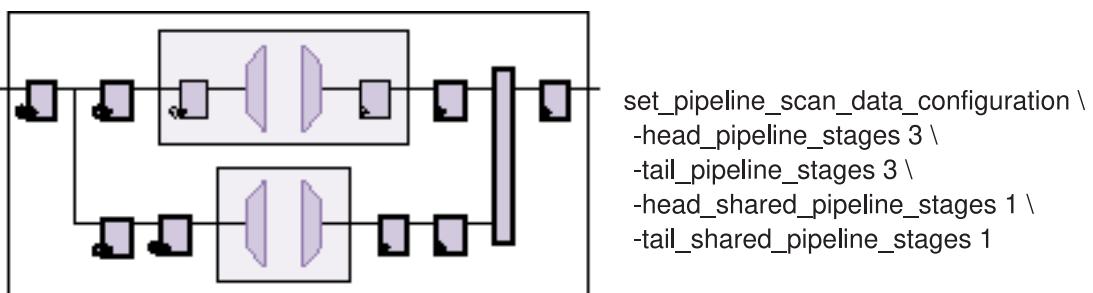


By default, the tool uses as many shared pipeline registers as possible to minimize cell count, and it uses dedicated pipeline registers only to balance cores of differing depths. In some cases, such as to adjust layout characteristics, you might want to change the allocation of shared and dedicated pipeline registers. To do this, you can use the **-head_shared_pipeline_stages** and **-tail_shared_pipeline_stages** options of the **set_pipeline_scan_data_configuration** command:

```
set_pipeline_scan_data_configuration
  -head_pipeline_stages total_depth
  -tail_pipeline_stages total_depth
  -head_shared_pipeline_stages shared_depth
  -tail_shared_pipeline_stages shared_depth
  ...
```

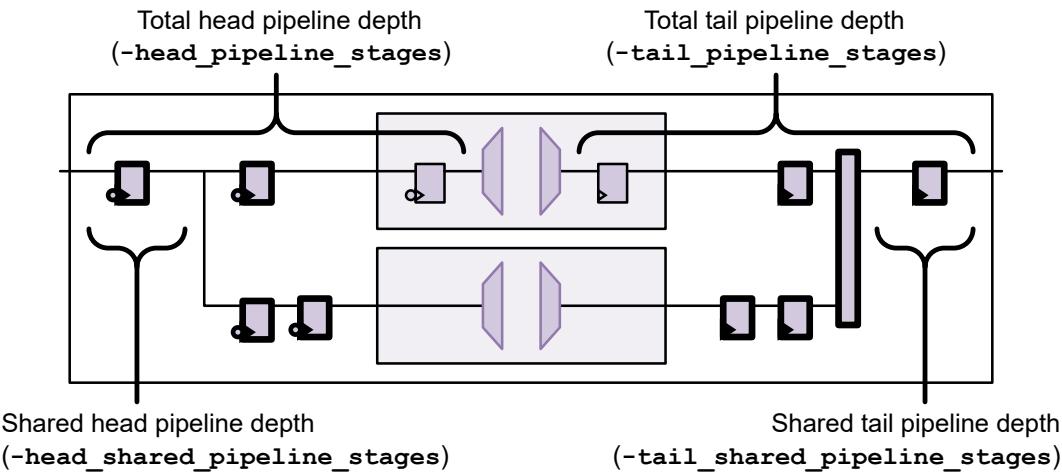
The tool uses the specified number of shared pipeline registers along the shared scan path, then it uses dedicated pipeline registers as needed for the remaining stages to meet the total pipeline depth target. [Figure 362](#) shows the previous example with a single shared head and tail pipeline stage.

Figure 362 Using a Reduced Number of Shared Pipeline Register Stages



[Figure 363](#) shows the relationship between the total and shared pipeline depth specification options of the `set_pipeline_scan_data_configuration` command.

Figure 363 Relationship Between the Total and Shared Pipeline Depth Specification Options



If your design contains dedicated pipeline registers, note the following:

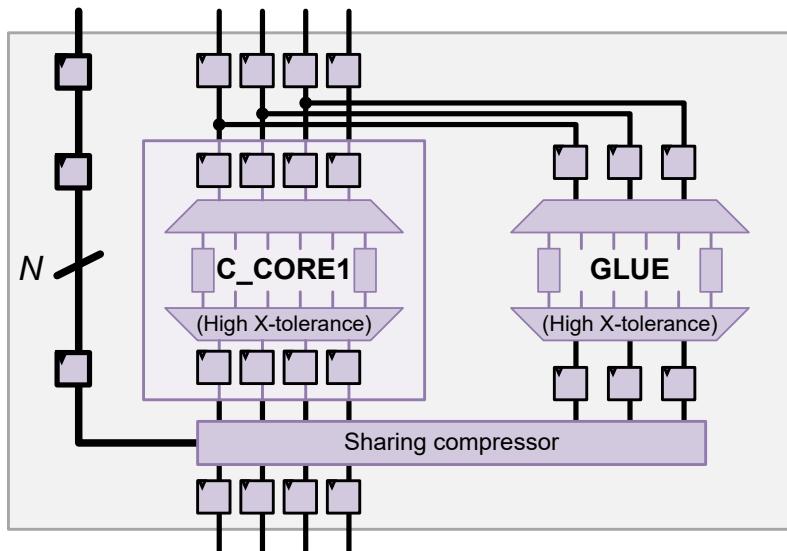
- If you perform a post-DFT incremental compile with the `compile_ultra` command, redundant register removal might collapse parallel dedicated pipeline registers together. To disable redundant register removal on pipeline register cells, issue the following command before performing the post-DFT incremental compile:

```
dc_shell> set_size_only [get_cells -hier {SNPS_Pipe*}] true
dc_shell> compile_ultra -scan -incremental
```

- Dedicated pipeline registers are inserted on a per-test-mode basis, which might result in parallel pipeline registers along the same scan path with duplicate functionality.

If you are using high X-tolerance with codec I/O sharing and pipelined scan data, the tool inserts pipeline stages on the block-select signals to match the total head and tail pipeline latency leading to (but not following) the sharing compressor. [Figure 364](#) shows pipeline stages added to the block-select signals in a Hybrid flow.

Figure 364 I/O Sharing With High X-Tolerance Codecs and Pipelined Scan Data



See Also

- [Pipelined Scan Data on page 755](#) for more information about the pipelined scan data feature
- [Hierarchical Flows With Pipelined Scan Data on page 770](#) for more information about the requirements of the pipelined scan data feature in the HASS and Hybrid flows

Integrating Cores That Contain Shared Codec I/O Connections

When you integrate cores, you can include cores that contain shared codec I/O connections along with compressed scan cores or standard scan cores.

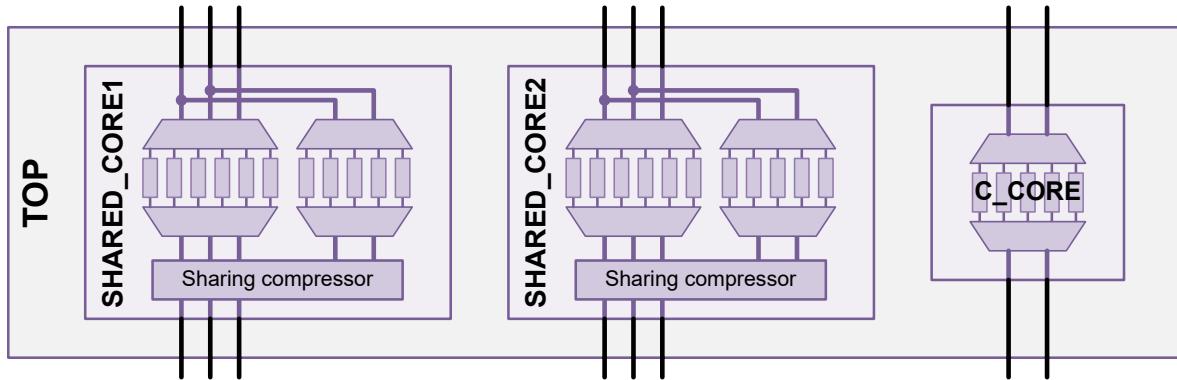
The following topics describe how cores are integrated with shared codec I/O connections:

- [Integrating Shared I/O Cores](#)
- [Integrating Identical High X-Tolerance Shared I/O Cores](#)
- [Integrating Shared I/O Cores Using Shared Codec Controls](#)
- [Integrating Shared I/O Cores That Contain Shared Codec Controls](#)

Integrating Shared I/O Cores

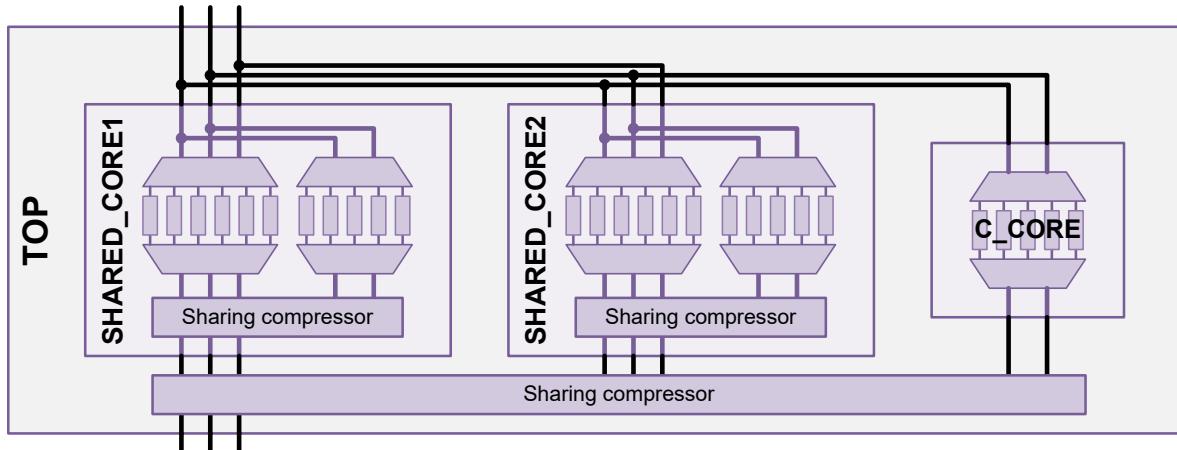
If you do not enable codec I/O sharing at the integration level, the tool promotes the scan I/O connections of each core to top-level scan connections during integration. See [Figure 365](#).

Figure 365 HASS Integration of Shared Codec I/O Cores



If you enable codec I/O sharing at the integration level with the `-shared_inputs` option, the tool performs nested codec I/O sharing. The scan I/O connections of each core are shared at the integration level. See [Figure 366](#).

Figure 366 Shared-I/O HASS Integration of Shared Codec I/O Cores



The values provided to the `-shared_inputs` and `-shared_outputs` options must meet requirements that are similar to other shared codec I/O flows. The value specified for the `-shared_inputs` option must be at least as wide as the widest core scan input. The value specified for the `-shared_outputs` option must be equal to the widest core scan output.

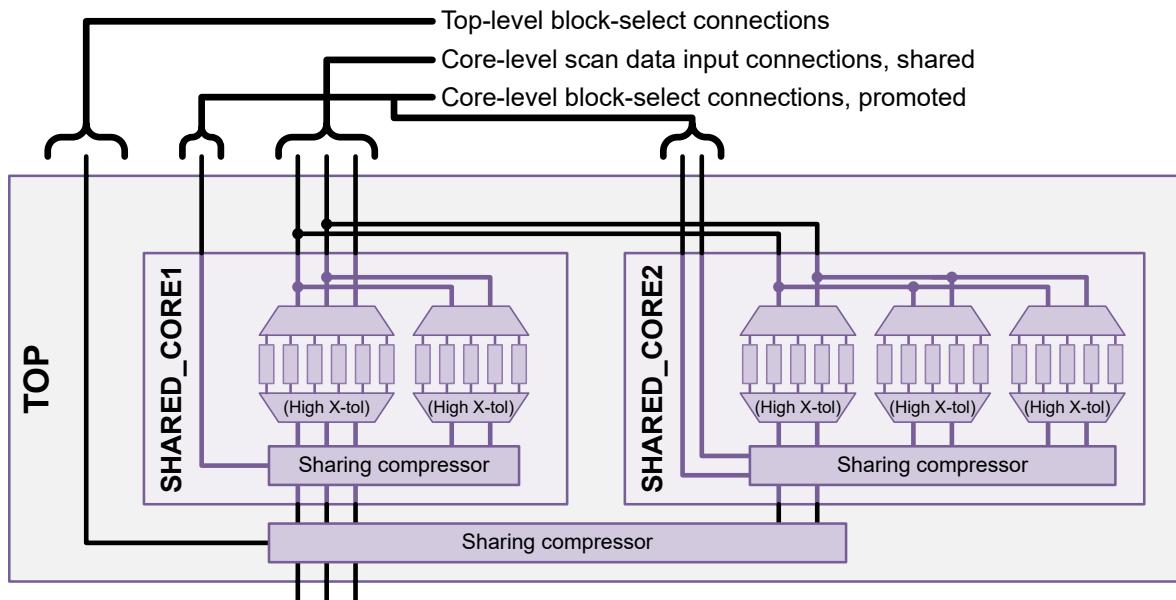
If some cores have high X-tolerance capability, you must account for additional block-select pins when specifying a value for the `-shared_inputs` option. The value must be at least as large as the sum of the following values:

- The width of the widest core-level scan data input across all cores
- The sum of the block-select signals used across all shared codec I/O cores
- The number of top-level block-select signals needed, which is \log_2 of the number of high X-tolerance shared codec I/O cores, rounded up to the next integer value

[Figure 367](#) shows the scan data inputs needed for two shared codec I/O cores.

`SHARED_CORE1` has a wider set of scan data inputs. Because there are two shared codec I/O cores, an additional top-level block-select signal is also required. In this example, a minimum value of 7 must be specified with the `-shared_inputs` option.

Figure 367 Shared-I/O HASS Integration of High X-Tolerance Cores



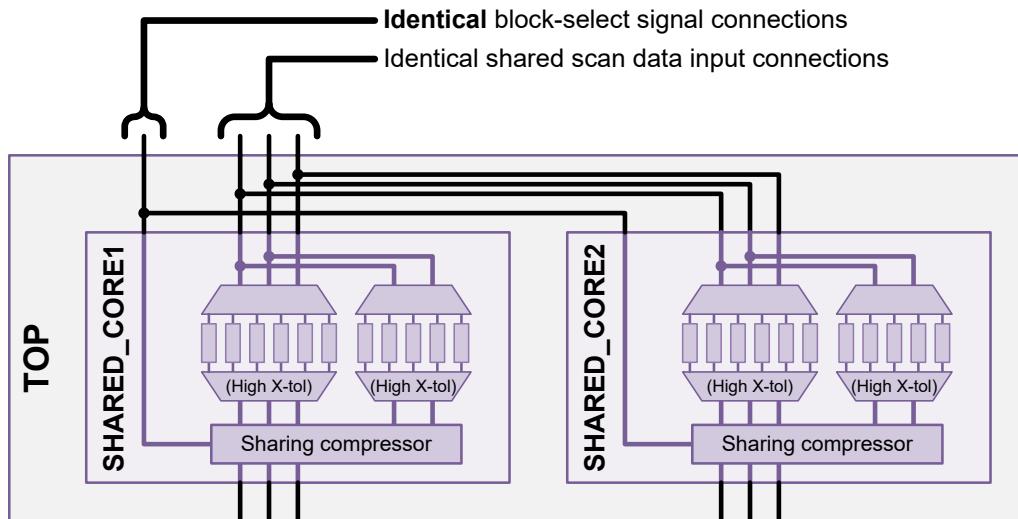
See Also

- [Adding High X-Tolerance Block-Select Pins on page 780](#) for more information about shared codec I/O block-select signals

Integrating Identical High X-Tolerance Shared I/O Cores

If you integrate identical high X-tolerance shared I/O cores using dedicated (unshared) outputs, you can use identical block-select signal connections for the cores, as shown in [Figure 368](#).

Figure 368 Integrating Identical Shared-I/O High X-Tolerance Cores



To enable this feature, use the following command:

```
dc_shell> set_scan_compression_configuration -shared_block_select true
```

This option can be enabled only when the following conditions are met:

- Shared codec I/O is enabled with the `-shared_inputs` option.
- The cores contain shared-I/O codecs with high X-tolerance, such that the core has one or more block-select signals.
- All cores in the current design or partition are identical instances of this core, specified with the `-identical_cores` option.
- Dedicated (unshared) outputs are used for the cores by specifying the fully unshared value with the `-shared_outputs` option. (Using shared block-select signals with shared core outputs would degrade the high X-tolerance functionality.)

The default is to use separate block-select signals for each identical core.

See Also

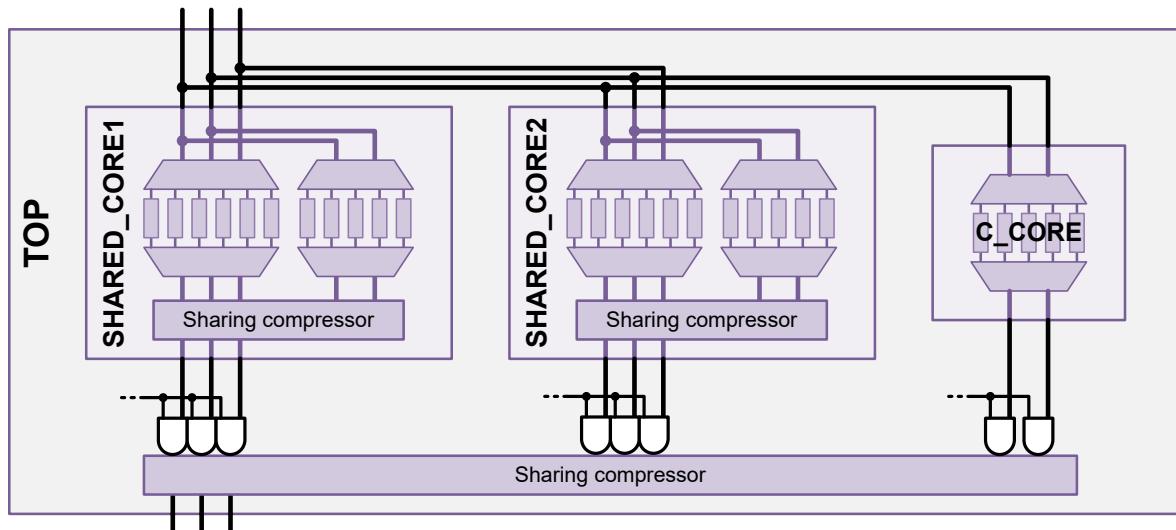
- [Adding High X-Tolerance Block-Select Pins on page 780](#) for more information on core-level block-select signals

Integrating Shared I/O Cores Using Shared Codec Controls

When you integrate cores with shared codec controls enabled, the tool adds AND-gating logic at the codec inputs of output sharing compressors, as described in [Codec I/O Sharing With Shared Codec Controls on page 793](#).

If a core is a shared I/O core (that is, it contains shared I/O codecs), the gating logic enables or disables all shared codecs that drive the gating logic. See [Figure 369](#).

Figure 369 Integrating Shared I/O Cores Using Shared Codec Controls



When the `preview_dft` command reports the codec enable pins, it uses the name of the shared (merged) codec stored in the core CTL model. For this example, the report is as follows:

```
dc_shell> preview_dft
...
*****
** Compressors Control Report ****
U_sharing_compressor/codec_enable_0 controls
  C_CORE:C_CORE_U_decompressor_ScanCompression_mode
U_sharing_compressor/codec_enable_1 controls
  SHARED_CORE1:SHARED_CORE_P1_U_decompressor_ScanCompression_mode
U_sharing_compressor/codec_enable_2 controls
  SHARED_CORE2:SHARED_CORE_P1_U_decompressor_ScanCompression_mode
*****
```

See Also

- [Codec I/O Sharing With Shared Codec Controls on page 793](#) for more information about using shared codec controls to disable codecs

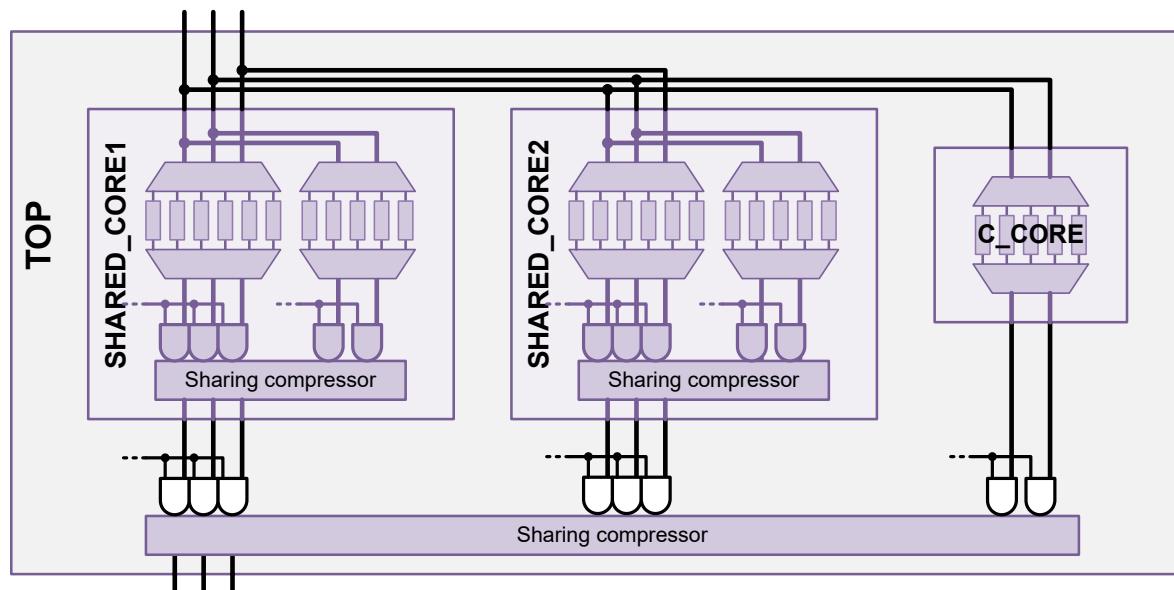
Integrating Shared I/O Cores That Contain Shared Codec Controls

When you integrate shared I/O cores that contain shared codec controls,

- If the core uses IEEE 1500 test-mode control, the core-level codec controls become available through IEEE 1500 test-mode control at the top level.
- If the core does not use IEEE 1500 test-mode control, the tool has no visibility of the core-level codec controls, and you cannot disable individual core-level codecs from the top level.

[Figure 370](#) shows an example where two cores with IEEE 1500 test-mode control and codec controls are integrated in a top level with codec controls enabled.

Figure 370 Integrating Shared I/O Cores That Contain Shared Codec Controls



When cores use IEEE 1500 test-mode control, the `preview_dft` command reports core-level shared codec controls as “can be controlled.” For this example, the report is as follows:

```
dc_shell> preview_dft
...
*****
* Compressors Control Report *
*****
U_sharing_compressor/codec_enable_0 controls
  C_CORE:C_CORE_U_decompressor_ScanCompression_mode
U_sharing_compressor/codec_enable_1 controls
  SHARED_CORE1:SHARED_CORE_P1_U_decompressor_ScanCompression_mode
U_sharing_compressor/codec_enable_2 controls
  SHARED_CORE2:SHARED_CORE_P1_U_decompressor_ScanCompression_mode
SHARED_CORE2:SHARED_CORE2_SHARED_CORE_P2_U_decompressor_Sc
```

```

anCompression_mode can be controlled.
SHARED_CORE2:SHARED_CORE2_SHARED_CORE_SHARED_CORE_P1_U_decompressor_Sc
    anCompression_mode can be controlled.
SHARED_CORE1:SHARED_CORE1_SHARED_CORE_SHARED_CORE_P2_U_decompressor_Sc
    anCompression_mode can be controlled.
SHARED_CORE1:SHARED_CORE1_SHARED_CORE_SHARED_CORE_P1_U_decompressor_Sc
    anCompression_mode can be controlled.
*****

```

You do not need to enable shared codec controls at the top level to integrate cores that contain shared codec controls.

See Also

- [Codec I/O Sharing With Shared Codec Controls on page 793](#) for more information about using shared codec controls to disable codecs
- [Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces on page 377](#) for more information about test-mode control through an IEEE 1500 interface

Shared Codec I/O Limitations

Note following requirements and limitations of the shared codec I/O feature:

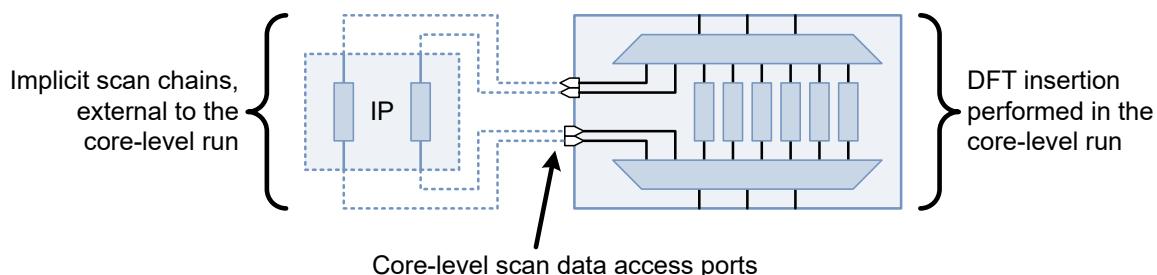
- All codecs must have the same X-tolerance type.
- You can only share outputs when inputs are also shared. If you specify dedicated inputs, all sharing is disabled.
- To use dedicated (fully unshared) outputs, all inputs must be fully shared and all cores must be identical in a single group. See [Specifying Shared Codec Inputs With Dedicated Codec Outputs on page 792](#).
- In Hybrid mode, the compressed scan chain characteristics of a shared top-level codec must be specified with the `-chain_count` or `-max_length` options of the `set_scan_compression_configuration` command. They cannot be automatically derived using the normal chain count heuristics.
- When you use partitions,
 - Codec I/O can be shared within each top-level partition, but not across top-level partition boundaries.
 - Standard scan chains are balanced within each top-level partition, but not across top-level partition boundaries.
 - In the `-include` list of the `define_dft_design` command, you can reference cores to be shared only by cell instance; you cannot reference them by design name.

- Subpartition definitions (top-level partition definitions that reference other DFT partitions by name) are only supported when using codec I/O sharing, as a way to define sharing groups. They are not supported for other flows.
- In flows that use subpartitions, which are the top-down flat flow and the Hybrid flow with shared top-level codecs, the default partition cannot contain subpartitions because there is no explicit `define_dft_partition` command that allows you to reference a subpartition. You can only include subpartitions in user-defined top-level partitions.
- If you define multiple compression modes and you have DFTMAX cores to be integrated with shared codec I/O, you must schedule the cores using the `-target` option of the `define_test_mode` command.
- When implementing shared codec controls,
 - At least one codec must remain active for each output sharing compressor.
 - In the Hybrid integration mode, if the top-level decompressor name differs between the `preview_dft` and `list_test_models -compressors` commands, you must use the decompressor name from the `preview_dft` command.
- When integrating cores that contain shared codec controls, you must use IEEE 1500 test-mode control to make core-level codec controls available at the top level.

Implicit Scan Chains

Implicit scan chains provide a mechanism to specify one or more “implicit” scan segments that exist outside the current design but should be included in compressed scan insertion. This is useful when an IP block to be scan-compressed exists at the chip level but compressed scan is inserted at the core level, as shown in [Figure 371](#).

Figure 371 Implicit Scan Chains Defined in a Core-Level Run



Each implicit scan chain segment is defined in the core level run, characterized by the following information:

- Chain name
- Chain length
- Scan clock
- Core-level scan data access ports

When implicit scan chain segments are defined, DFTMAX compression incorporates them into compressed scan insertion by connecting to the core-level scan data access ports. It uses the clock and length information to construct scan chains that are optimally balanced while respecting the chain count and clock-mixing configuration settings applied to the current design.

Just as with any other user-defined scan segment, implicit scan chains are incorporated into both the standard scan and compressed scan modes. Reconfiguration MUXs are added as needed.

When implicit scan chains are used, the tool produces a partial test protocol. This protocol is not complete and contains only a partial ScanChain definition for the implicit scan chains. The protocol cannot be used in the TestMAX ATPG tool directly.

Defining Implicit Scan Chains

To define implicit scan chain segments, use the following two commands:

- `set_scan_group`
- `set_scan_path`

Use the `set_scan_group` command to define a scan group for each implicit scan chain:

```
set_scan_group group_name
    -serial_routed true -segment_length length
    -access {ScanDataIn core_output_port ScanDataOut core_input_port}
    -clock clock_name
    [-edge rising | falling]
```

where

- The `group_name` argument is a unique user-defined scan group name.
- The `chain_length` argument is the chain length of the implicit scan chain.
- The `clock_name` argument is the scan clock which clocks the implicit scan chain. It must be defined as a scan clock in the core-level run using the `set_dft_signal` command.

- The `core_output_port` argument is the core-level output port that externally connects to the scan input of the implicit chain.
- The `core_input_port` argument is the core-level input port that externally connects to the scan output of the implicit chain.

By default, an implicit scan chain is defined as a rising-edge scan segment. To define it as a falling-edge scan segment, add the `-edge falling` option to the `set_scan_group` command. You cannot define an implicit scan chain that represents a mix of rising-edge and falling-edge cells.

Use the `set_scan_path` command to specify how each scan group is to be incorporated into scan stitching:

```
set_scan_path chain_name -test_mode all
    -ordered_elements {group_name ...}
    | -include_elements {group_name ...}
    [-complete true]
```

where

- The `group_name` argument is a scan group name, previously defined with the `set_scan_group` command.
- The `chain_name` argument is a unique user-defined scan chain name.

By default, implicit scan chains can be mixed with core-level scan cells if DFT requirements such as scan clock mixing and chain lengths are met. To force implicit scan chains to be standalone compressed scan chains, add the `-complete true` option to the `set_scan_path` command. This can be useful when you have an implicit scan chain with a mix of rising-edge and falling-edge cells that cannot be described by the `set_scan_group -edge` command.

After you have defined the implicit scan chains, use the `preview_dft` command to report how they will be integrated into the core-level scan structures. Implicit scan chains are represented as scan segments. The following partial preview report shows a core-level scan chain that contains an implicit scan chain and two core-level scan cells:

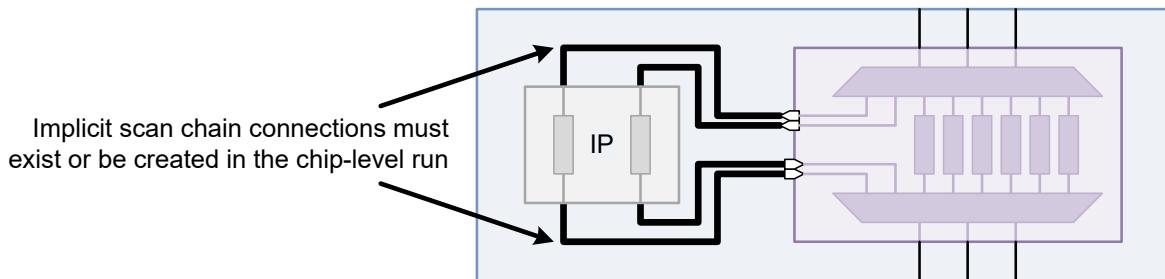
```
dc_shell> preview_dft -show {segments cells}
...
Scan chain 'c1' contains 5 cells
  Active in modes: ScanCompression_mode :
    sub1/Z_reg[0]
    sub1/Z_reg[1] (m)
    IMPLICIT1 (s) (m)
...

```

For details on previewing scan segments, see [Previewing Additional Scan Chain Information on page 604](#).

At the chip level, the implicit scan chain segments must be connected to the core-level scan access ports. These connections, highlighted in [Figure 372](#), can exist in the chip-level RTL or they can be created using netlist editing commands. the tool does not make these connections.

Figure 372 Implicit Scan Chain Connections to Core in a Chip-Level Run



Implicit Scan Chain Script Example

The following example shows the use of implicit scan chains in a core-level scan compression insertion run.

```
current_design CORE

# create scan port that connects to the implicit scan chain's scan input
# (which is an output from CORE)
create_port -direction out dout

# create scan port that connects to the implicit scan chain's scan output
# (which is an input from CORE)
create_port -direction in din

# define all test signals; test clocks must be defined before being
# referenced by 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 the external scan
# chain and din is driven by the scan output of the external 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 -ordered_elements IMPLICIT1 -test_mode all \
-complete true

# enable DFTMAX compression insertion
set_dft_configuration -scan_compression enable
```

```

# configure DFTMAX compression
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-DFT DRC
dft_drc -verbose

# preview DFT insertion
preview_dft -show all

# perform DFT insertion
insert_dft

# post-DFT DRC is not supported
change_names -rules verilog -hierarchy

# write out the compression mode protocol for TestMAX ATPG
write_test_protocol -test_mode ScanCompression_mode \
    -output stil/scan_compression.stil -names verilog

# write out the pure scan protocol for TestMAX ATPG
write_test_protocol -test_mode Internal_scan \
    -output stil/internal_scan.stil -names verilog

# write out the inserted design in Verilog and Synopsys ddc format
write -format verilog -hierarchy -output vg/design_with_implicit.v
write -format ddc -hierarchy -output db/design_with_implicit.ddc

```

Protocol Example

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 out, and the scan clock in the test protocol.

```

ScanStructures {
    ScanChain "c1" {
        ScanLength 67;
        ScanOut "din";
        ScanMasterClock "clk_st";
    }
}

```

In the same protocol, an example of a normal (nonimplicit) scan chain definition:

```

ScanChain "2" {
    ScanLength 10;
    ScanEnable "test_se";
    ScanMasterClock "clk_st";
}

```

Limitations

Note the following limitations when using implicit scan chains:

- The `report_scan_path` command does not show the presence of implicit scan chain segments in the core-level scan chains; use the `preview_dft` command before DFT insertion instead.
- At the chip level, the implicit scan chain segments must be connected to the core-level scan access ports; the tool does not make these connections.
- Implicit scan chains do not reliably mix with scan cells of opposite edge polarity when edge-mixing is enabled with the `set_scan_configuration -clock_mixing` command.
- When implicit scan chains are used, the TestMAX DFT tool produces a partial test protocol. This protocol is not complete and contains only a partial ScanChain definition for the implicit scan chains. The protocol cannot be used in the TestMAX ATPG tool directly.
- Post-DFT DRC is not supported when implicit scan chains are used. Use the TestMAX ATPG tool to perform DRC checking.

21

DFTMAX Compression With Serializer

DFTMAX compression with serializer can be used to improve the ATPG quality of results (QoR) for designs or blocks with a limited number of top-level ports. This QoR improvement is accomplished by employing a serial connection between the codec and the top-level ports.

This chapter includes the following topics:

- [Overview](#)
- [Architecture](#)
- [Serializer Operation](#)
- [Higher Shift Speed and Update Stage](#)
- [Scan-Enable Signal Requirements for Serializer Operation](#)
- [Timing Paths](#)
- [Scan Clocks](#)
- [User Interface](#)
- [Configuring Serialized Compressed Scan](#)
- [Deserializer/Serializer Register Size](#)
- [Serializer Implementation Flow](#)
- [Serialized Compressed Scan Core Creation](#)
- [Top-Down Flat Flow](#)
- [Top-Down Partition Flow](#)
- [HASS Flow](#)
- [Hybrid Flow](#)
- [Serializer IP Insertion](#)
- [Wide Duty Cycle Support for Serializer](#)
- [Serializer in Conjunction With On-Chip Clocking Controllers](#)

- [Using Integrated Clock-Gating Cells in the Serializer Clock Controller](#)
- [User-Defined Pipelined Scan Data](#)
- [Running TestMAX ATPG on Serializer Designs](#)
- [DFTMAX Compression With Serializer Limitations](#)
- [Out-of-Scope Serializer Functionality](#)
- [DFTMAX Compression Error Messages](#)

Overview

The scan architecture that DFTMAX compression creates is called *compressed scan*. By default, the connection from the input and output of the compressor/decompressor (codec) to the top-level ports or pins created by DFTMAX scan compression is combinational. To improve the ATPG quality of results (QoR) for designs or blocks with a limited number of top-level ports, DFTMAX compression also supports an optional serial connection between the codec and the top-level ports.

This chapter uses the term *combinational compressed scan*, or more generally, *compressed scan*, to refer to compressed scan with the default combinational codec-to-top-level-ports connection. The term *serialized compressed scan* refers to compressed scan that uses serializing logic to provide a serial connection to the codec. The term *serializer* refers to the logic that provides the serial connection.

For a given number of top-level scan inputs and outputs, DFTMAX compression can create up to a maximum number of chains with full X-tolerance, using combinational compressed scan, as shown in [Table 55](#).

Table 55 Number of Available Compressed Scan Chains

Number of inputs	Number of outputs	Max number of full Xtol chains = (number of outputs) x (2(#inputs-1))
1	1	Unavailable
2	2	4
3	3	12
4	4	32

[Table 55](#) shows that the maximum number of chains with full X-tolerance is limited for low pin-count designs and is not available for one-scan-in-one-scan-out designs. The serializer

overcomes this limitation and achieves full X-tolerance for any number of scan inputs and outputs, including as few as 1 scan-in pin and 1 scan-out pin.

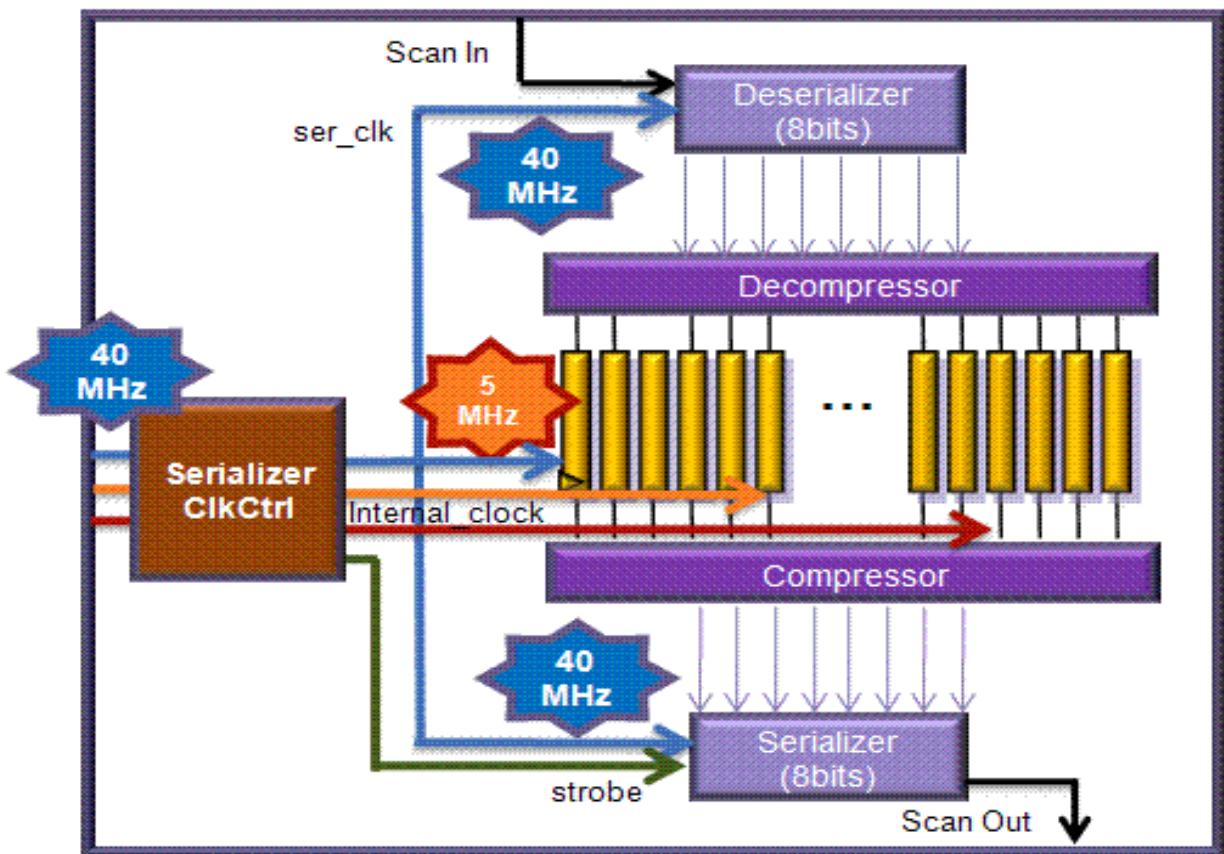
Note:

The maximum number of chains shown in [Table 55](#) will be different if an on-chip clocking (OCC) chain is present in the design.

Architecture

[Figure 373](#) shows the basic serializer architecture.

Figure 373 Basic Serializer Architecture



The serializer architecture consists of the following additional logic:

- Serializer clock controller
- Deserializer registers
- Serializer registers

Serializer Clock Controller

The serializer clock controller contains an FSM counter and clock-gating cells.

- FSM Counter: finite state machine counter creates the clock-enable signal (routed to clock-gating cells) and the strobe signal (routed to the serializer registers). The counter is driven by an external clock.
- CGCs: clock-gating cells are inserted at the specified scan-shift clocks. They are enabled by the FSM counter value and produce internally generated clocks during scan chain shifting.

Deserializer Registers

Deserializing registers are placed in the decompressor IP at the input side. These registers load the scan input data serially and supply the data to the compressed chains.

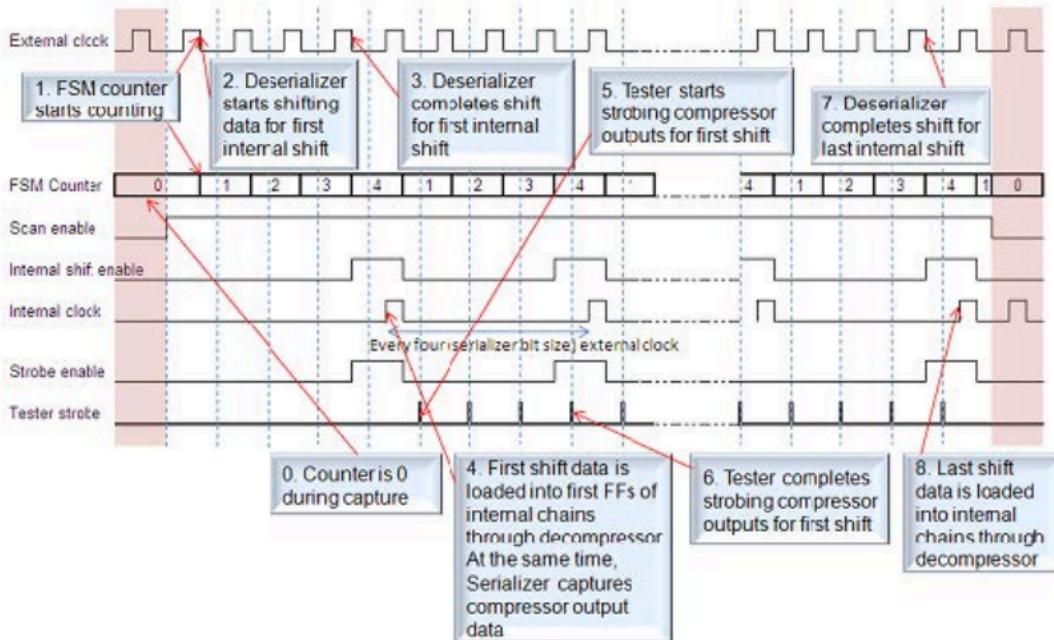
Serializer Registers

Serializing registers are placed in compressor IP at the output side. These registers capture data from the compressor outputs and stream the data to the scan output.

Serializer Operation

[Figure 374](#) shows the serializer operation and timing diagram involving 4-bit deserializer and serializer registers.

Figure 374 Serializer Operation



The sequence of events shown in Figure 374 is as follows:

1. FSM counter starts counting.
2. Deserializer registers start shifting data for the first internal shift.
3. Deserializer registers complete the shift for the first internal shift.
4. First shift data is loaded into the first flip-flops of the compressed chains through the decompressor. At the same time, the serializer registers capture the compressor outputs.
5. Tester starts strobing the compressor outputs for the first shift.
6. Tester completes strobing the compressor outputs for the first shift.
7. Deserializer registers complete the shift for the last internal shift.
8. Last shift data is loaded into the compressed chains through the decompressor.

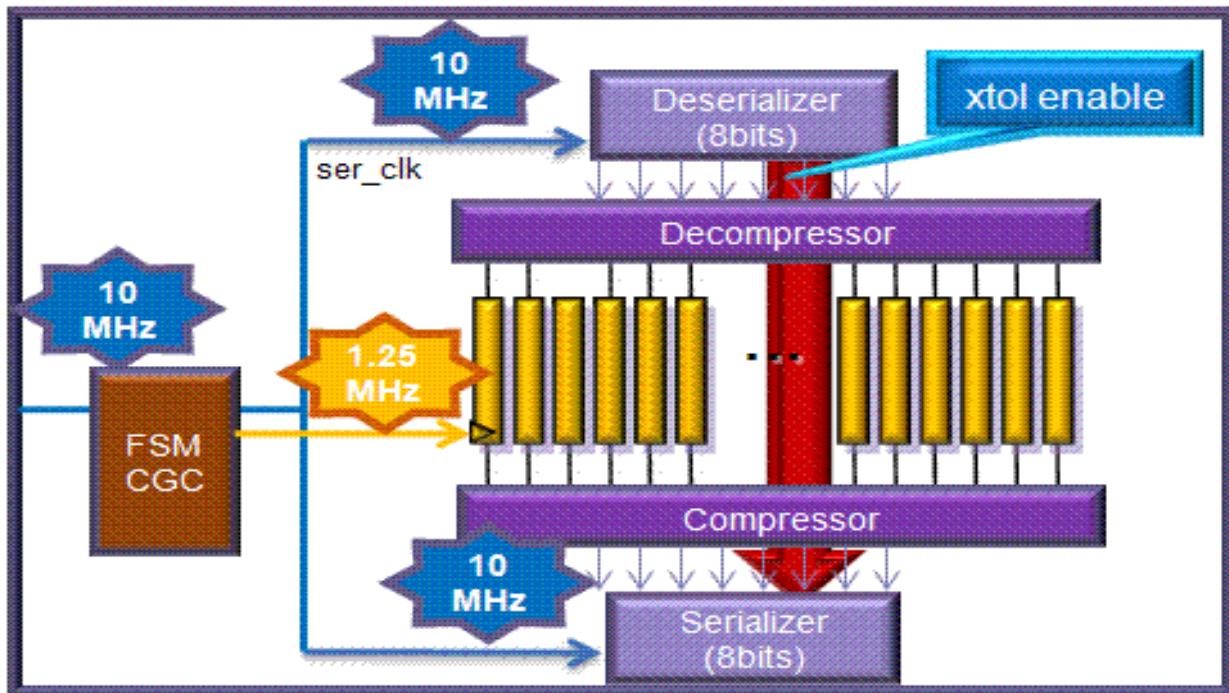
Higher Shift Speed and Update Stage

Because the serializer clock controller behaves like a clock divider, the internal scan clocks are generated by the external clocks being divided by S, where S is the depth of

the serializer register segment. For example, when S is 8 and the external clock speed is 10 MHz, the internal clock speed reduces to 1.25 MHz. Thus, the speed of the external clock can be increased up to S-times faster without affecting the compressed chain shift timing. (An 80 MHz external clock would result in a 10 MHz internal shift clock.)

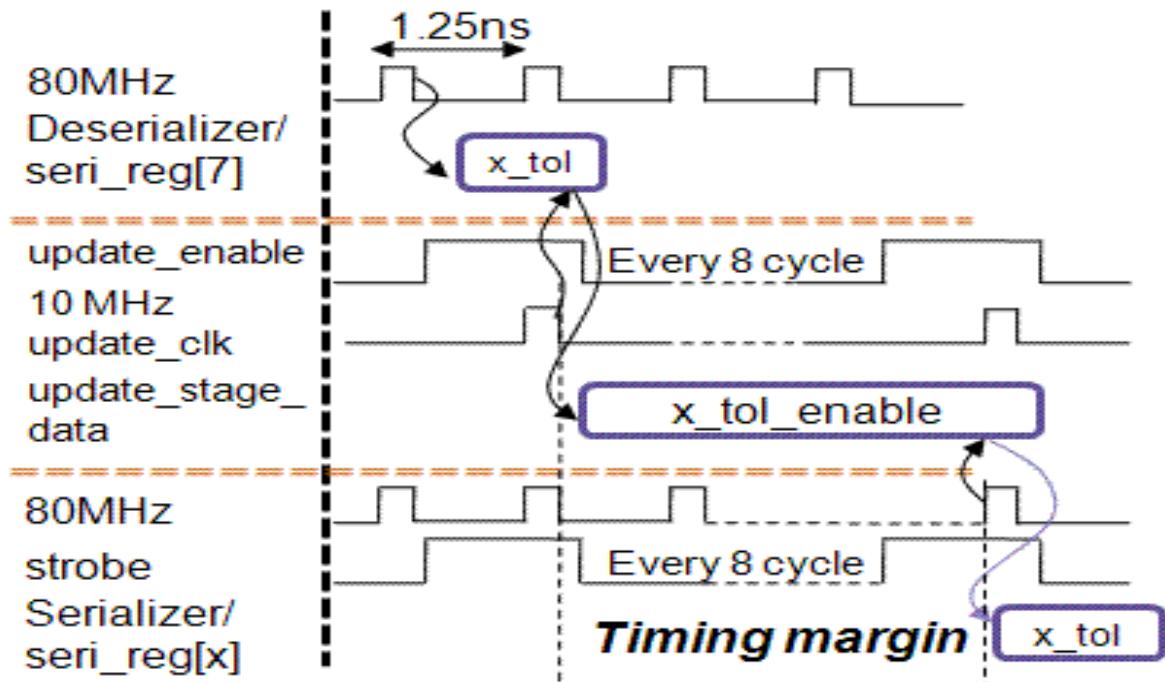
However, in a fully X-tolerant implementation of the compression logic, the longest path traverses from the deserializer flip-flop output pin through the decompressor gates and then through the compressor selector gates to the serializer flip-flop input pin. This path is shown in red in [Figure 375](#).

Figure 375 Serializer Architecture Without Update Stage



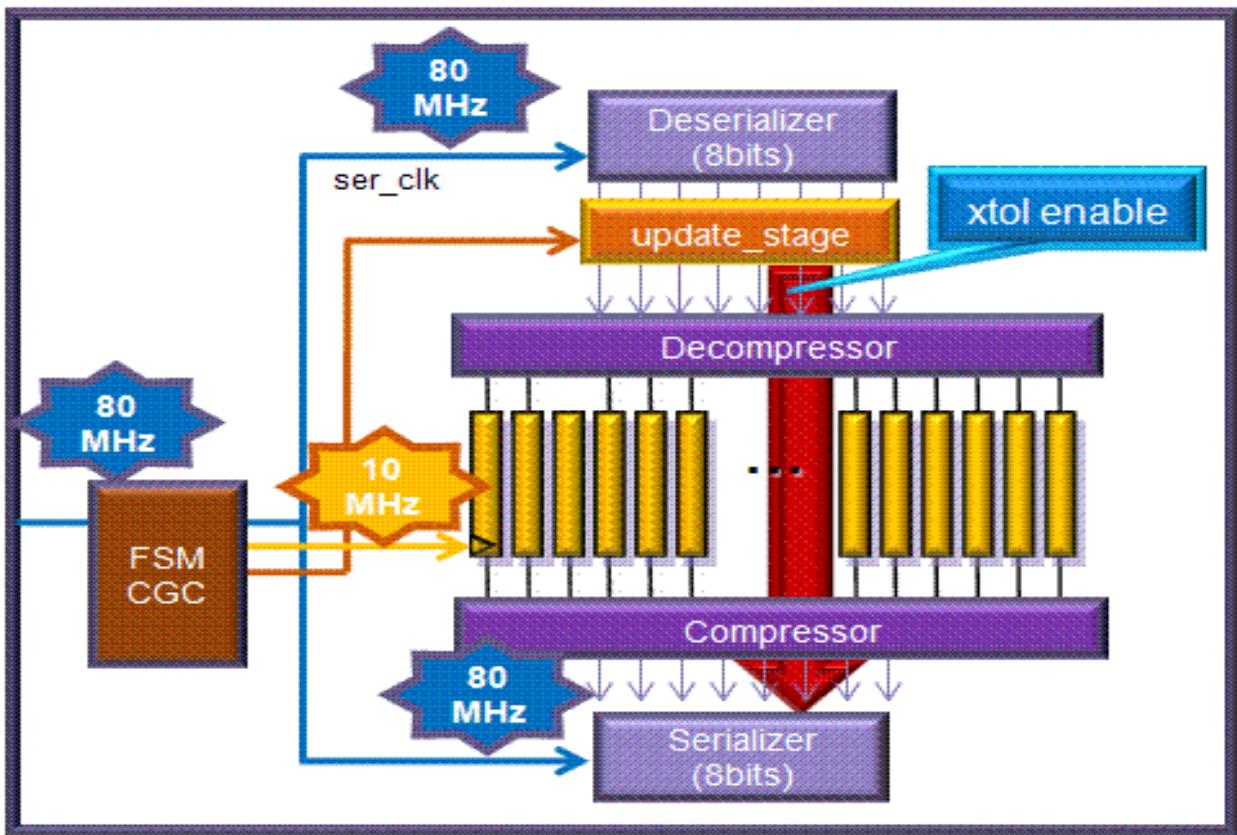
As the period of the serializer clock decreases, this path becomes critical because it cannot exceed the clock period.

Figure 376 Timing Diagram With Update Stage



One way to decouple this critical path from the serializer clock timing is to insert an update stage between the deserializer register output pins and the decompressor combinational logic inputs, as shown in [Figure 377](#). This update stage is synchronized by a clock that runs as fast as the internal shift clock.

Figure 377 Serializer Architecture With Update Stage

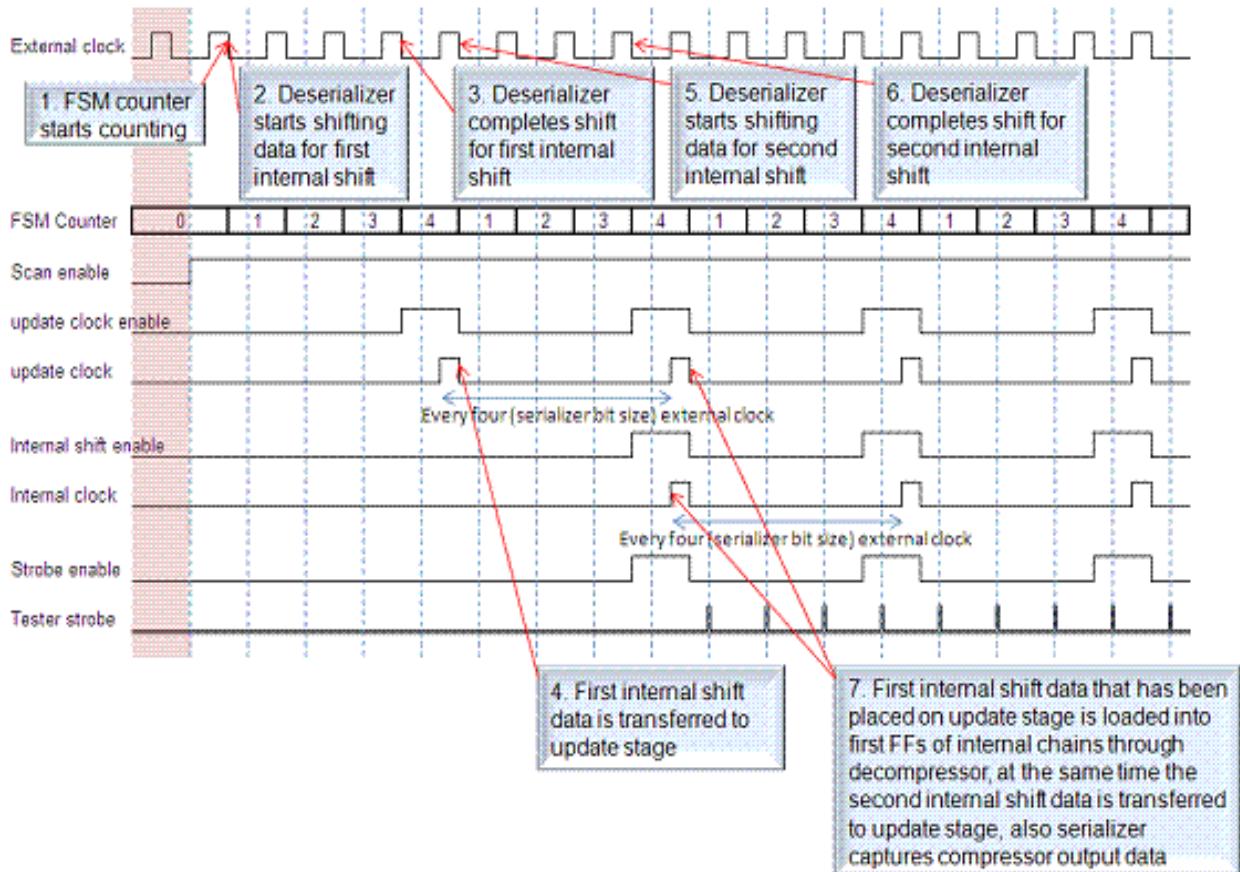


The update stage has the following effects:

- Provides an extra timing margin, as indicated in [Figure 376 on page 828](#).
- Results in one internal clock cycle longer than a serializer architecture without an update stage. So, for example, if there were 100 test cycles per pattern without the update stage, then with the update stage, the whole test cycle per pattern would be 100 cycles plus 1 internal clock cycle.
- Decreases the test time by using a faster external clock.

[Figure 378](#) shows the scan shift operation with the update stage implemented.

Figure 378 Serializer Operation With Update Stage



Scan-Enable Signal Requirements for Serializer Operation

As previously discussed, the clocks for compressed scan chains are internally generated by the serializer clock controller. The serializer FSM counter in the controller counts when the scan-enable signal is active and feeds the clocks to the compressed scan chains at the proper time. When the scan-enable signal becomes inactive during capture, the FSM counter is reset and the external clocks directly clock the compressed scan chain. Therefore, for proper operation, *the scan-enable signal must be held in the inactive state in all capture procedures*.

If you use the STIL protocol file created by the TestMAX DFT tool, the protocol already meets this requirement. The tool constrains all scan-enable signals to the inactive state in the capture procedures, excluding any scan-enable signals defined with the `-usage clock_gating` option of the `set_dft_signal` command.

If you use a custom STIL protocol file, keep the following in mind:

- Make sure that all scan-enable signals used by serializer clock controllers are constrained to the inactive state in all capture procedures.
- If clock pulses are needed to initialize the internal registers whose clock pins are gated by the serializer clock controller, make sure that the scan-enable signal is inactive throughout the `test_setup` procedure. By keeping the scan-enable signal inactive, the clock pulses needed for initialization can reach the initialization registers. Otherwise, the initialization might not be performed properly, and you could see unexpected R-rule violations in DRC.

Timing Paths

The following figures show the datapaths specific to the serialized compressed scan architecture. [Figure 379](#) shows the datapaths without the update stage. [Figure 380](#) shows the datapaths with the update stage. The figures demonstrate the use of the DFT-inserted OCC controller and external clocks.

Figure 379 Datapath Diagrams Without Update Stage

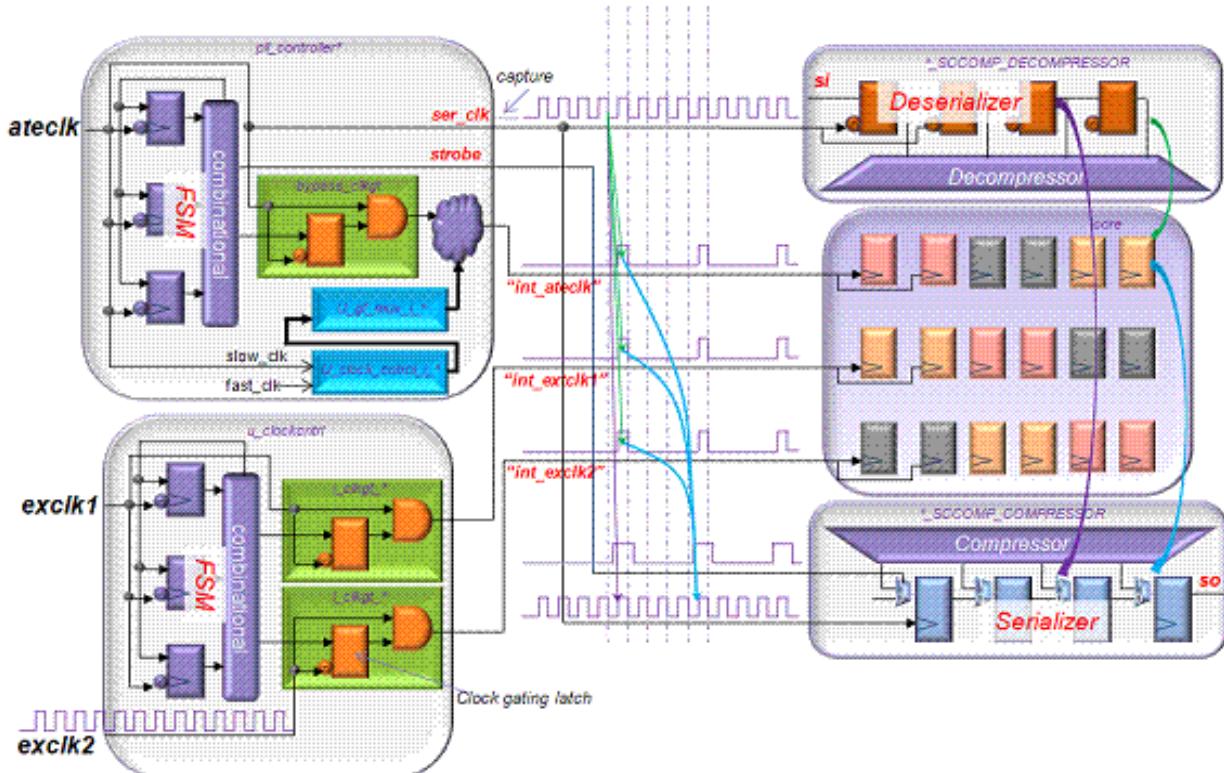
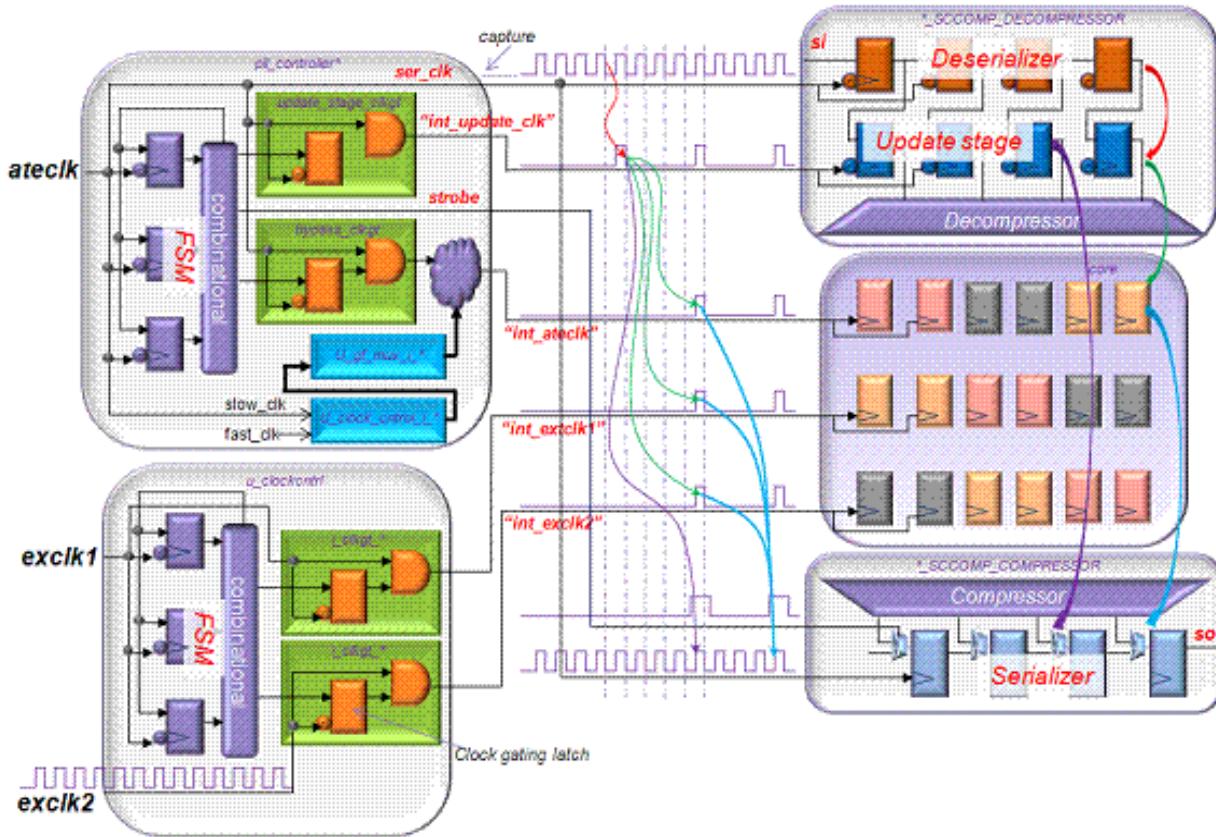


Figure 380 Datapath Diagrams With Update Stage



In Figure 379 and Figure 380, the red, purple, green, and blue arrows in the waveforms and the colored arrows in the block diagrams at the right side of the figures correspond to each other. The arrows are focused on the paths from the decompressor block to the core and then from the core to the compressor block. These diagrams are also useful in understanding the role of the update stage, which relaxes the timing path from decompressor to core to compressor.

Scan Clocks

The following topics discuss scan shift clocks when using a serializer:

- Deserializer/Serializer Update Stage Register Clocks
- Specifying a Clock for Deserializer/Serializer Registers
- Staggered Scan Clocks
- Specifying Scan Clock Ports

Deserializer/Serializer Update Stage Register Clocks

You use one of the scan shift clocks, defined with the command `set_dft_signal -type ScanClock` command, as the clock for the deserializer/serializer registers. If you do not specify a particular clock, the clock for these registers is selected as follows:

- When OCC is used and an automated pipeline scan data register clock is defined as one of the ATE clocks, then the pipeline scan data clock is used for the deserializer/serializer registers.
- In other cases, any clock that reduces the number of lock-up latches is used for the deserializer/serializer registers.

You can determine which clock is selected for the deserializer/serializer registers by using the `preview_dft` command, as shown by the following:

```
Load/Unload Serializer Clock = CLK1
```

Load stands for the input-side deserializer registers, and Unload stand for the output-side serializer registers.

The update stage is implemented by specifying the command

`set_serialize_configuration -update_stage true`. The clock for the update stage always has the same clock source as the deserializer/serializer register clock. The update-stage clock is a gated version of the deserializer/serializer register clock.

Specifying a Clock for Deserializer/Serializer Registers

You can specify a clock for the deserializer/serializer registers in a chip-level flow. If you have multiple scan clocks and have a particular clock that you need to use for the deserializer/serializer registers, you can follow [Example 120](#).

Example 120 Script Example for Specifying a Clock for the Deserializer/Serializer Registers

```
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port EXT_CLK1 -test_mode all
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port EXT_CLK2 -test_mode all
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port EXT_CLK3 -test_mode all
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -serializer_clock EXT_CLK2 \
    -update_stage true
```

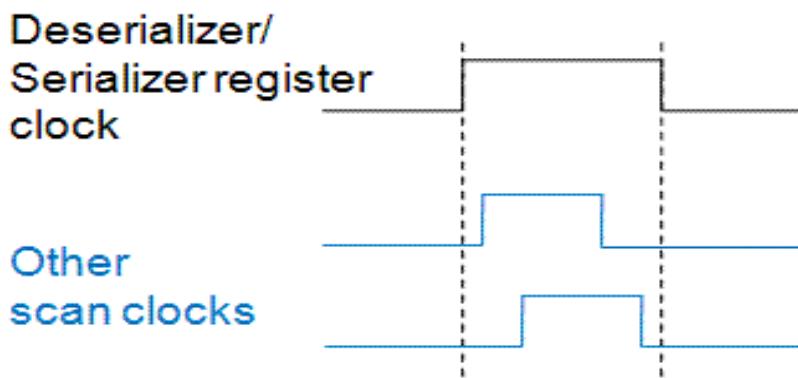
You can specify a clock for the deserializer/serializer registers with the command `set_serialize_configuration -serializer_clock`. The clock specified with the

`-serializer_clock` option has to be predefined with the command `set_dft_signal -type ScanClock`.

Staggered Scan Clocks

If you want to use staggered scan-shift clocks to reduce power during shift, the clock waveforms should be carefully considered. Since the input side of the deserializer registers and the update stage are triggered by the trailing edge, the timing of internal scan cells that obtain the data from the deserializer registers-update stage must not occur later than the trailing edge. Similarly, since the output side of the serializer registers is triggered by the leading edge, the timing of the internal scan cells that launch the data through the compressor to the serializer registers must not occur earlier than the leading edge. A possible way to employ staggered clocks is to use the timing waveforms shown in [Figure 381](#) for the shift mode.

Figure 381 Staggered Clock Waveform



Suppose, for example, your design has five external clocks, each with different timing, as shown the following group of `set_dft_signal` command:

```
set_dft_signal -type ScanClock -view existing_dft -timing {47 88} \
    -port EXT_CLK1 -test_mode all
set_dft_signal -type ScanClock -view existing_dft -timing {50 68} \
    -port EXT_CLK2 -test_mode all
set_dft_signal -type ScanClock -view existing_dft -timing {55 73} \
    -port EXT_CLK3 -test_mode all
set_dft_signal -type ScanClock -view existing_dft -timing {60 78} \
    -port EXT_CLK4 -test_mode all
set_dft_signal -type ScanClock -view existing_dft -timing {65 83} \
    -port EXT_CLK5 -test_mode all
```

With these specifications, the tool automatically selects EXT_CLK1 as the deserializer(serializer register clock. It is the only clock that can make the serializer scheme work.

Note:

For this example, even if you choose a clock other than EXT_CLK1 by using the `set_serialize_configuration -serializer_clock clock_name` command, the tool ignores your specification.

Specifying Scan Clock Ports

When you use the serializer technology, you should instruct the tool where to insert the serializer clock controller. It should be a pin on a clock line that drives all the scan cells. The following command example shows how to make this specification when you have pads for the scan clock ports:

```
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port EXT_CLK1 -test_mode all

set_dft_signal -view spec -type ScanClock -port EXT_CLK1 \
    -hookup_pin F1/Y -test_mode all
```

The value of the `-hookup_pin` option tells the tool where to insert the serializer clock controller. Without this option specification, the clock controller would be inserted close to the port, which might be outside the clock pad and therefore might adversely affect circuit behavior.

User Interface

The `set_scan_compression_configuration -serialize` command allows you to specify the location of the serializer logic. When you want to implement the serializer, specify the either `chip_level` or `core_level` for the `-serialize` option. The default is `none`, which indicates that no serialization is requested and combinational compressed scan is implemented.

```
set_scan_compression_configuration ...
    -serialize chip_level | core_level | none
```

The `set_serialize_configuration` command is used specifically to define the serializer options. Additional options are available to configure the serializer insertion:

```
set_serialize_configuration ...
    -test_mode name
    -parallel_mode name
    -inputs number
    -outputs number
    -update_stage true | false
    -exclude_clocks name
    -serializer_clock name
    -update_clock name
    -strobe name
```

```
-ip_inputs number
-ip_outputs number
-wide_duty_cycle true | false
```

Note the following conditions:

- The default for the `-update_stage` option is false.
- The `-exclude_clocks` option can be used for certain clocks that are not to be gated by the serializer clock controller.
- The specified number of `-inputs` and `-outputs` has to be the same.
- The `-ip_inputs` and `-ip_outputs` options are used for serializer IP insertion flow. For information on this flow, see [Serializer IP Insertion on page 859](#).
- For the usage of `-serializer_clock`, `-update_clock` and `-strobe`, see [Scan Clocks on page 832](#) and [Serialized Compressed Scan Core Creation on page 838](#).
- The default for the `-wide_duty_cycle` option is false. For the wide duty cycle support, see [Wide Duty Cycle Support for Serializer on page 875](#).

To see how these options are used, refer to the script examples in the following top-down flat flow, top-down partition flow, HASS flow, and Hybrid flow sections.

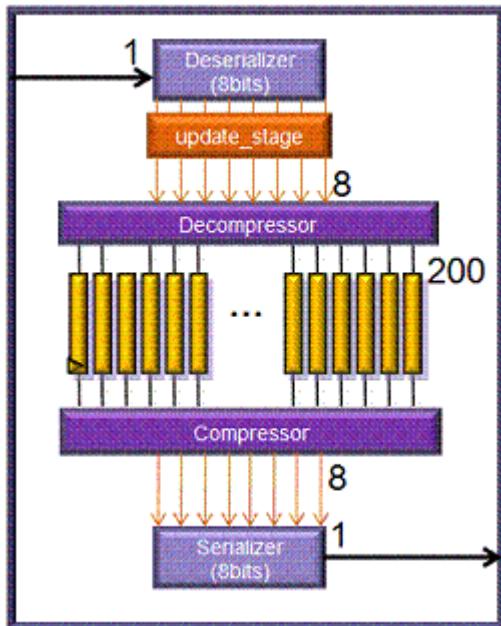
Configuring Serialized Compressed Scan

[Figure 382](#) and the accompanying script to the right of the figure show you how to use the following two commands together:

- `set_scan_compression_configuration` defines the codec specification.
- `set_serialize_configuration` defines the serializer specification.

Using these commands results in the architecture shown in [Figure 382](#).

Figure 382 Association of set_serialize_configuration With Existing set_scan_compression_configuration Command



```
set_scan_compression_configuration \
# codec inputs
-inputs 8 \
# codec outputs
-outputs 8 \
# Compressed chains
-chain_count 200 \
# Enabling serializer specification
-serialize chip_level
set_serialize_configuration \
# serial input
-inputs 1 \
serial output
-outputs 1 \
# update stage
-update_stage true
```

Deserializer/Serializer Register Size

Deserializer/serializer register size is determined by the command:

```
set_scan_compression_configuration -inputs number -outputs number
```

The deserializer(serializer register size is equal to the number of codec inputs and outputs. For example, if this number is 8, then 8-bit deserializer(serializer registers are implemented.

The deserializer(serializer register is divided into some number of segments, depending on the number specified in this command:

```
set_serialize_configuration -inputs number -outputs number
```

For example, if the number is 2, then two 4-bit deserializer(serializer segments are created.

Serializer Implementation Flow

When you insert a serializer, two implementation flows are available:

- Core-level flow
- Chip-level flow

The core-level flow is used for serialized compressed scan core creation. The serialized compressed scan cores are integrated at the chip level in either a HASS or Hybrid flow.

The core-level flow is enabled by the command `set_scan_compression_configuration -serialize core_level`. See [Serialized Compressed Scan Core Creation on page 838](#).

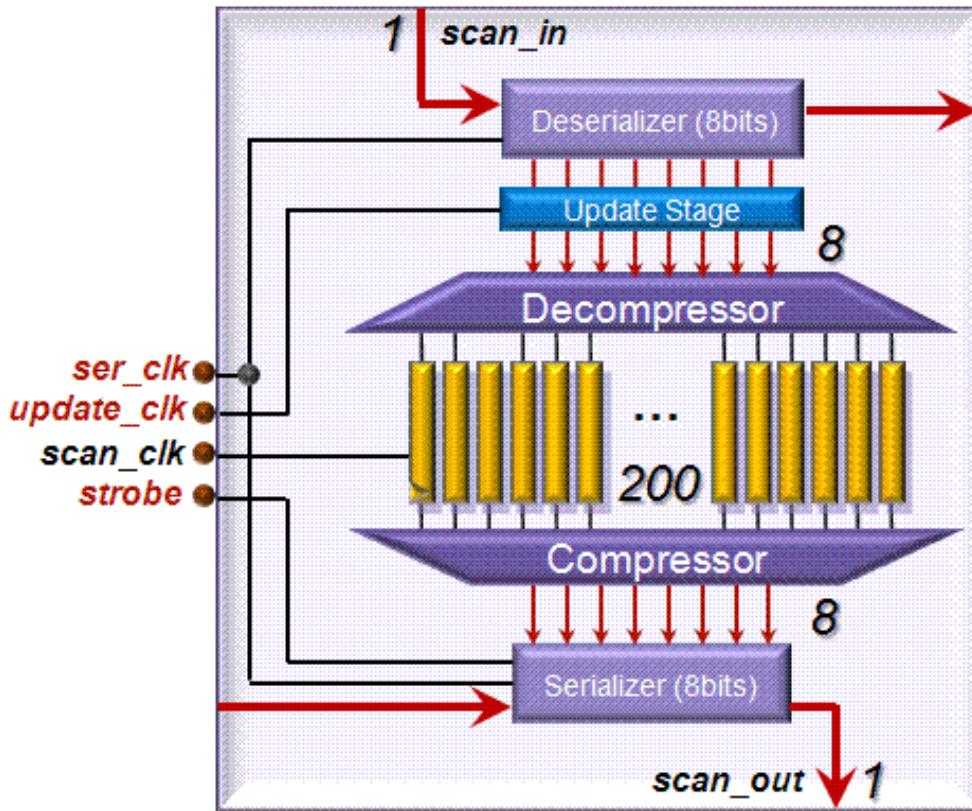
The chip-level flow is used for top-down flat flow, top-down partition flow, HASS flow, Hybrid flow, and serializer IP insertion flow. The chip-level flow is enabled by the command `set_scan_compression_configuration -serialize chip_level`. See

- [Top-Down Flat Flow on page 842](#)
- [Top-Down Partition Flow on page 844](#)
- [HASS Flow on page 849](#)
- [Hybrid Flow on page 856](#)
- [Serializer IP Insertion on page 859](#)

Serialized Compressed Scan Core Creation

You can create serialized compressed scan cores and integrate them at the chip level in a HASS flow or a Hybrid flow. For a serialized compressed scan core-level flow, use the command `set_scan_compression_configuration -serialize core_level`. [Figure 383](#) shows a diagram for the serialized compressed scan core.

Figure 383 Serialized Compressed Scan Core



Note the following in Figure 383:

- No serializer clock controller is inserted.
 - The ports for `ser_clk`, `update_clk`, and `strobe` are automatically created. Therefore the serializer clock controller that is inserted at the top level during HASS or Hybrid flows feeds signals to these ports.
- Each deserializer(serializer register segment) has an interface that connects to and from the different cores. Even if you specify `set_serialize_configuration -inputs 1 -outputs 1`, two scan-in and scan-out ports are created, as shown in Figure 383.

Serializer Core-Level Flow

Example 121 shows a typical serialized compressed scan core-level flow:

Example 121 Script Example for a Serialized Compressed Scan Core-Level Flow

```
set_scan_configuration -chain_count 1
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 200 \
    -inputs 8 \
    -outputs 8 \
    -serialize core_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
create_test_protocol
dft_drc
insert_dft
```

Note:

With regard to command requirements for serialized compressed scan core creation, you must specify the same value in the `set_scan_configuration -chain_count number` and `set_serialize_configuration -inputs number -outputs number` commands for each core-level configuration. Otherwise, you will see unexpected errors in the HASS or Hybrid flows at the chip level.

User-Defined Ports for the Serializer Core-Level Flow

You might need to specify the port usage for each core before core creation. For example, suppose you want to specify the particular ports `ser_clk`, `update_clk`, and `strobe`, as the serializer control ports. You could use a script such as the one shown in [Example 122](#).

Example 122 Script Example for Defining Core-Level Serializer Ports

```
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port MY_SERI_CLK -test_mode all
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port MY_UPD_CLK -test_mode all
set_dft_signal -view spec -type TestData \
    -port MY_STROBE -test_mode all
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true \
    -serializer_clock MY_SERI_CLK \
    -update_clock MY_UPD_CLK \
    -strobe MY_STROBE
```

The ports specified with the `-serializer_clock` and `-update_clock` options must be predefined as test clocks with the `set_dft_signal -type ScanClock` command,

and the port specified with the `-strobe` option must be predefined as test data with the `set_dft_signal -type TestData` command. All of these ports must be dedicated existing test ports. The tool connects the signals to the specified ports, instead of creating them.

Nondefault Scan Clock Timing for Core-Level Flows

If you use clock timing that is not the default clock timing (period of 100, rise of 45, and fall of 55) for a flow that is specified with the `set_scan_compression_configuration -serialize core_level` command, you might encounter the following error message when you run the `preview_dft` command on the core-level flow:

```
Error: Cannot synchronize scan chain cells 'U_core1/stat_reg_1_' and
'Serializer_clk_seg'. Both edges of clock 'clkA' occur before clock
'ser_clk' triggers. (TEST-344)
```

This error can occur because by default the tool automatically creates a serializer clock named `ser_clk` and assumes it has the default timing. If you need to use nondefault clock timing in a core-level implementation flow, you should define a clock port with nondefault timing specified, as shown in [Example 123](#).

Example 123 Serializer Core-Level Flow Using a Nondefault Clock

```
set_dft_signal -view existing_dft -type ScanClock -timing {20 30} \
    -port EXT_CLK1 -test_mode all
set_dft_signal -view existing_dft -type ScanClock -timing {20 30} \
    -port EXT_CLK2 -test_mode all
set_dft_signal -view existing_dft -type ScanClock -timing {20 30} \
    -port MY_SER_CLK -test_mode all
set_dft_signal -view existing_dft -type ScanClock -timing {20 30} \
    -port MY_UPD_CLK -test_mode all
set_dft_signal -view spec -type TestData -port MY_STROBE -test_mode all
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -serializer_clock MY_SERI_CLK \
    -update_clock MY_UPD_CLK \
    -strobe MY_STROBE \
    -update_stage true
```

As in this example, when you specify a clock port with `-update_clock`, you should set the same clock timing as the clock port specified with `-serializer_clock`, because the clock timing becomes the same after a serializer clock controller is inserted at the top level during a HASS or Hybrid flow.

Top-Down Flat Flow

The following two topics describe typical command flows when a serializer codec is inserted at a top level that does not yet contain a codec:

- [Serial Mode and Standard Scan Mode](#)
- [Serial Mode, Parallel Mode, and Standard Scan Mode](#)

Serial Mode and Standard Scan Mode

By default, without a user-defined test mode, DFTMAX compression architects the Internal_scan and ScanCompression_mode modes. When the `-serialize chip_level` option is specified, the ScanCompression_mode becomes the serializer mode, which in this discussion is called *serial mode* as a matter of convenience. The script in [Example 124](#) shows you how to create the serial mode.

Example 124 Script Example for a Top-Down Flat Flow: Two Modes

```
set_scan_configuration -chain_count 2
set_scan_compression_configuration \
    -inputs 8 \
    -outputs 8 \
    -chain_count 200 \
    -serialize chip_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
create_test_protocol
dft_drc
insert_dft
...
# Serial mode
write_test_protocol -output SERIAL.spf -test_mode ScanCompression_mode

# Standard scan mode
write_test_protocol -output SCAN.spf -test_mode Internal_scan
```

In this script example, one scan-in port and one scan-out port are created in the serial mode and two scan-in ports and two scan-out ports are created in the standard scan mode.

Serial Mode, Parallel Mode, and Standard Scan Mode

In addition to the serial mode provided with the serializer, you can also create a “parallel mode.” This mode is logically equivalent to combinational compressed scan as shown in

Figure 384. The parallel mode is created with the `-parallel_mode` option, as shown in the script of [Example 125](#).

Example 125 Script Example for a Top-Down Flat Flow: Three Modes

```

define_test_mode my_regular -encoding {TM1 0 TM2 0} \
    -usage scan
define_test_mode my_parallel -encoding {TM1 0 TM2 1} \
    -usage scan_compression
define_test_mode my_serial   -encoding {TM1 1 TM2 0} \
    -usage scan_compression
for {set i 0} {$i < 8} {incr i} {
    set_dft_signal -view spec -type ScanDataIn -port SI_${i} \
        -test_mode all
    set_dft_signal -view spec -type ScanDataOut -port SO_${i} \
        -test_mode all
}
set_scan_configuration -chain_count 2 -test_mode my_regular
set_scan_compression_configuration \
    -base_mode my_regular \
    -test_mode my_serial \
    -xtolerance high -chain_count 200 -inputs 8 -outputs 8 \
    -static_x_chain_isolation true \
    -serialize chip_level
set_serialize_configuration \
    -test_mode my_serial \
    -parallel_mode my_parallel \
    -inputs 1 -outputs 1 \
    -update_stage true
create_test_protocol
dft_drc
insert_dft
...
write_test_protocol -output SERIAL.spf -test_mode my_serial
write_test_protocol -output PARALLEL.spf -test_mode my_parallel
write_test_protocol -output SCAN.spf -test_mode my_regular

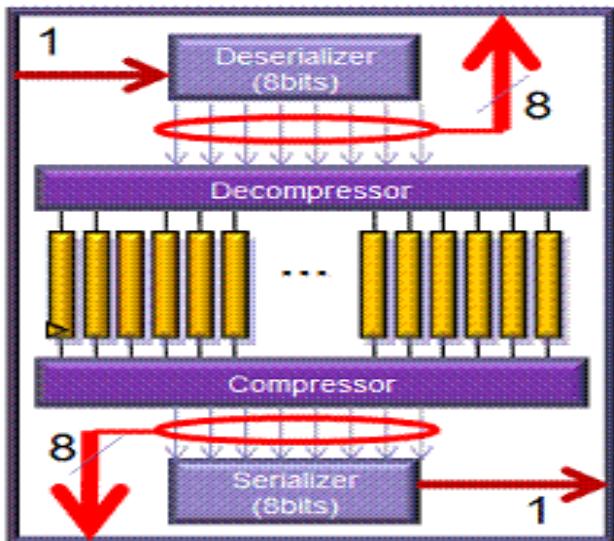
```

Be aware that `my_parallel` as a parallel mode is specified with the `-parallel_mode` option of the `set_serialize_configuration` command and `my_serial` as a serial mode is specified with the `-test_mode` option of the `set_scan_compression_configuration` command.

In this example, one scan-in port and one scan-out port are created in the serial mode, eight scan-in and scan-out ports are created in the parallel mode, and two scan-in and scan-out ports are created in the standard scan mode.

When the parallel mode is implemented along with the serial mode, only one codec for the serial mode is implemented and shared with the parallel mode. [Figure 384](#) shows how it is shared.

Figure 384 Top-Down Flat Flow Diagram



Top-Down Partition Flow

For a top-down partition flow, you can specify serializer insertion in each partition. If you specify the `-update_stage true` option with the `set_serialize_configuration` command in one partition, the option setting is applied to all other partitions.

There are two top-down partition flows: one that uses dedicated serializer chains for each partition, and one that concatenates the serializer chains across partitions.

You can create a parallel mode, such as a top-down flat flow, by employing user-defined test modes, but this capability is available only on the dedicated serializer chain flow.

For the concatenated serializer chain flow, the generated test modes are

- `ScanCompression_mode`: serial mode
- `Internal_scan`: standard scan mode

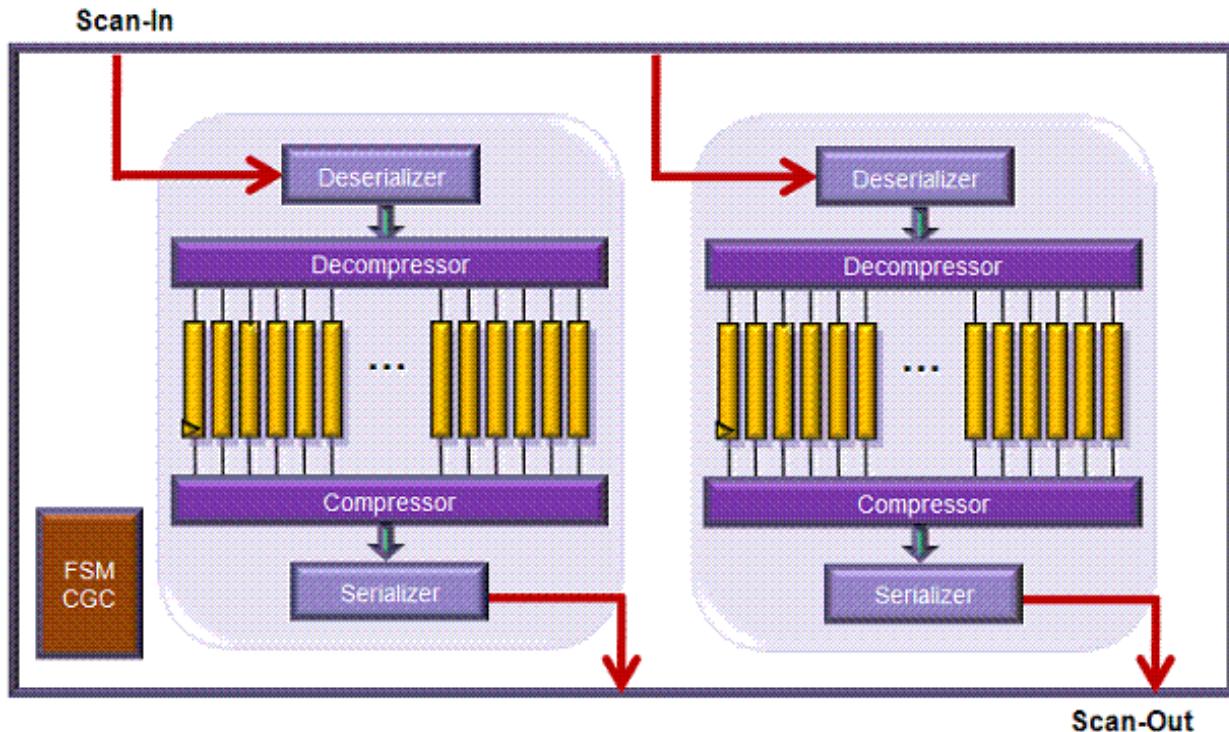
Note:

The top-down partition flow with core-level serializer insertion is not supported. Core-level serializer insertion is discussed in the [Serialized Compressed Scan Core Creation on page 838](#).

Serializer Chains Dedicated to Each Partition

You can divide the design into multiple partitions and insert serializer codecs into each partition. [Figure 385](#) shows a top-down partition flow for which serializer chains are dedicated to each partition. [Example 126](#) shows the script for this case for two modes.

Figure 385 Top-Down Partition Flow With Serializer Chains Dedicated to Each Partition



Example 126 Script Example for a Top-Down Partition Flow With Serializer Chains Dedicated to Each Partition: Two Modes

```
define_dft_partition partition1 -include [list U0 U1 U2 U3 U4]
current_dft_partition partition1
set_scan_configuration -chain_count 8 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -location U0 -xtolerance high -chain_count 200 \
    -inputs 8 -outputs 8 \
    -serialize chip_level
set_serialize_configuration \
    -inputs 1 -outputs 1 \
    -update_stage true

current_dft_partition default_partition
set_scan_configuration -chain_count 10 -clock_mixing mix_clocks
```

```

set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -location U5 -xtolerance high -chain_count 300 \
    -inputs 10 -outputs 10 \
    -serialize chip_level
set_serialize_configuration \
    -inputs 1 -outputs 1 \
    -update_stage true
create_test_protocol
dft_drc
insert_dft
...
write_test_protocol -output SERIAL.spf -test_mode ScanCompression_mode
write_test_protocol -output SCAN.spf -test_mode Internal_scan

```

In the script shown in [Example 126](#), two scan-in and scan-out ports are created in the serial mode and 18 scan-in and scan-out ports are created in the standard scan mode, at the top level.

[Example 127](#) shows this case for three modes.

Example 127 Script Example for a Top-Down Partition Flow With Serializer Chains Dedicated to Each Partition: Three Modes

```

define_test_mode my_regular -encoding {TM1 0 TM2 0} \
    -usage scan
define_test_mode my_parallel -encoding {TM1 0 TM2 1} \
    -usage scan_compression
define_test_mode my_serial -encoding {TM1 1 TM2 0} \
    -usage scan_compression

define_dft_partition partition1 -include [list U0 U1 U2 U3 U4]
current_dft_partition partition1
set_scan_configuration -chain_count 8 -test_mode my_regular
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -base_mode my_regular \
    -test_mode my_serial \
    -location U0 -xtolerance high -chain_count 200 \
    -inputs 8 -outputs 8 \
    -serialize chip_level
set_serialize_configuration \
    -test_mode my_serial \
    -parallel_mode my_parallel \
    -inputs 1 -outputs 1 \
    -update_stage true

current_dft_partition default_partition
set_scan_configuration -chain_count 10 -test_mode my_regular
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \

```

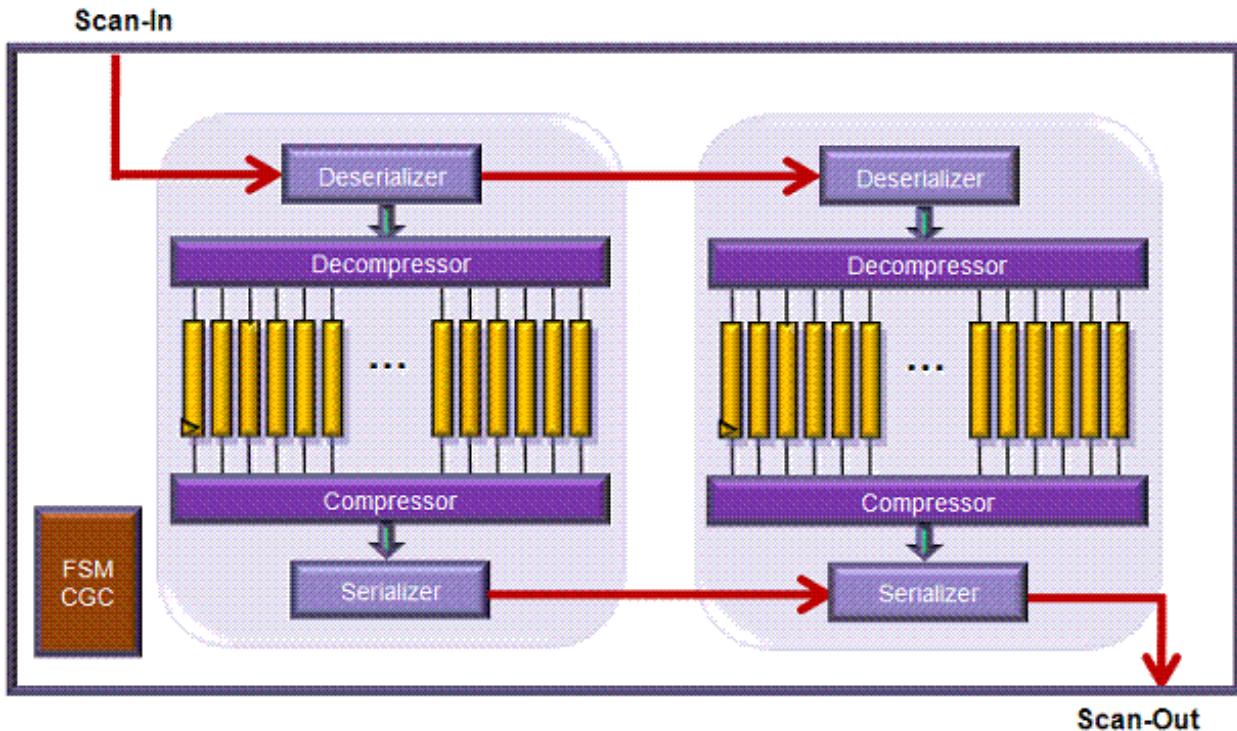
```
-base_mode my_regular \
-test_mode my_serial \
-location U5 -xtolerance high -chain_count 300 \
-inputs 10 -outputs 10 \
-serialize chip_level
set_serialize_configuration \
-test_mode my_serial \
-parallel_mode my_parallel \
-inputs 1 -outputs 1 \
-update_stage true
create_test_protocol
dft_drc
insert_dft
...
write_test_protocol -output SERIAL.spf -test_mode my_serial
write_test_protocol -output PARALLEL.spf -test_mode my_parallel
write_test_protocol -output SCAN.spf -test_mode my_regular
```

In the script shown in [Example 127](#), two scan-in and scan-out ports are created in the serial mode, and 18 scan-in and scan-out ports are created in the parallel mode, and 18 scan-in and scan-out ports are created in the standard scan mode, at the top level.

Serializer Chains Concatenated Across Partitions

Scan ports can access across the multiple partitions. [Figure 386](#) shows the case for a design with two partitions. [Example 128](#) shows the script for this case.

Figure 386 Top-Down Partition Flow With Serializer Chains Concatenated Across Partitions



For this architecture, the following three command requirements should be observed:

- You should use the `-partition all` option when defining top-level scan ports with the `set_dft_signal` command.
- You should not specify the `-inputs` and `-outputs` options with the `set_serialize_configuration` command in any partition.
- The number of scan-in and scan-out port pairs defined by the `set_dft_signal -partition all` commands should be exactly the same number as the `set_scan_configuration -chain_count number` command specified in each partition.

Example 128 Script Example for a Top-Down Partition Flow With Serializer Chains Concatenated Across Partitions

```
set_dft_signal -view spec -type ScanDataIn -port SI_TOP -partition all
set_dft_signal -view spec -type ScanDataOut -port SO_TOP -partition all
define_dft_partition partition1 -include [list U0 U1 U2 U3 U4]
current_dft_partition partition1
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
```

```

-location U0 -xtolerance high -chain_count 200 \
-inputs 8 -outputs 8 \
-serialize chip_level
set_serialize_configuration \
-update_stage true

current_dft_partition default_partition
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -location U5 -xtolerance high -chain_count 300 \
    -inputs 10 -outputs 10 \
    -serialize chip_level
set_serialize_configuration \
    -update_stage true
create_test_protocol
dft_drc
insert_dft
...
write_test_protocol -output SERIAL.spf -test_mode ScanCompression_mode
write_test_protocol -output SCAN.spf -test_mode Internal_scan

```

In the script shown in [Example 128](#), the commands have one pair of scan-in and scan-out settings with the `-partition all` option, and a chain count value of 1 is specified with the `set_scan_configuration -chain_count` command. Additionally, the `-inputs number` and `-outputs number` options are not specified in the `set_serialize_configuration` command.

This example creates one scan-in port and one scan-out port in the serial mode and the standard scan mode at the top level.

The tool relies only on the `set_dft_signal -partition all` command to determine whether you are using the concatenated serializer chain flow. Be careful to define the `set_dft_signal -partition all` command for the scan-in and scan-out ports correctly; otherwise, you might encounter unexpected errors.

Note:

Implementing a different number of scan ports between the serial mode and the standard scan mode is not supported.

HASS Flow

You can insert serializer codecs into the core modules and then integrate the multiple cores at the top level.

For core-level serializer insertion, use the `set_scan_compression_configuration -serialize core_level` command on each core. See [Serialized](#)

[Compressed Scan Core Creation on page 838](#). At the top level, use the `set_scan_compression_configuration -serialize chip_level -integration_only true` command. This command enables the tool to insert a serializer clock controller at the top level and to integrate the cores.

Similar to the top-down partition flow, there are two HASS flows:

- One that uses dedicated serializer chains for each core.
- One that concatenates the serializer chains across the cores.

You can create a parallel mode by employing user-defined test modes, but this capability is limited to the dedicated serializer chain flow.

For the concatenated serializer chain flow, the generated test modes are

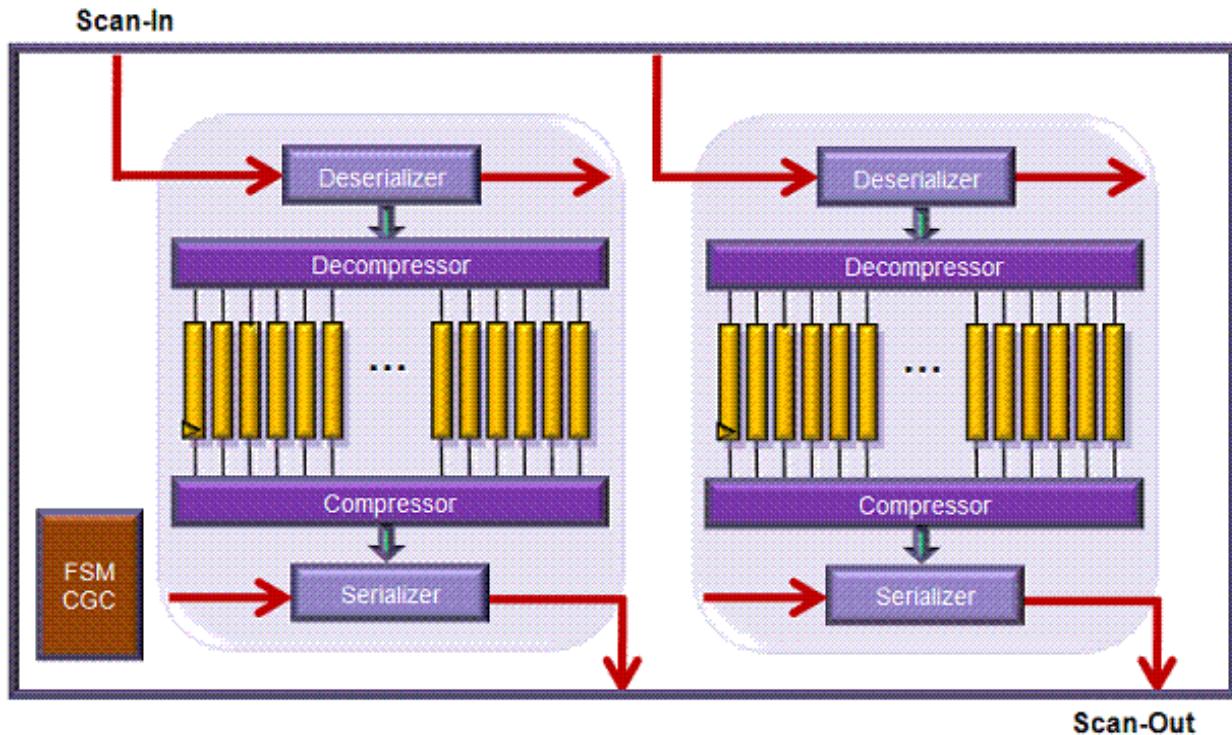
- `ScanCompression_mode`: serial mode
- `Internal_scan`: standard scan mode

Some limitations apply to core integration in the serializer flow. See [DFTMAX Compression With Serializer Limitations on page 909](#).

Serializer Chains Dedicated to Each Core

Serializer chains are created and dedicated to each core. If you have two cores, you would need to have at least two scan-in and scan-out ports at the top level. [Figure 387](#) shows an example for a design with two cores. [Example 129](#) shows the script for this case.

Figure 387 HASS Flow With Serializer Chains Dedicated to Each Core



Example 129 Script Example for a HASS Flow With Serializer Chains Dedicated to Each Core

```
### core1
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks
report_scan_configuration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 150 \
    -inputs 8 \
    -outputs 8 \
    -serialize_core_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
dft_drc
preview_dft
insert_dft
current_test_mode ScanCompression_mode
dft_drc
```

```

current_test_mode Internal_scan
dft_drc

### core2
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks
report_scan_configuration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 250 \
    -inputs 8 \
    -outputs 8 \
    -serialize core_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
dft_drc
preview_dft
insert_dft
current_test_mode ScanCompression_mode
dft_drc
current_test_mode Internal_scan
dft_drc
###For core-level serializer insertion,
###you have to specify the same <number> in the
###set_scan_configuration -chain_count <number> and
###set_serialize_configuration -inputs <number> -outputs <number>
###commands for each core level configuration; otherwise,
###you will see unexpected errors.

### top level
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -integration_only true \
    -serialize chip_level
create_test_protocol
dft_drc
preview_dft
insert_dft
write_hierarchy -output TOP.v -format verilog
write_test_protocol -output SERIAL.spf -test_mode ScanCompression_mode
write_test_protocol -output SCAN.spf -test_mode Internal_scan

```

[Example 130](#) shows the script for the HASS parallel mode flow. In this script example, 2 scan-in and 2 scan-out ports are created for the serial mode and the standard scan mode.

Example 130 Script Example for a HASS Parallel Mode Flow With Serializer Chains Dedicated to Each Core

```

### core1
define_test_mode my_regular -usage scan
define_test_mode my_serial -usage scan_compression
define_test_mode my_parallel -usage scan_compression
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
    -test_mode my_regular
report_scan_configuration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -base_mode my_regular \
    -test_mode my_serial \
    -xtolerance high \
    -chain_count 150 \
    -inputs 8 \
    -outputs 8 \
    -serialize core_level
set_serialize_configuration \
    -test_mode my_serial \
    -parallel_mode my_parallel \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
dft_drc
preview_dft
insert_dft
current_test_mode my_regular
dft_drc
current_test_mode my_serial
dft_drc
current_test_mode my_parallel
dft_drc

### core2
define_test_mode my_regular -usage scan
define_test_mode my_serial -usage scan_compression
define_test_mode my_parallel -usage scan_compression
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
    -test_mode my_regular
report_scan_configuration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -base_mode my_regular \
    -test_mode my_serial \
    -xtolerance high \
    -chain_count 250 \
    -inputs 8 \
    -outputs 8 \

```

```

-serialize core_level
set_serialize_configuration \
    -test_mode my_serial \
    -parallel_mode my_parallel \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
dft_drc
preview_dft
insert_dft
current_test_mode my_regular
dft_drc
current_test_mode my_serial
dft_drc
current_test_mode my_parallel
dft_drc
### For core-level serializer insertion,
### you have to specify the same <number> in the
### set_scan_configuration -chain_count <number> and
### set_serialize_configuration -inputs <number> -outputs <number>
### commands for each core level configuration; otherwise,
### you will see unexpected errors.

### top level
define_test_mode my_regular -encoding {TM1 0 TM2 0} \
    -usage scan
define_test_mode my_serial -encoding {TM1 0 TM2 1} \
    -usage scan_compression
define_test_mode my_parallel -encoding {TM1 1 TM2 0} \
    -usage scan_compression
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks \
    -test_mode all
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -integration_only true \
    -serialize chip_level
create_test_protocol
dft_drc
preview_dft
insert_dft
write -hierarchy -output TOP.v -format verilog
write_test_protocol -output SCAN.spf -test_mode my_regular
write_test_protocol -output SERIAL.spf -test_mode my_serial
write_test_protocol -output PARALLEL.spf -test_mode my_parallel

```

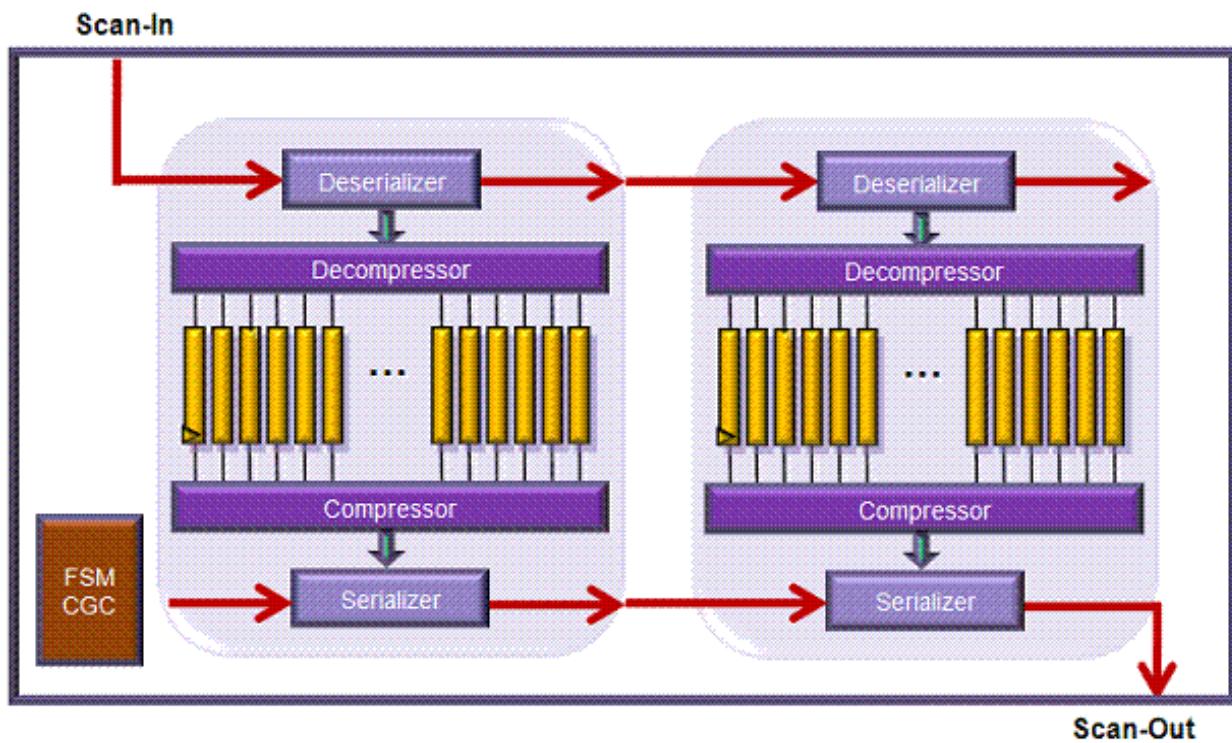
In this script example, 2 scan-in and 2 scan-out ports are created for the serial mode and the standard scan mode. For the parallel mode, 16 scan-in and 16 scan-out ports are created.

Serializer Chains Concatenated Across Cores

You can specify the number of scan ports at the top level so that the serializer chains can be concatenated across multiple cores, as shown in [Figure 388](#). This is controlled at the top level by specifying the `set_scan_configuration -chain_count` command. When you specify a number equivalent to the sum of the numbers specified by `set_serialize_configuration -inputs number -outputs number` at each core level, the core-level scan ports are brought up to the top-level scan ports. They are not concatenated across the cores. On the other hand, if you specify a number less than the sum of the numbers specified by `set_serialize_configuration -inputs number -outputs number` at each core level, only the specified number of scan ports are created at the top level, which is accomplished by concatenating each core-level deserializer/serializer register segment accordingly.

For the design example shown in [Figure 388](#), if you specify the chain count at the top level by using the `set_scan_configuration -chain_count 1` command, only one scan-in and one scan-out port is created for the serial mode and the standard scan mode.

Figure 388 HASS Flow With Serializer Chains Concatenated Across Multiple Cores



You should not specify a number larger than the sum of the numbers specified with the `set_serialize_configuration -inputs number -outputs number` command for each core.

Note:

In the concatenated serializer chain flow, DFTMAX compression might not be able to build an optimal length serializer chain segment. Suppose, for example, that you have 2 scan-ins and scan-outs for a 6-bit serializer on core1, 2 scan-ins and scan-outs for a 6-bit serializer on core2, 2 scan-ins and scan-outs for a 4-bit serializer on core3, and that you want to create 2 scan-ins and scan-outs at the top level.

You might expect two 8-bit serializer segments ($6/2 + 6/2 + 4/2$) to be created at the top level, by concatenating core-level serializer segments. But this might not happen.

The workaround in this case is to create 6 scan-ins and scan-outs for core1 and core2, and 4 scan-ins and scan-outs for core3 at the serializer core-level creation. This would mean that each serializer segment is 1-bit. The tool would then have more flexibility to create 8-bit serializer segments, which is the optimal length at the top level.

Hybrid Flow

If you have performed multiple core-level implementations but still have some user-defined logic at the top level, you can apply the Hybrid flow. The Hybrid flow provides core integration and serializer insertion for the user-defined logic at the same time at the top level.

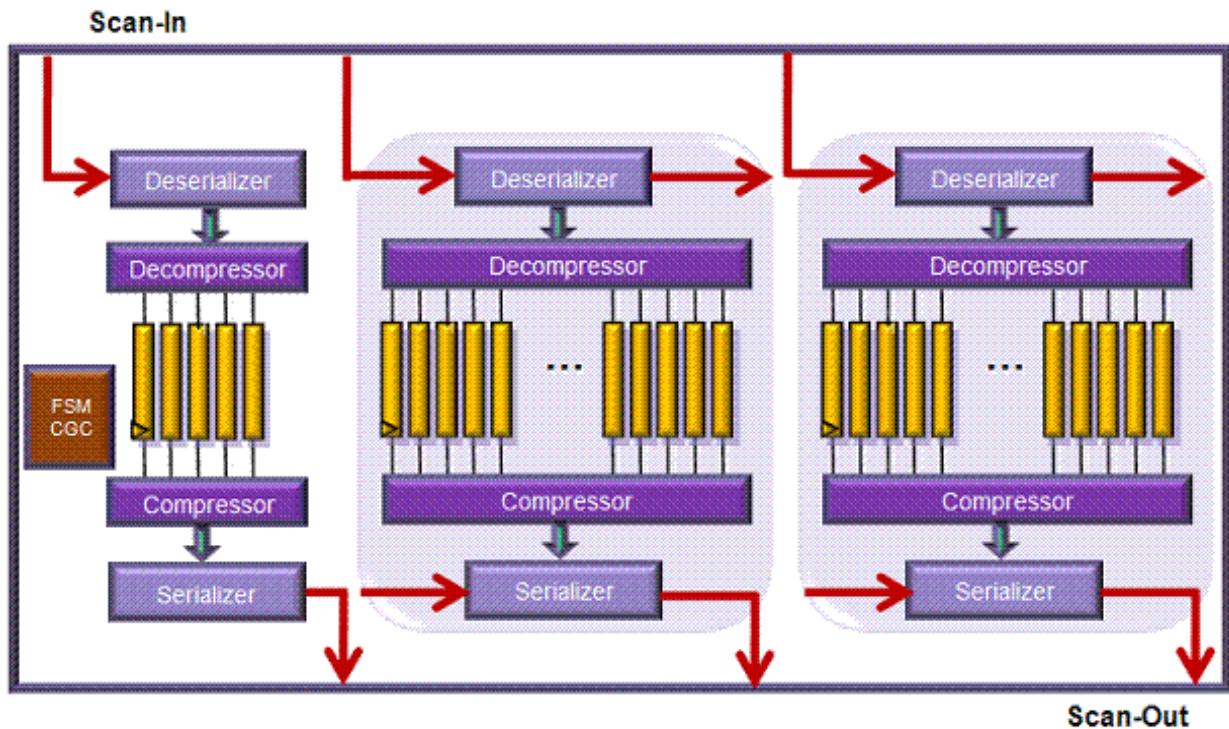
To create a core implemented with a serializer codec, use the `set_scan_compression_configuration -serialize core_level` command at each core level. At the top level, specify the `set_scan_compression_configuration -serialize chip_level -hybrid true` command. This enables the tool to insert a serializer clock controller and a serializer codec for the top-level user-defined logic and then integrate all the serializer cores.

The Hybrid flow does not support a user-defined test mode. You cannot have both a parallel mode and a serial mode. Only a serial mode and a standard scan mode are supported together. The generated test modes are

- `ScanCompression_mode`: serial mode
- `Internal_scan`: standard scan mode

[Figure 389](#) shows the Hybrid flow diagram. [Example 131](#) shows the script for this case.

Figure 389 Hybrid Flow Diagram



Example 131 Script Example for a Hybrid Flow

```
### core1
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks
report_scan_configuration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 150 \
    -inputs 8 \
    -outputs 8 \
    -serialize core_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
dft_drc
preview_dft
insert_dft
current_test_mode ScanCompression_mode
dft_drc
```

```

current_test_mode Internal_scan
dft_drc

### core2
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks
report_scan_configuration
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 600 \
    -inputs 12 \
    -outputs 12 \
    -serialize core_level
set_serialize_configuration \
    -inputs 2 \
    -outputs 2 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
preview_dft
dft_drc
insert_dft
current_test_mode ScanCompression_mode
dft_drc
current_test_mode Internal_scan
dft_drc
### For core-level serializer insertion,
### you have to specify the same <number> in the
### set_scan_configuration -chain_count <number> and
### set_serialize_configuration -input <number> -output <number>
### commands for each core level configuration; otherwise,
### you will see unexpected errors.

### top level
set_scan_configuration -chain_count 4 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 50 \
    -inputs 6 \
    -outputs 6 \
    -hybrid true \
    -serialize chip_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -update_stage true
report_scan_compression_configuration
report_serialize_configuration
create_test_protocol
dft_drc
preview_dft

```

```
insert_dft
write -hierarchy -output TOP.v -format verilog
write_test_protocol -output SERIAL.spf -test_mode ScanCompression_mode
write_test_protocol -output SCAN.spf -test_mode Internal_scan
```

For the Hybrid flow, you set the numbers by using the `set_scan_configuration -chain_count number` and the `set_serialize_configuration -inputs number -outputs number` commands at the top level as follows:

The `set_serialize_configuration -inputs number -outputs number` command specifies the number of scan-in and scan-out ports of the serializer codec used to take care of the top-level user-defined logic. The `set_scan_configuration -chain_count number` command specifies the total number of the top-level scan ports. The number specified with the `set_scan_configuration -chain_count number` command at the top level has to be exactly the same number as the sum of the numbers specified with the `set_serialize_configuration -inputs number -outputs number` command for each serializer codec.

In [Example 131](#), you have one scan-in and one scan-out port for core1 and two scan-in and two scan-out ports for core2. Then, because you have one scan-in and one scan-out port for the serializer that takes care of the top-level user-defined logic, you have to specify four scan-in and scan-out ports with the `set_scan_configuration -chain_count number` command at the top level. This configuration ends up with four scan-in and four scan-out ports created for both serial and standard scan mode.

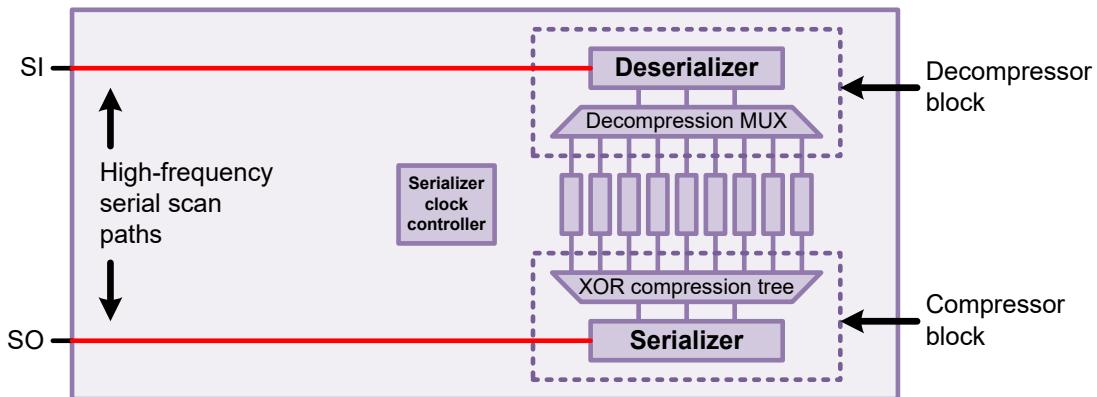
Serializer Chains Concatenated Across Cores

Serializer scan chain concatenation is not supported in the Hybrid flow.

Serializer IP Insertion

When serialized compressed scan is inserted, the tool places the deserializer register inside the decompression MUX block, and it places the serializer register inside the XOR compression tree block. This architecture, shown in [Figure 390](#), keeps the scan compression logic together and minimizes the top-level routing requirements.

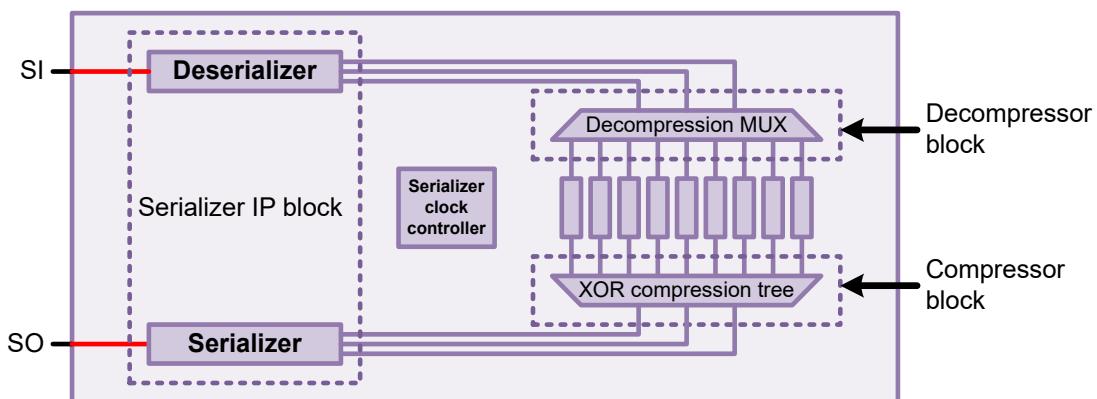
Figure 390 Default Serialized Compressed Scan Architecture



However, depending on layout characteristics, the long routes from the top-level scan I/O connections to the serialization logic can reduce the maximum operating frequency for these serial scan paths.

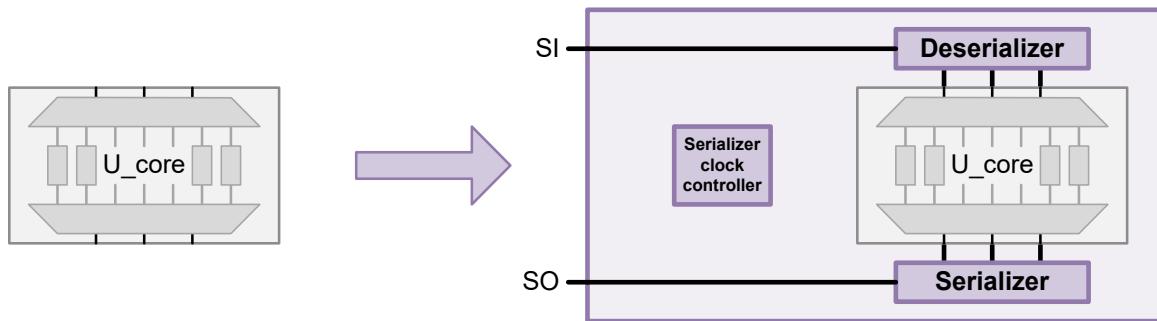
DFTMAX compression provides a feature called *serializer IP insertion* that separates the deserializer and serializer registers, collectively known as the *serializer IP*, from the combinational codec logic. This architecture, shown in Figure 391, places the serializer IP logic separately so that the layout characteristics of the high-frequency serial scan paths can be improved.

Figure 391 Serializer IP Insertion Architecture



Serializer IP insertion can also be applied during HASS and Hybrid integration of cores that already have combinational compressed scan inserted, as shown in Figure 392. The tool inserts and connects the deserializer registers to the existing core-level decompressors, and inserts and connects the serializer registers to the existing core-level compressors. In addition to improving layout characteristics, this flow can be used to reduce the scan pin requirements for existing compressed scan cores.

Figure 392 Inserting Serializer IP Around Combinational Compressed Scan Core



The following topics provide more information about serializer IP insertion:

- [Configuring Serializer IP Insertion](#)
- [Serializer IP Insertion in the Top-Down Flat Flow](#)
- [Serializer IP Insertion in the Top-Down Flat Flow With Partitions](#)
- [Serializer IP Insertion in the HASS Flow](#)
- [Serializer IP Insertion in the Hybrid Flow](#)
- [Serializer IP Insertion and Standard Scan Chains](#)
- [Limitations](#)

Configuring Serializer IP Insertion

To insert serializer IP into a design or partition or around a compressed scan core, use the `-ip_inputs` and `-ip_outputs` options of the `set_serialize_configuration` command:

```
set_serialize_configuration \
    -ip_inputs {object_name n object_name n ...} \
    -ip_outputs {object_name n object_name n ...}
```

These options specify the number of top-level scan ports allocated to the serializer IP for each object. They replace the `-inputs` and `-outputs` options used for normal serializer insertion. The `object_name` argument can be a design name, partition name, or compressed scan core instance name. The input and output port counts must be equal for each object. If multiple objects exist in the design, multiple object name and port count pairs can be specified.

If an update stage register is enabled with the `-update_stage true` option of the `set_serialize_configuration` command, the tool includes the update stage register in the serializer IP block.

[Example 132](#) shows the commands used to identify the core shown in [Figure 392](#) on page 861.

Example 132 Specifying Serializer IP Insertion for Compressed Scan Cores

```
set_serialize_configuration \
    -ip_inputs {U_core_1 1} \
    -ip_outputs {U_core_1 1}
```

You can use the `set_dft_location` command to specify the insertion locations for the serializer IP logic and the codec logic:

```
set_dft_location -include {SERIAL_REG} ser_IP_instance_name
set_dft_location -include {DFTMAX} codec_instance_name
```

If the specified hierarchy level does not exist, the tool creates it during DFT insertion. For more information, see [Specifying a Location for DFT Logic Insertion](#) on page 280.

When serializer IP insertion is enabled, the `preview_dft` and `insert_dft` commands issue messages about each serializer IP block:

```
Inserting Load Deserializer IP
    top_U_core_1_U_deserializer_ScanCompression_mode
        for mode ScanCompression_mode
    Number of inputs = 1
    Maximum size per input = 3
Inserting Unload Serializer IP
    top_U_core_1_U_serializer_ScanCompression_mode
        for mode ScanCompression_mode
    Number of outputs = 1
    Maximum size per output = 3
```

Serializer IP Insertion in the Top-Down Flat Flow

In the top-down flat flow, the tool inserts the combinational codec and the serializer IP at the same time. The script must define both the codec characteristics and the serializer characteristics.

Since the serializer IP is inserted in the top-level design, the design name is specified using the `-ip_inputs` and `-ip_outputs` options. In [Example 133](#), serializer IP insertion is performed for a top-level design named `top_design`.

Example 133 Inserting Serializer IP in a Top-Down Flat Flow

```
current_design top_design

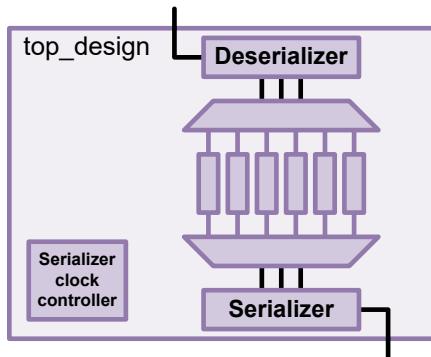
set_scan_configuration \
    -chain_count 1 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable

set_scan_compression_configuration \
```

```
-chain_count 6 \
-inputs 3 -outputs 3 \
-serialize chip_level
set_serialize_configuration \
-ip_inputs {top_design 1} \
-ip_outputs {top_design 1}
```

[Figure 393](#) shows the resulting logic from [Example 133](#).

Figure 393 Serializer IP Insertion in the Top-Down Flat Flow



In this example, one scan-in port and one scan-out port are created for the serial mode. The chain count value of one results in the same scan port count in both the standard scan mode and the serial mode.

Serializer IP Insertion in the Top-Down Flat Flow With Partitions

In the top-down flat flow with partitions, the combinational codec and the serializer IP are inserted at the same time. The script must define both the codec characteristics and the serializer characteristics on a per-partition basis.

Since the serializer IP is inserted inside partitions, the partition names are specified using the `-ip_inputs` and `-ip_outputs` options. In [Example 134](#), serializer IP insertion is performed for two partitions, a user-defined partition and the default partition.

Example 134 Inserting Serializer IP in a Top-Down Flat Flow With Partitions

```
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
-serialize chip_level

# partition1
define_dft_partition partition1 -include {TOP_UDL_1 TOP_UDL_2}
current_dft_partition partition1
set_scan_configuration \
-chain_count 1 -clock_mixing mix_clocks
set_scan_compression_configuration \
```

```

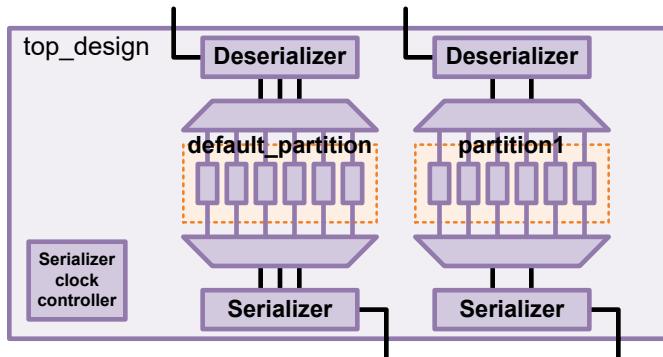
-chain_count 6 \
-inputs 2 -outputs 2
set_serialize_configuration \
-ip_inputs {partition1 1} -ip_outputs {partition1 1}

# default partition
current_dft_partition default_partition
set_scan_configuration \
-chain_count 1 -clock_mixing mix_clocks
set_scan_compression_configuration \
-chain_count 6 \
-inputs 3 -outputs 3
set_serialize_configuration \
-ip_inputs {default_partition 1}
-ip_outputs {default_partition 1}

```

Figure 394 shows the resulting logic from Example 134.

Figure 394 Serializer IP Insertion in the Top-Down Flat Flow With Partitions



In this example, the tool creates one scan-in port and one scan-out port for each partition, for a total of two scan ports used in the serialized scan and standard scan modes.

Serializer IP Insertion in the HASS Flow

In the HASS integration flow, you can add serializer IPs around one or more compressed scan cores and perform top-level core integration. The prerequisites for inserting serializer IP around existing compressed scan cores are as follows:

- The core must have the same number of scan ports in standard scan mode and compressed scan mode.

```

set_scan_configuration \
-chain_count N
set_scan_compression_configuration \
-inputs N -outputs N

```

- The core must have symmetrical codec scan I/O ports, where the values provided to the `-inputs` and `-outputs` options of the `set_scan_compression_configuration` command are equal.
- All scan chains in the core are compressed by a codec.
- No other test modes are defined except `Internal_scan` and `ScanCompression_mode`.
- No pipelined scan data is implemented inside the core.
- The core must be provided as a `.ddc` file or a test model file, so that test model information is available for the scan compression logic.

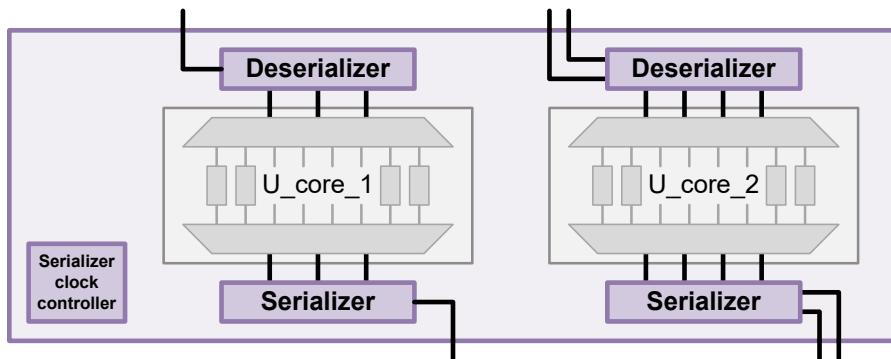
In [Example 135](#), serializer IP insertion is performed for core instances `U_core_1` and `U_core_2`.

Example 135 Inserting Serializer IP Around Cores in a HASS Flow

```
set_dft_configuration -scan_compression enable
set_scan_configuration -chain_count 3 -clock_mixing mix_clocks
set_scan_compression_configuration \
    -integration_only true \
    -serialize chip_level
set_serialize_configuration \
    -ip_inputs {U_core_1 1 U_core_2 2} \
    -ip_outputs {U_core_1 1 U_core_2 2}
```

[Figure 395](#) shows the resulting logic from [Example 135](#).

Figure 395 Serializer IP Insertion in the HASS Flow



In this example, the tool creates three scan-in ports and three scan-out ports for the serial scan mode, one pair for `U_core_1` and two pairs for `U_core_2`. The number specified with the `set_scan_configuration -chain_count` command at the top level for the standard scan mode should be large enough to satisfy the total number of serial mode scan ports across all cores.

Referencing Multiple Codecs in Compressed Scan Cores

When a compressed scan core contains multiple codecs, you must specify the serializer IP characteristics for each codec. When you specify serializer IP characteristics with the `-ip_inputs` and `-ip_outputs` options, follow the compressed scan core cell name with the name of a decompressor or compressor, respectively, inside the core. You must provide a separate entry for each codec inside the core.

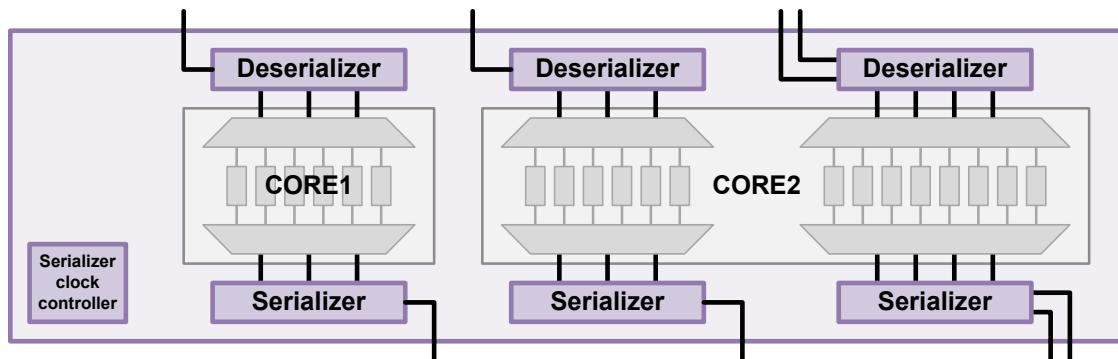
[Example 136](#) specifies serializer IP characteristics for a single codec in a compressed scan core named CORE1 and two codecs in a compressed scan core named CORE2.

Example 136 Serializer IP Insertion for Multiple Codecs in a Compressed Scan Core

```
set_serialize_configuration \
    -ip_inputs {CORE1 1 \
        CORE2 core2_P1_U_decompressor_ScanCompression_mode 1 \
        CORE2 core2_P2_U_decompressor_ScanCompression_mode 2} \
    -ip_outputs {CORE1 1 \
        CORE2 core2_P1_U_compressor_ScanCompression_mode 1 \
        CORE2 core2_P2_U_compressor_ScanCompression_mode 2}
```

Figure 396 shows the resulting logic from [Example 136](#).

Figure 396 Serializer IP Insertion for Multiple Codecs in a Compressed Scan Core



You can obtain the decompressor and compressor names using one of the following methods:

- Use the `list_test_models -compressors` command at the top level before DFT insertion.
- Look in the `CompressorStructures` section of a core-level ASCII CTL model file.
- Look in the `CompressorStructures` section of a core-level STIL protocol file that is generated for the scan compression mode.

[Example 137](#) shows a report example from the `list_test_models -compressors` command, run at the top level before DFT insertion.

Example 137 Report Example From the `list_test_models -compressors` Command

```
dc_shell> list_test_models -compressors
core1                               /home/user/core1.db
  Codecs:
    core1_U_decompressor_ScanCompression_mode
    core1_U_decompressor_ScanCompression_mode

core2                               /home/user/core2.db
  Codecs:
    core2_P1_U_decompressor_ScanCompression_mode
    core2_P2_U_decompressor_ScanCompression_mode
    core2_P1_U_compressor_ScanCompression_mode
    core2_P2_U_compressor_ScanCompression_mode

top                                 /home/user/top.db
  Codecs:
```

The report from the `list_test_models -compressors` command shows the list of designs with CTL model information, along with the codec names defined in each CTL model. However, the `-ip_inputs` and `-ip_outputs` options require core instance names, not design names. To convert a design name to a list of instances, use the `get_references` command. For example,

```
dc_shell> get_references core2
{CORE2}
```

[Example 138](#) shows how the decompressor and compressor names are provided in an `example CompressorStructures` block, contained in an ASCII CTL model file or STIL protocol file.

Example 138 Decompressor and Compressor Names in a CompressorStructures Block

```
CompressorStructures {
  Compressor "core2_P1_U_decompressor_ScanCompression_mode" {
    ...
  }
  Compressor "core2_P2_U_decompressor_ScanCompression_mode" {
    ...
  }
  Compressor "core2_P1_U_compressor_ScanCompression_mode" {
    ...
  }
  Compressor "core2_P2_U_compressor_ScanCompression_mode" {
    ...
  }
}
```

Serializer IP Insertion in the Hybrid Flow

The Hybrid flow is an extension of the HASS flow that applies scan compression to any uncompressed top-level logic. The requirements and features of serializer IP insertion in the HASS flow also apply to the Hybrid flow. See [Serializer IP Insertion in the HASS Flow on page 864](#).

[Example 139](#) shows a typical Hybrid flow that inserts serializer IPs around existing compressed scan cores, inserts an additional serializer and codec into the top-level user-defined logic, and then integrates all the structures.

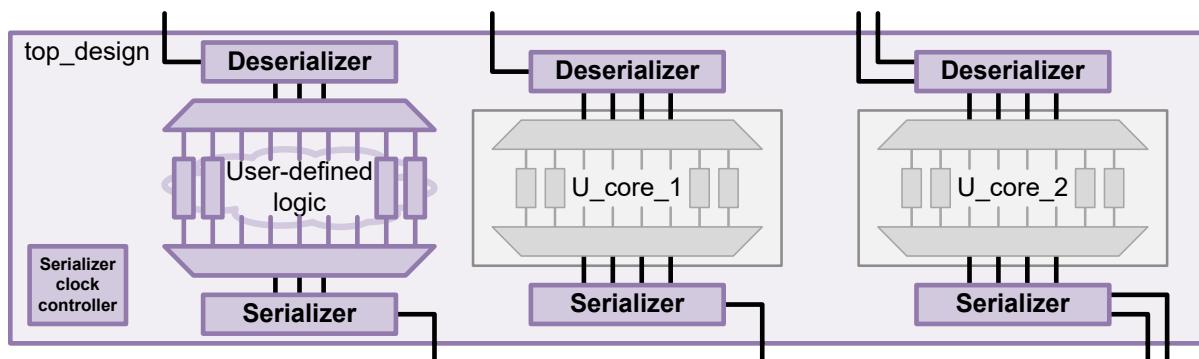
Example 139 Inserting Serializer IP Around Cores in the Hybrid Flow

```
current_design top_design

set_dft_configuration -scan_compression enable
set_scan_configuration \
    -chain_count 4 -clock_mixing mix_clocks
set_scan_compression_configuration \
    -chain_count 6 \
    -inputs 3 -outputs 3 \
    -hybrid true \
    -serialize_chip_level
set_serialize_configuration \
    -ip_inputs {top_design 1 U_core_1 1 U_core_2 2} \
    -ip_outputs {top_design 1 U_core_1 1 U_core_2 2}
```

[Figure 397](#) shows the resulting logic from [Example 139](#).

Figure 397 Serializer IP Insertion in the Hybrid Flow



In this example, the script assigns a single scan-in and scan-out port to the top-level serializer IP and codec by referencing the top-level design name with the `-ip_inputs` and `-ip_outputs` options. The tool creates one scan-in and scan-out port for the serialized IP inserted around `U_core_1`, and it creates two scan-in and scan-out ports for the serialized IP inserted around `U_core_2`. There are a total of four scan connections in the serialized scan mode.

The number specified with the `set_scan_configuration -chain_count` command at the top level for the standard scan mode should be large enough to satisfy the total number of serial mode scan ports across all cores. In this example, the chain count value of four results in four scan-in and scan-out ports in both the standard scan mode and the serial mode.

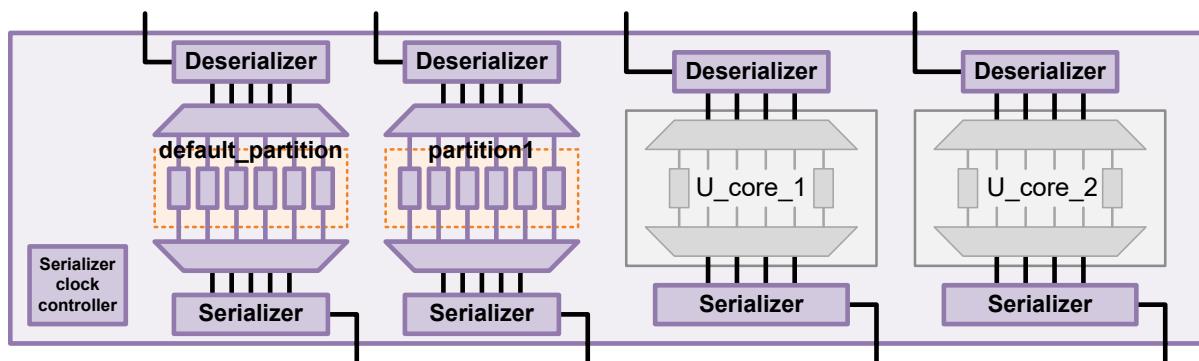
Serializer IP Insertion in the Hybrid Flow With Top-Level Partitions

If you need to integrate combinational compressed scan cores but also have a great deal of user-defined logic at the top level, you might want to use the Hybrid flow with top-level partitions. This flow allows you to divide the user-defined logic into multiple partitions, each of which has a serializer codec.

Consider the scenario shown in [Figure 398](#).

- You have two compressed scan cores, `U_core_1` and `U_core_2`.
- You want to distribute the top-level user-defined logic among two partitions, `partition1` and the default partition.
- You want to insert serializer IP around each of the two compressed scan cores, with one top-level serial scan-in and scan-out port for each core.
- You want to insert serialized compressed scan for each of the two top-level partitions, with one top-level serial scan-in and scan-out port for each partition.
- You want to use four existing scan-in and four scan-out ports at the top level in both standard scan mode and serial mode.

Figure 398 Serializer IP Insertion in the Hybrid Flow With Top-Level Partitions



[Example 140](#) shows a Hybrid flow with top-level partitions script for this scenario.

Example 140 Script for a Serializer Hybrid Flow With Top-Level Partitions

```
# global settings
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
```

```

-hybrid true \
-serialize chip_level

# partition1
define_dft_partition partition1 -include {TOP_UDL_1 TOP_UDL_2}
current_dft_partition partition1
set_scan_configuration \
    -chain_count 1 -clock_mixing mix_clocks
set_scan_compression_configuration \
    -chain_count 6 \
    -inputs 5 -outputs 5
set_serialize_configuration \
    -ip_inputs {partition1 1} -ip_outputs {partition1 1}

set_dft_signal -view spec -type ScanDataIn -test_mode all \
    -port {SI_PART1_0}
set_dft_signal -view spec -type ScanDataOut -test_mode all \
    -port {SO_PART1_0}

# default partition
current_dft_partition default_partition
set_scan_configuration \
    -chain_count 3 -clock_mixing mix_clocks
set_scan_compression_configuration \
    -chain_count 6 \
    -inputs 5 -outputs 5
set_serialize_configuration \
    -ip_inputs {default_partition 1 U_core_1 1 U_core_2 1} \
    -ip_outputs {default_partition 1 U_core_1 1 U_core_2 1}

set_dft_signal -view spec -type ScanDataIn -test_mode all \
    -port {SI_PARTD_0 SI_PARTD_1 SI_PARTD_2}
set_dft_signal -view spec -type ScanDataOut -test_mode all \
    -port {SO_PARTD_0 SO_PARTD_1 SO_PARTD_2}

create_test_protocol
dft_drc
insert_dft

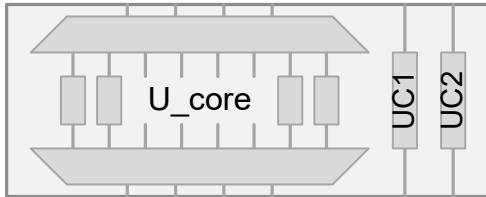
```

In this flow, the compressed scan cores must exist in the default partition. As a result, they are configured by the `set_serialize_configuration` command applied to the default partition. The chain count applied to the default partition with the `set_scan_configuration -chain_count` command includes both the compressed scan cores and the new top-level codec.

Incorporating External Chains Into the Hybrid Serializer IP Flows

You might have compressed scan cores that contain one or more uncompressed external scan chains, as shown in [Figure 399](#).

Figure 399 External Uncompressed Chains in a Compressed Scan Core



These uncompressed external scan chains are specified at the core level with the `set_scan_path` command, as shown in [Example 141](#).

Example 141 Defining External Uncompressed Chains at the Core Level

```
set_scan_path UC1 \
    -view spec -test_mode all_dft \
    -complete true -dedicated_scan_out true \
    -scan_data_in SI_0 -scan_data_out SO_0 \
    -ordered_elements {...}
set_scan_path UC2 \
    -view spec -test_mode all_dft \
    -complete true -dedicated_scan_out true \
    -scan_data_in SI_1 -scan_data_out SO_1 \
    -ordered_elements {...}
set_scan_configuration \
    -chain_count 5 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -chain_count 8 \
    -inputs 4 -outputs 4
create_test_protocol
dft_drc
preview_dft -show scan
insert_dft
```

In [Example 142](#), the Hybrid flow is used to insert serializer IP around a compressed scan core, and to insert a full serialized codec around the top-level user-defined logic. The tool includes the external chains in `U_core` as part of the user-defined logic.

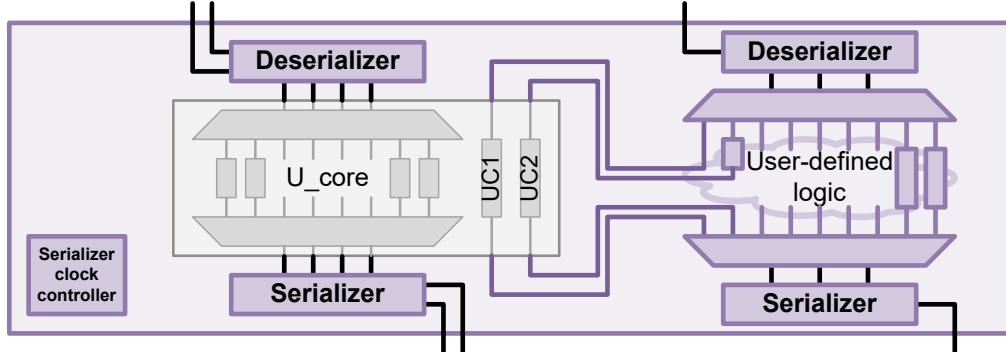
Example 142 Incorporating External Chains Into Serializer IP Hybrid Integration

```
set_scan_configuration \
    -chain_count 3 -clock_mixing mix_clocks
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -hybrid true \
    -serialize chip_level \
    -chain_count 9 \
    -inputs 3 -outputs 3
set_serialize_configuration \
```

```
-ip_inputs [U_core 2 top_design_name 1] \
-ip_outputs [U_core 2 top_design_name 1]
```

[Figure 400](#) shows the resulting logic from [Example 142](#).

Figure 400 Serializer IP Insertion in the Hybrid Flow With External Chains

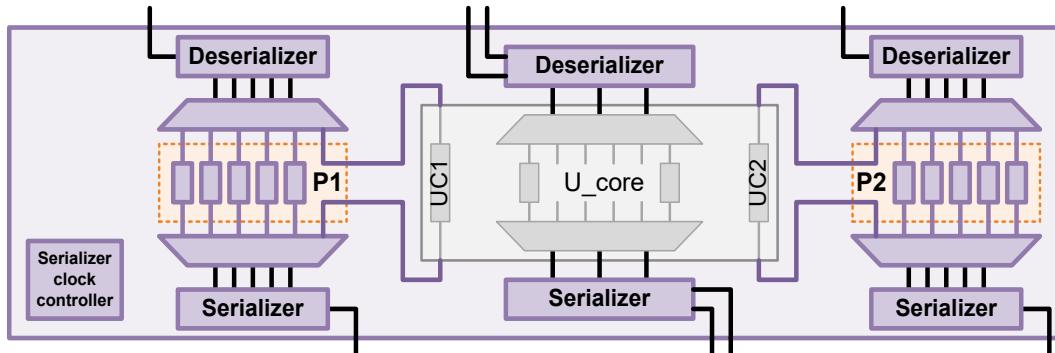


If you are inserting serializer IP using the Hybrid flow with partitions, all external chains are placed in the default partition by default. If you want to incorporate an external chain into a different partition, you can include scan chain names in the partition definitions:

```
define_dft_partition P1 -include {top_UDL1 U_core/UC1}
define_dft_partition P2 -include {top_UDL2 U_core/UC2}
```

[Figure 401](#) shows how these commands allocate the external chains between the two partitions.

Figure 401 Allocating External Chains to Partitions in the Hybrid Serializer IP Flow



Note:

Scan chain names are only supported in partition definitions when this flow is being used.

See Also

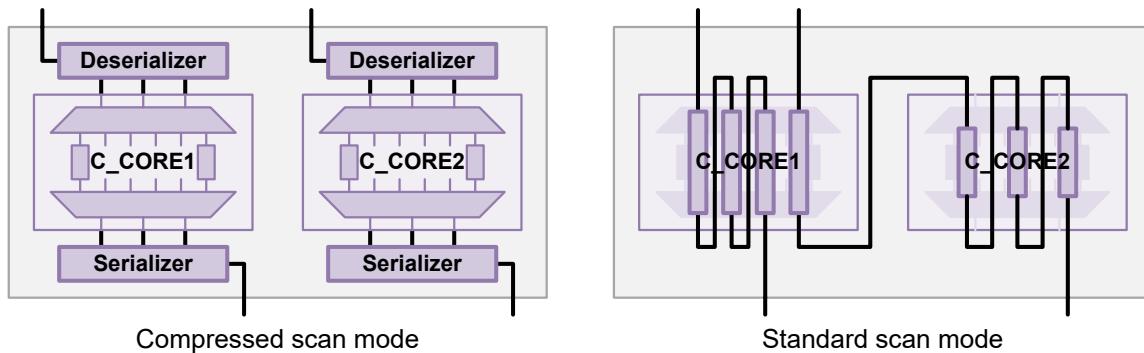
- [Excluding Scan Chains From Scan Compression on page 686](#) for more information about external chains

Serializer IP Insertion and Standard Scan Chains

When compressed scan cores are wrapped with serializer IP logic in HASS and Hybrid integration flows, the standard scan mode is also affected.

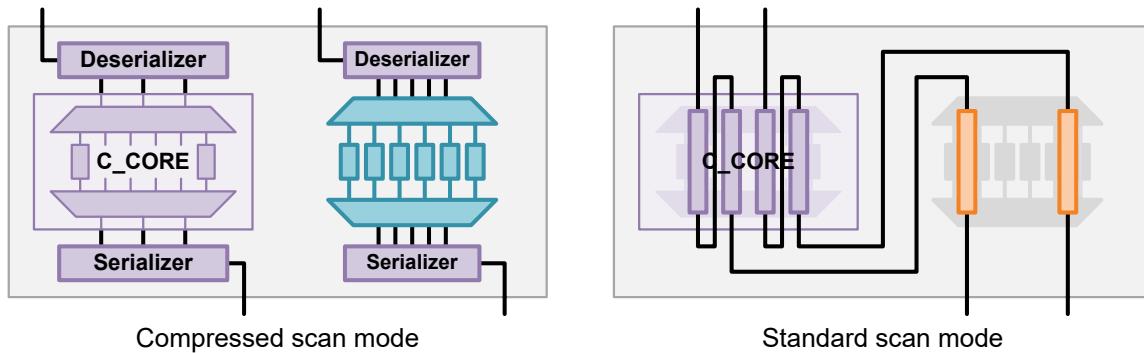
Due to the reduced number of available top-level scan I/O pins, the standard scan chains inside the compressed cores can no longer be promoted to dedicated top-level connections. To remedy this, standard scan chains in compressed scan cores become scan segments that can be concatenated, if needed, by top-level integration. [Figure 402](#) shows the compressed scan and standard scan chains for a design in the HASS serializer IP insertion flow.

Figure 402 Standard Scan Chains in the HASS Serializer IP Insertion Flow



In the Hybrid flow with serializer IP insertion, core-level scan segments can be mixed with top-level scan cells to achieve optimal balancing. [Figure 403](#) shows the compressed scan and standard scan chains for a design in the Hybrid serializer IP insertion flow.

Figure 403 Standard Scan Chains in the Hybrid Serializer IP Insertion Flow



Standard scan chain concatenation is not needed for cores that are already serialized because such a core's standard scan mode is architected to use the same scan I/O resources as its serialized compressed scan mode.

Limitations

The serializer IP insertion feature has the following limitations:

- For each combinational compressed scan core, the number of scan inputs and scan outputs must be the same for the standard scan mode and the compressed scan mode.
- All cores must have the same X-tolerance type. A mix of default X-tolerance and high X-tolerance is not supported.
- Multiple user-defined compressed scan test modes are not supported at the core level or the top level.
- Pipeline scan data registers are not supported at the core level.
- A mix of combinational compressed scan cores and serialized compressed scan cores is not supported.
- Serializer chains cannot be concatenated across cores at the top level.
- There is no support for core-specific serializer IP insertion.
- The number of serializer scan ports specified with the `-ip_inputs` and `-ip_outputs` options must be less than the number of combinational codec inputs and outputs, respectively.

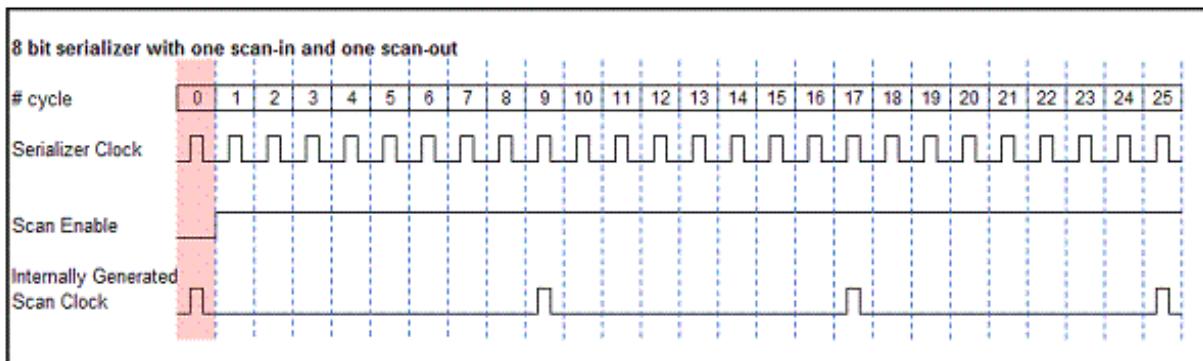
- Lock-up latch insertion is not supported between the serializer IP and core scan cells.
 - If scan clocks exist inside the compressed scan core that differ from the serializer register clock, you should insert lock-up latches inside the compressed scan core. Insert them between the decompressor outputs and first scan elements and between the last scan elements and the compressor inputs. This must be done manually. The tool cannot modify DFT-inserted cores during serializer IP insertion.

Wide Duty Cycle Support for Serializer

By default, the internally generated scan shift clocks and the update stage clock are created by gating external clocks with enable signals generated by the serializer FSM counter. The enable signals go active every S cycles, where S is the length of the serializer register segment. Therefore, the internally generated scan shift clocks and the update stage clock are inactive for (S-1) external clock cycles and pulse only at cycle S.

When you have an 8-bit serializer with one scan-in port and one scan-out port, the length of the serializer segment is 8. When the external clock has a 10 percent duty cycle (for example, rise = 45ns, fall = 55ns, period = 100ns), the clock duty cycle of the internally generated clock is 1.25 percent. [Figure 404](#) illustrates this scenario.

Figure 404 Timing Diagram With Default Duty Cycle



With a narrow pulse width, two issues exist:

- The clock skew for internal clocks might be more than the pulse width, thereby leading to problems in shift.
- Because of rise and fall slew, the clock might not reach its logic level completely, and the clock waveform might be clipped.

Wide duty cycle support makes the clock duty cycle close to 50 percent, resolving these problems.

To enable the wide duty cycle support feature, use the following option setting:

```
set_serialize_configuration -wide_duty_cycle true
```

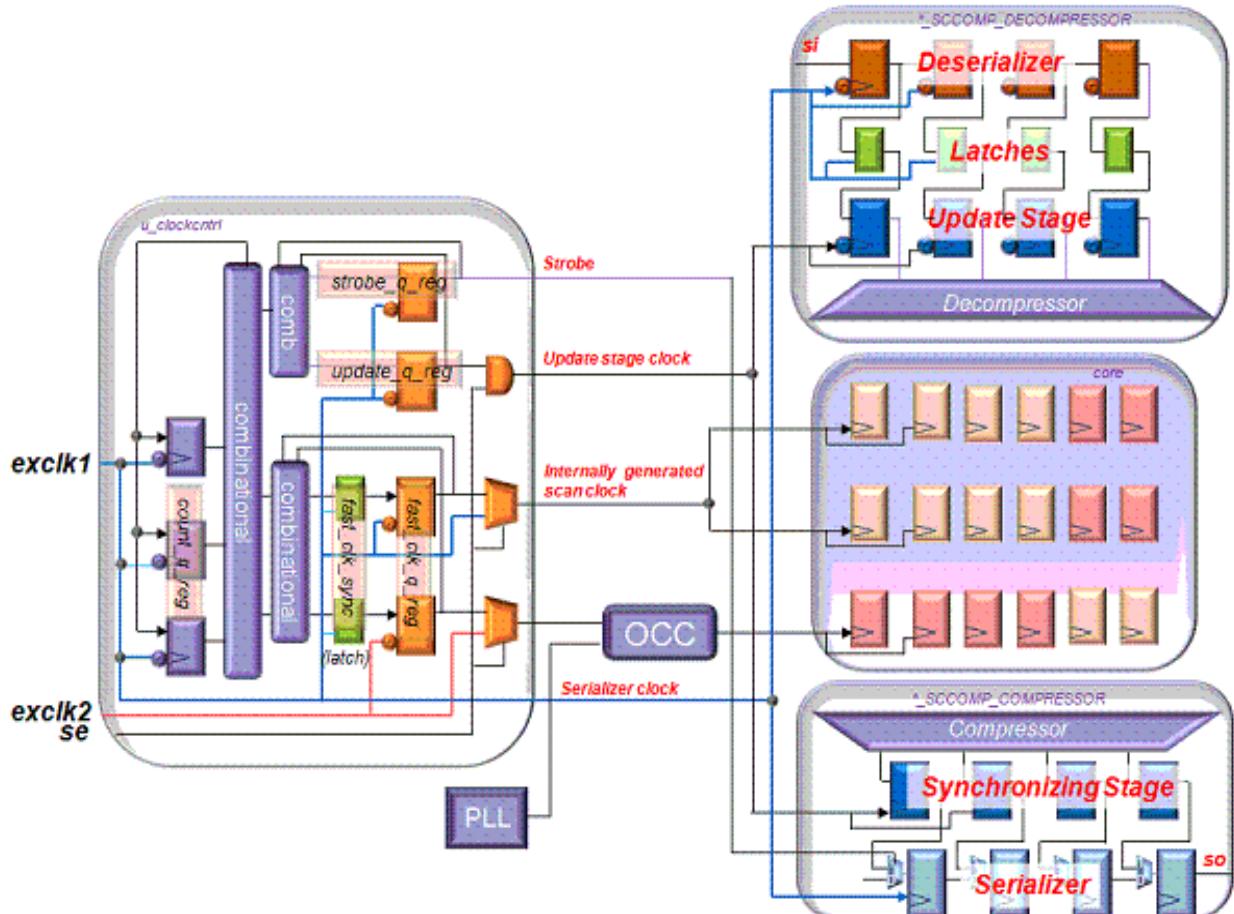
After the option is accepted, the `preview_dft` command shows the following information message:

Information: Implementing Wide Duty Cycle Serializer Clock Controller.

Block Diagram

[Figure 405](#) shows the block diagram when you implement the wide duty cycle.

[Figure 405 Block Diagram With Wide Duty Cycle Support](#)



When the `-wide_duty_cycle true` option is specified, the tool uses the following behaviors:

- The update stage is always inserted in the decompressor IP, and the following information message appears:

Information: the update stage will be enabled in presence of wide duty cycle.

- Lock-up latches are always inserted in the decompressor IP between the deserializer registers and the update stage registers. Note that lock-up latch insertion is also available when the update stage is inserted without the wide duty cycle support enabled by using the following variable:

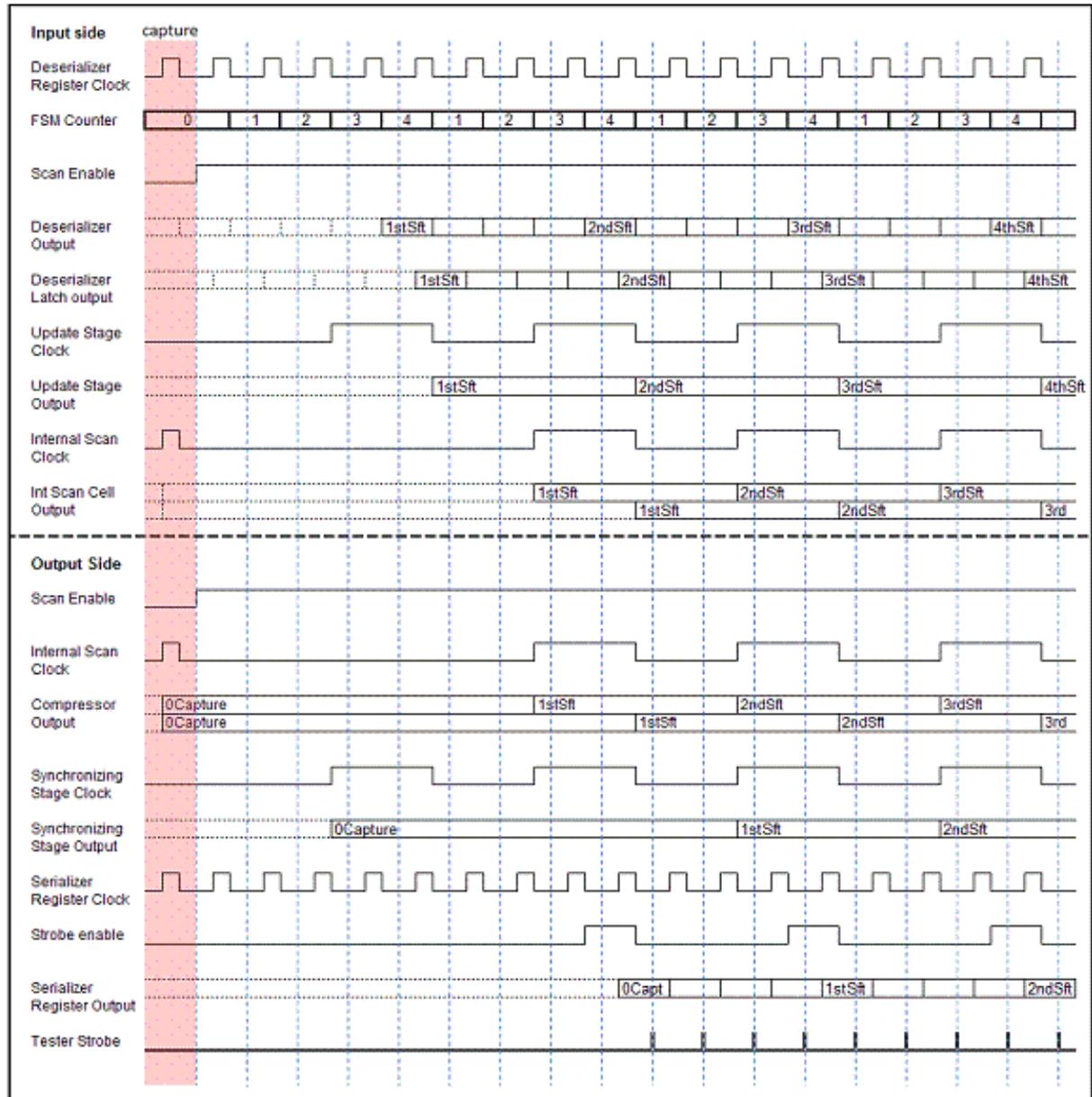
```
set_app_var test_elpc_lul_in_deserializer true
```

- A synchronizing stage is inserted in the compressor IP. The synchronizing stage registers help hold the scan-out data until serializer registers capture it. The clock of the synchronizing stage registers is the same as the clock of the update stage registers inserted in the decompressor IP.
- No clock-gating logic is used in the serializer clock controller.

Timing Diagram

[Figure 406](#) shows the timing diagram for a 4-bit serializer, indicating how it behaves.

Figure 406 Timing Diagram With Wide Duty Cycle Support

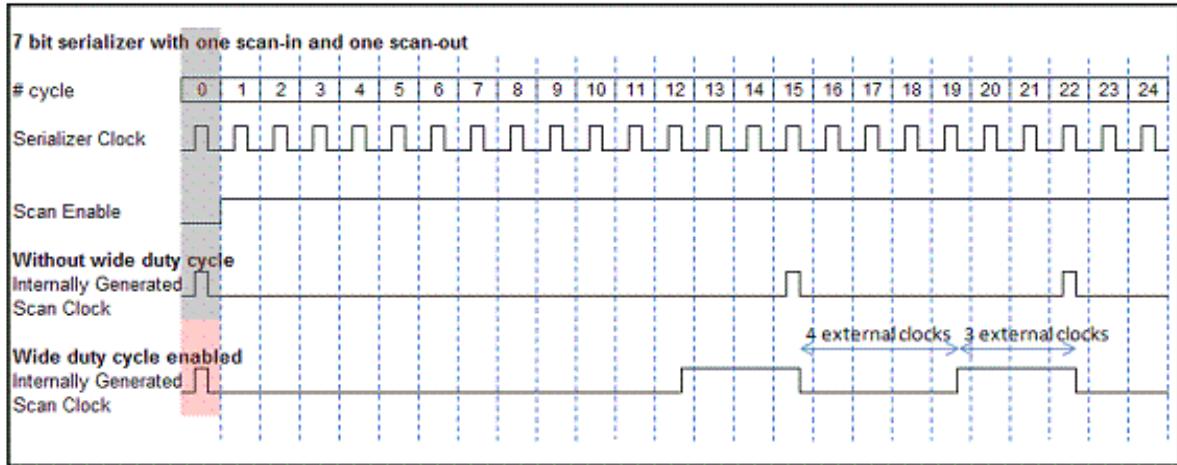


Internally Generated Clocks

The following examples show how the internally generated scan shift clocks are created when wide duty cycle support is enabled.

When you have a 7-bit serializer with one scan-in port and one scan-out port, the length of the serializer segment is 7. With the wide duty cycle enabled, the clock is on for 3 external clock cycles and off for 4 external clock cycles, as illustrated in the [Figure 407](#).

Figure 407 Timing Diagram of Default Duty Cycle and Wide Duty Cycle Clock



[Table 55](#) provides a table that shows the clock width based on the length of the serializer segment.

Serializer segment length	# of external clock cycles for clock ON (OFF)
2	1 (1)
3	1 (2)
4	2 (2)
5	2 (3)
6	3 (3)
7	3 (4)
8	4 (4)
9	4 (5)
10	5 (5)

Wide Duty Cycle in a Core-Level Flow

When you use the wide duty cycle feature, you must set the `-wide_duty_cycle` option to `true` at each core creation. This setting allows the tool to insert the update stage and the synchronization stage into the decompressor and the compressor, respectively.

Use the following command example for the core-level flow:

```
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 400 \
    -inputs 8 \
    -outputs 8 \
    -serialize core_level
set_serialize_configuration \
    -inputs 1 \
    -outputs 1 \
    -wide_duty_cycle true
```

Wide Duty Cycle in the HASS Flow

In the HASS flow, use the following command example:

```
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -integration_only true \
    -serialize chip_level
set_serialize_configuration \
    -wide_duty_cycle true
```

Note the following when you use the wide duty cycle feature in the HASS flow:

- All serializer cores must be implemented with the `-wide_duty_cycle true` option.
- The option `-wide_duty_cycle true` must also be set at the top level.

Wide Duty Cycle in the Hybrid Flow

In the Hybrid flow, use the following command example:

```
set_dft_configuration -scan_compression enable
set_scan_compression_configuration \
    -xtolerance high \
    -chain_count 400 \
    -inputs 1 \
    -outputs 1 \
    -hybrid true \
    -serialize chip_level
set_serialize_configuration \
```

```
-inputs 1 \
-outputs 1 \
-wide_duty_cycle true
```

Note the following when you use the wide duty cycle feature in the Hybrid flow:

- All serializer cores and the top-level serializer must be implemented with the option setting `-wide_duty_cycle true`.

Dual STIL Flow Parallel Patterns

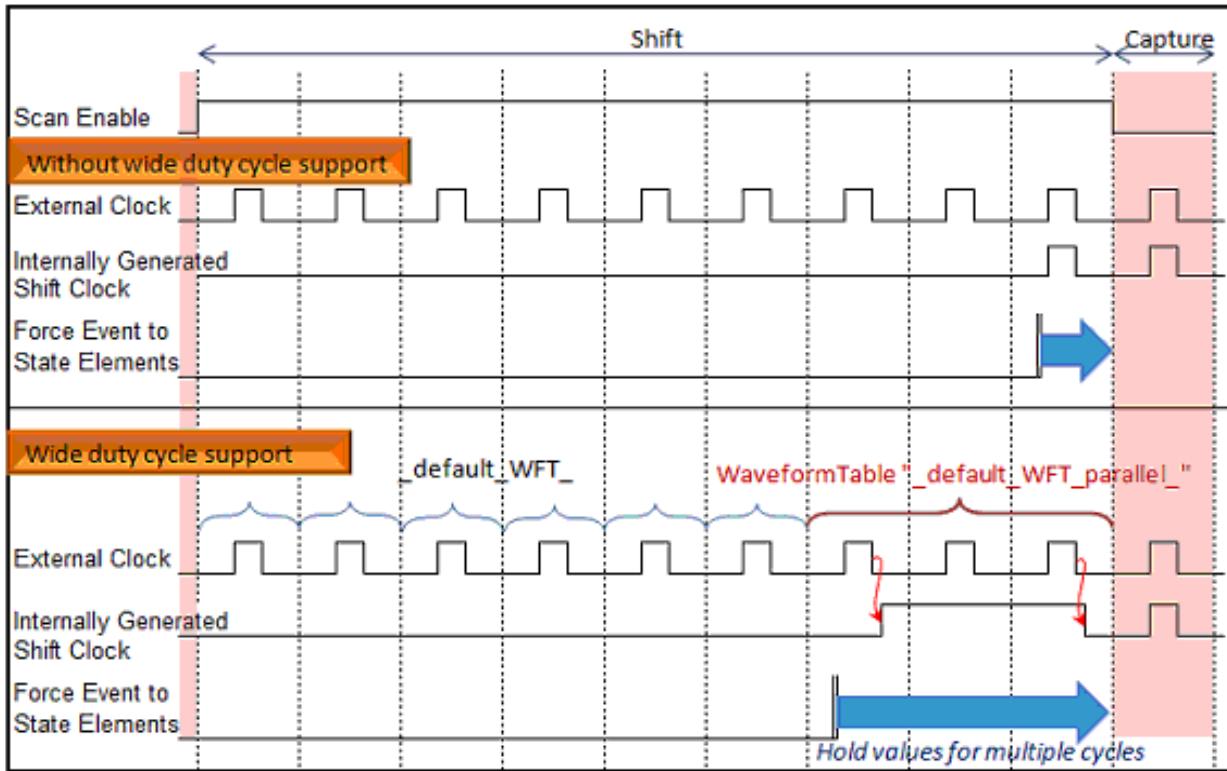
When you use Wide Duty Cycle support and write out parallel patterns with the Dual STIL flow in TestMAX ATPG, the waveform table contains multiple clock pulses as shown in [Example 143](#).

Example 143 Waveform Table _default_WFT_parallel

```
WaveformTable "_default_WFT_parallel_" {
    Period '30ns';
    Waveforms {
        "EXT_CLK1" { P { '0ns' D; '4ns' U; '7ns' D; '14ns' U; '17ns' D;
                      '24ns' U; '27ns' D; } }
        "EXT_CLK2" { P { '0ns' D; '4ns' U; '7ns' D; '14ns' U; '17ns' D;
                      '24ns' U; '27ns' D; } }
        "all_bidirectionals" { 0 { '0ns' D; } }
        "all_bidirectionals" { 1 { '0ns' U; } }
        "all_bidirectionals" { T { '0ns' Z; '3ns' T; } }
        "all_bidirectionals" { X { '0ns' Z; '3ns' X; } }
        "all_bidirectionals" { H { '0ns' Z; '3ns' H; } }
        "all_bidirectionals" { Z { '0ns' Z; } }
        "all_bidirectionals" { L { '0ns' Z; '3ns' L; } }
        "all_bidirectionals" { N { '0ns' N; } }
```

This waveform table holds forced values at the state elements for multiple external clock cycles so that scan cells driven by both the leading and trailing edges of the generated wide duty cycle clocks can capture the values. [Figure 408](#) shows how the waveform is used.

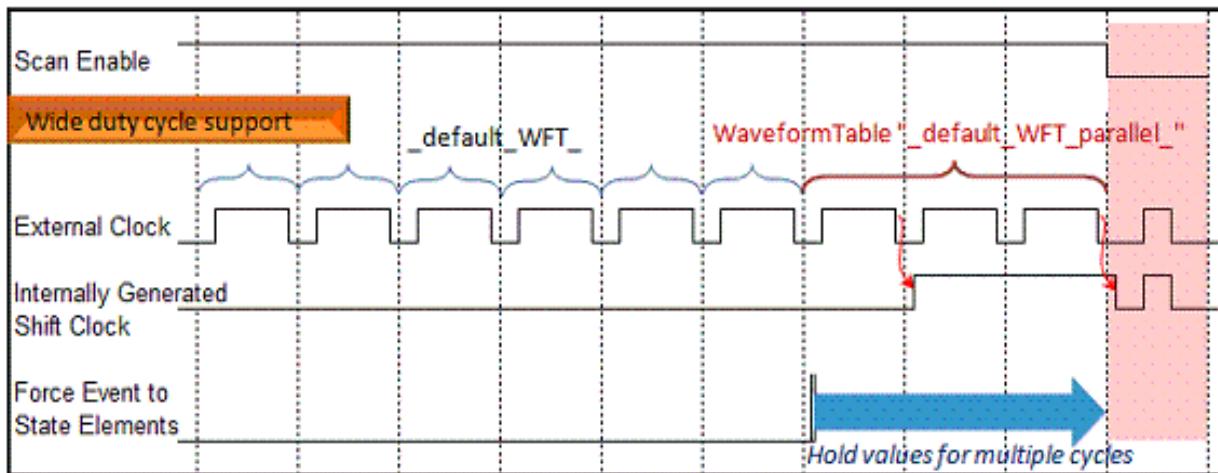
Figure 408 Application of the _default_WFT_parallel Waveform



This waveform example uses a 4-bit serializer. The clock-on for the internally generated wide duty cycle clock is equal to two external clock cycles. The forced value is held across three external clock cycles. Regardless of whether the scan cells are driven by the leading or trailing edge of the generated wide duty cycle clock, the scan cells can capture the forced value.

However, if you make both the external clocks wide and the trailing edge close to the end of the cycle, the trailing edge of the generated wide duty cycle clock might cross into the capture window due to internal delay on the clock line. This potential timing issue is shown in [Figure 409](#).

Figure 409 Potential Timing Issues for the _default_WFT_parallel Waveform



If this timing problem occurs, the shift operation cannot perform properly on scan cells triggered by the trailing edge of the generated wide duty cycle clock. The problem is independent of pattern format. Therefore, you should plan carefully to avoid this problem.

Limitations

Note the following limitation with the wide duty cycle feature:

- Staggered clock is not supported.

Serializer in Conjunction With On-Chip Clocking Controllers

The relationship between serializer clock controllers and on-chip clocking (OCC) controllers is discussed in the following topics.

OCC and SPC Chains in a Serializer Design

When you use on-chip clocking (OCC) controllers or shift power control (SPC) chains in a design with a serializer,

- The head of the OCC/SPC chains is driven directly by a dedicated bit in the deserializer register.
- The tail of the OCC/SPC chains drives the compressor.

Using SPC chains in a design with a serializer without OCC controllers is not supported.

This is the serializer equivalent of the OCC/SPC chain dataflow used by combinational DFTMAX compression (dedicated input, compressed output).

Because the clock chain must be driven by the deserializer register to ensure correct scan operation, you cannot define an external (port-driven) OCC or SPC chain in the serializer flow.

Using Serializer With User-Defined OCC Controllers

By default, the serializer clock controller generated by the TestMAX DFT tool considers the clock pulse of the preamble vector outside the shift procedure to ensure correct operation for the DFT-inserted OCC controller. Existing user-defined OCC controllers that require the preamble clock pulse to enter shift mode are compatible with the default serializer clock controller. However, some user-defined OCC controllers do not require the preamble clock pulse. In this case, specify the following variable before the test protocol generation:

```
dc_shell> set_app_var test_ate_sync_cycles 0
```

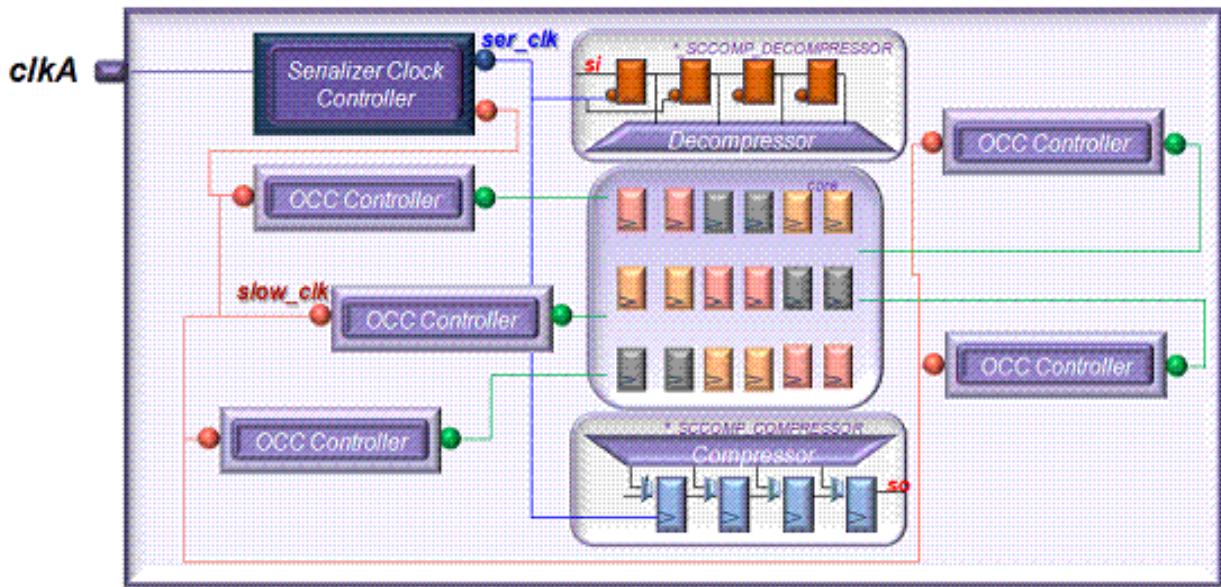
See Also

- [SolvNet article 035708, “What Does the test_ate_sync_cycles Variable Do?”](#) for more information about the test_ate_sync_cycles variable

Using a Serializer Clock Controller With Multiple OCC Controllers

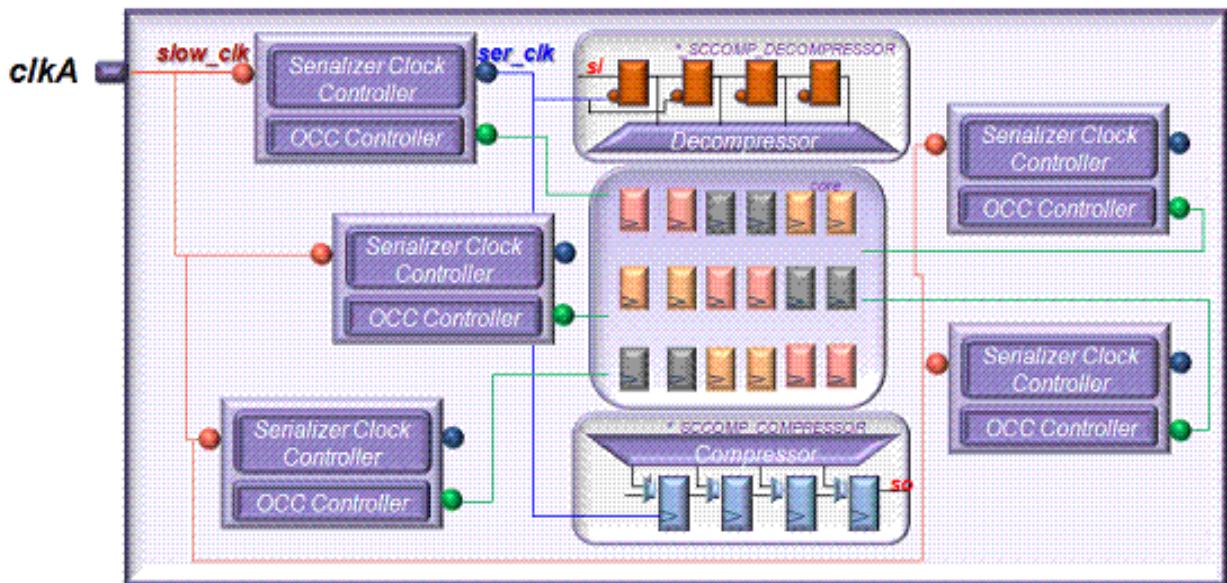
When multiple DFT-inserted OCC controllers are specified with a serializer, a single serializer clock controller is inserted. This single serializer clock controller internally generates a slow clock that connects to the `slow_clock` pin of each OCC controller. This clock and controller structure is shown in [Figure 410](#).

Figure 410 Serializer Clock Controller With Multiple OCC Controllers, Default Architecture



An alternate architecture is also available by setting the `test_elpc_unique_fsm` variable to `false`. In this alternate architecture, a separate serializer clock controller is inserted into each OCC controller, so that the resulting locally-generated slow serializer clock can feed the glitch-free clock selection MUX inside the OCC controller. See Figure 411.

Figure 411 Serializer Clock Controller With Multiple OCC Controllers, Alternate Architecture



This alternate architecture is supported in both the top-down flat flow and the top-down partition flow. This architecture might lead to long wires connecting the external clock port to the `slow_clk` pin of each OCC controller, which can produce congestion and timing issues when the external clock frequency is high.

See Also

- [Chapter 12, On-Chip Clocking Support](#) for more information about OCC controllers

Waveforms for a Serializer With OCC Controllers

If you use a DFT-inserted OCC controller without a serializer, the tool connects the `slow_clk` pin of the OCC controller to an ATE-provided clock, which is one of the clocks specified with the `set_dft_signal -type ScanClock` command. Even in a serializer flow, the OCC controller must be connected to a clock corresponding to a scan shift clock that is actually an internally-generated scan shift clock created by the serializer clock controller.

The two waveform examples shown in [Figure 412](#) and [Figure 413](#) illustrate parallel mode and serial mode behavior. For the serial mode example, a single serializer clock controller is used, as discussed in [Using a Serializer Clock Controller With Multiple OCC Controllers on page 884](#).

You can compare the waveforms directly at the OCC controller pins.

Figure 412 OCC With Parallel Mode

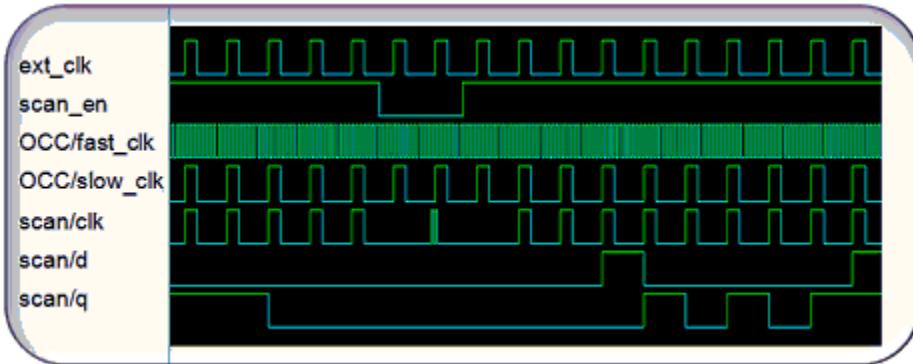
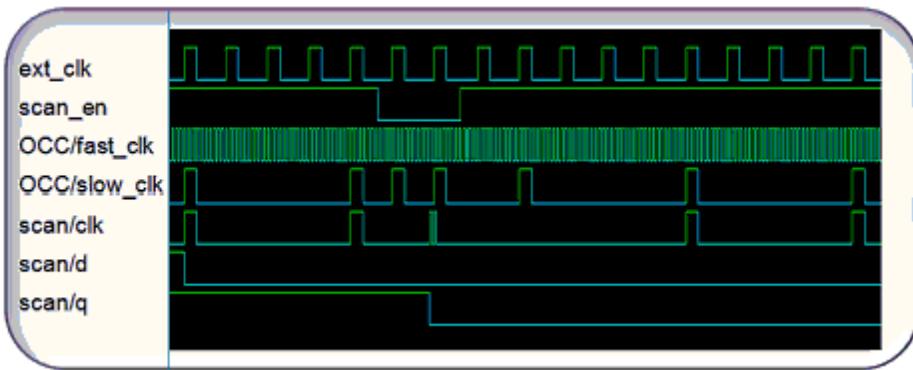


Figure 413 OCC With Serial Mode



As seen in these two figures, the serial mode behavior is the same as the parallel mode behavior during capture, while the scan-enable signal (`scan_en`) is low. For the shift mode, the internally generated clocks need to drive the scan cells. Since this flow uses an inserted OCC controller, the internally generated scan clock drives the OCC controller `slow_clk` input pin. Then, one clock pulse after capture is consumed inside the OCC controller, the output of the OCC controller drives the scan cells.

Using Integrated Clock-Gating Cells in the Serializer Clock Controller

By default, the tool uses discrete cells for clock-gating logic in a serializer clock controller. Using the following variables, you can specify an integrated clock-gating cell library cell reference (without the library name) for the serializer clock gating logic:

```
set_app_var test_icg_p_ref_for_dft library_cell_ref
set_app_var test_icg_n_ref_for_dft library_cell_ref
```

The `test_icg_p_ref_for_dft` variable specifies a library cell to be used to gate return-to-zero clocks. The `test_icg_n_ref_for_dft` variable specifies a library cell to be used to gate return-to-one clocks. The tool automatically inserts the specified integrated clock-gating cells depending on the clock polarity.

User-Defined Pipelined Scan Data

If you implement user-defined pipelined scan data registers by hand, be careful of the driving edge and the timing. The input-side pipelined scan data registers are connected from the scan-in ports to the deserializer registers directly. The deserializer registers are triggered by the trailing edge. Ideally, the pipelined scan data registers should be triggered by the same edge and operate with the same timing to be safe. In the same manner,

the output-side serializer registers should be triggered by the leading edge, so as not to produce a shift error. The same edge and the same timing are recommended.

Running TestMAX ATPG on Serializer Designs

You can perform two types of ATPG with the TestMAX ATPG tool:

- Chip-level ATPG where the serializer clock controller is inserted and the compressed scan chain clock is provided by the serializer clock controller
- Core-level ATPG where both a serializer clock and internal scan shift clocks are provided directly from the primary ports

Chip-level ATPG is performed on designs completed with the top-down flat flow, top-down partition flow, HASS flow, or Hybrid flow. Core-level ATPG is performed on a core implemented with the `set_scan_compression_configuration -serialize core_level` command, which is normally integrated later with other cores by using a HASS or Hybrid flow. Chip-level and core-level test protocols are somewhat different, but the TestMAX ATPG tool identifies them and performs ATPG accordingly without requiring any special commands or guidance.

The following topics are covered in this section:

- [Simulation and Patterns](#)
- [STIL Protocol File](#)
- [Debugging TestMAX ATPG Serializer DRC Errors](#)
- [Pattern Translation](#)
- [Known Issues](#)

Simulation and Patterns

Serializer designs use MAX Testbench for pattern validation. For details, see “Using MAX Testbench” in TestMAX ATPG and TestMAX Diagnosis Online Help.

In the TestMAX ATPG tool, patterns can be read and written in the WGL, serial STIL, and binary pattern formats, and also written out in the TDL91, TSTL2, and FTDL formats.

The ATPG limitations for DFTMAX designs also apply to serializer designs. See [DFTMAX Compression Limitations on page 662](#).

STIL Protocol File

STIL protocol file examples are presented for the following cases:

- `load_unload` procedures with and without an update stage
- Chip level with and without an update stage
- Core level

Also, the following compressor structure files are considered:

- Decompressor SPF
- Compressor SPF

load_unload Procedure

For the chip-level STIL protocol file, the usage of the `load_unload` procedure, including `shift` and `test_setup`, is identical to the parallel mode as well as combinational scan compression mode. Some other UserKeywords sections used by the TestMAX ATPG tool are provided. An additional sequence provided in the `load_unload` and `shift` procedures is only for the core-level STIL protocol file.

[Example 144](#) shows a STIL protocol file example that does not include an update stage.

Example 144 STIL Protocol File Example Without an Update Stage

```

"_clk" = '"ext_clk1" + "ext_clk2" + "ser_clk"';
...
"load_unload" {
    W "default_WFT_";
    ActiveScanChains core_group;
    C {
        "dat1[0]" = N;
        ...
    }
    "ScanCompression_mode_pre_shift" : V { -- (1)
        "_clk" = 00P;
        "_si" = #;
        "_so" = #;
        "strobe" = 0;
        "test_se" = 1;
    }
    Shift {
        V { -- (2)
            "_clk" = 00P;
            "_si" = #;
            "_so" = #;
            "strobe" = 0;
        }
        V { -- (3)
    }
}

```

```

    "_clk" = 00P;
    "_si" = #;
    "_so" = #;
}
V {                                     -- (4)
    "_clk" = 00P;
    "_si" = #;
    "_so" = #;
}
V {                                     -- (5)
    "_clk" = PPP;
    "_si" = #;
    "_so" = #;
    "strobe" = 1;
}
}
}

```

This STIL protocol file example defines a configuration with one scan-in and one scan-out, and a 4-bit deserializer(serializer registers *without* an update stage. The vector (1) named “*ScanCompression_mode_pre_shift*,” which is outside the shift procedure, uses the first serializer clock (“*ser_clk*”) to load the first internal shift data into the deserializer registers. The vector (4), which is the third vector of the shift procedure, completes loading the first internal shift data into the deserializer registers. At vector (5), internal scan clocks (“*ext_clk1*” and “*ext_clk2*”) are pulsed, and the first internal shift data that has been placed on the deserializer registers is transferred to the compressed chains; also, the second internal shift data starts loading into the deserializer registers. The “*ScanCompression_mode_pre_shift*” vector is applied only to the first vector, and then the shift procedure is repeatedly applied as many times as the number of compressed chain shifts per pattern. The scan-out measure also takes place with each vector.

If the number of compressed chain shifts is 5, the actual vector sequence in a single pattern is

(1) (2)(3)(4)(5) (2)(3)(4)(5) (2)(3)(4)(5) (2)(3)(4)(5) (2)(3)(4)(5)

and the capture takes place.

[Example 145](#) shows a STIL protocol file example that includes an update stage.

Example 145 STIL Protocol File Example With an Update Stage

```

"_clk" = '"ext_clk1" + "ext_clk2" + "ser_clk" + "update_clk"';
...
"load_unload" {
    W"_default_WFT_";
    ActiveScanChains core_group;
    C {
        "dat1[0]" = N;
        ...
    }
}
"ScanCompression_mode_pre_shift" : V { -- (1)
}

```

```

        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
        "strobe" = 0;
        "test_se" = 1;
    }
    V {                                     -- (2)
        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
    }
    V {                                     -- (3)
        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
    }
    V {                                     -- (4)
        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
    }
    V {                                     -- (5)
        "_clk" = 00PP;
        "_si" = #;
        "_so" = #;
    }
Shift {
    V {                                     -- (6)
        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
        "strobe" = 0;
    }
    V {                                     -- (7)
        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
    }
    V {                                     -- (8)
        "_clk" = 00P0;
        "_si" = #;
        "_so" = #;
    }
    V {                                     -- (9)
        "_clk" = PPPP;
        "_si" = #;
        "_so" = #;
        "strobe" = 1;
    }
}

```

If you implement the update stage, you see additional vectors only at the beginning of each pattern. The first serializer clock (“ser_clk”) is pulsed at the vector (1). The vector (4) completes loading the first internal shift data into deserializer registers. At the vector (5), the first internal shift data that was placed on the deserializer registers is transferred to the update stage. At the same time, the second shift data starts loading into the deserializer registers. Then, vector (8) completes loading the second internal shift data into the deserializer register. At vector (9), taking place at the same time, the first internal shift data that has been on the update stage is transferred to the compressed chains, the second internal shift data that has been on the deserializer registers is passed on to the update stage, and the third internal shift data starts loading into the deserializer registers. If the number of compressed chain shifts is 5, the actual vector sequence in a single pattern is

(1)(2)(3)(4)(5)(6)(7)(8)(9)(6)(7)(8)(9)(6)(7)(8)(9)(6)(7)(8)(9)(6)(7)(8)(9)

and the capture takes place.

UserKeywords SerializerStructures

For the serial mode, “UserKeywords SerializerStructures” is introduced. Some of its parameters are used during DRC.

Chip-Level STIL Protocol File

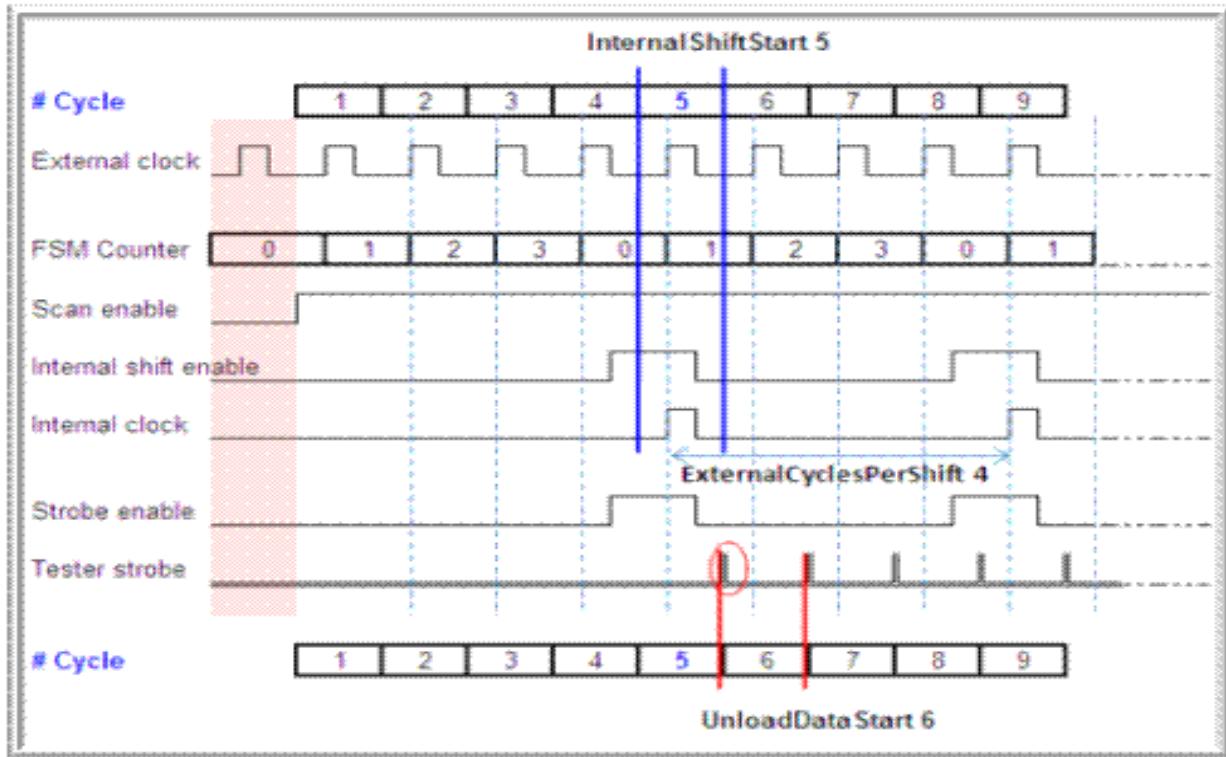
The Length <number> is the number of deserializer(serializer registers bits.

The InternalShiftStart <number>, UnloadDataStart <number>, and ExternalCyclePerShift <number> are determined by architecture. [Example 146](#) shows a chip-level STIL protocol file example without an update stage, and [Figure 414](#) shows how the Length, InternalShiftStart, UnloadDataStart, and ExternalCyclePerShift numbers are determined for this STIL protocol file example.

Example 146 Chip-Level STIL Protocol File Example Without an Update Stage

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 5;
    UnloadDataStart 6;
    ExternalCyclesPerShift 4;
    LoadSerializer "U0/U_deserializer_my_serial" {
        Length 4;
        ActiveScanChains load_group;      }
    UnloadSerializer "U0/U_serializer_my_serial" {
        Length 4;
        ActiveScanChains unload_group;
    }
}
```

Figure 414 Timing Diagram for SerializerStructures for Chip-Level STIL Protocol File Example Without an Update Stage



If the update stage is used, the “UserKeywords SerializerStructures” changes as follows:
 InternalShiftStart is delayed by 4 cycles, from 5 cycles to 9, and UnloadDataStart is delayed by 4 cycles, from 6 cycles to 10. [Example 147](#) shows this chip-level STIL protocol file example with an update stage.

Example 147 Chip-Level STIL Protocol File Example With an Update Stage

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 9;
    UnloadDataStart 10;
    ExternalCyclesPerShift 4;
    LoadSerializer "U0/U_deserializer_my_serial" {
        Length 4;
        ActiveScanChains load_group;
    }
    UnloadSerializer "U0/U_serializer_my_serial" {
        Length 4;
        ActiveScanChains unload_group;
    }
}
```

In addition to the configuration shown in [Example 147](#), if pipelined scan data is used (for example, two stages of head pipeline), the `InternalShiftStart` is delayed by another two cycles, from 9 to 11. If two stages of tail pipeline are also used, `UnloadDataStart` is delayed by two cycles, from 12 to 14. [Example 148](#) and [Example 149](#) show these two cases.

Example 148 SerializerStructures Example With Update Stage and Head Pipeline Registers

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 11;
    UnloadDataStart 12;
    ExternalCyclesPerShift 4;
    SerializerInputPipelineStages 2;
    LoadSerializer "U0/U_deserializer_my_serial" {
        Length 4;
        ActiveScanChains load_group;
    }
    UnloadSerializer "U0/U_serializer_my_serial" {
        Length 4;
        ActiveScanChains unload_group;
    }
}
```

Example 149 SerializerStructures Example With Update Stage and Head and Tail Pipeline Registers

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 11;
    UnloadDataStart 14;
    ExternalCyclesPerShift 4;
    SerializerInputPipelineStages 2;
    SerializerOutputPipelineStages 2;
    LoadSerializer "U0/U_deserializer_my_serial" {
        Length 4;
        ActiveScanChains load_group;
    }
    UnloadSerializer "U0/U_serializer_my_serial" {
        Length 4;
        ActiveScanChains unload_group;
    }
}
```

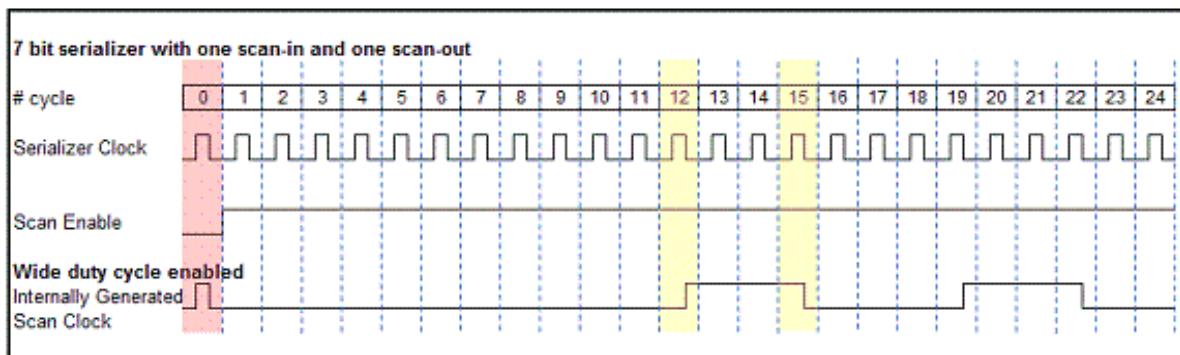
When you use the wide duty cycle feature, the “UserKeywords SerializerStructures” section of the STIL protocol file shows at which external clock cycle the leading and trailing edges of the internally generated scan shift clocks occur. [Example 150](#) provides an example with this information included.

Example 150 Chip-Level STIL Protocol File Example With Wide Duty Cycle Feature

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStartLeadingEdge 12;
    InternalShiftStartTrailingEdge 15;
    UnloadDataStart 16;
    ExternalCyclesPerShift 7;
    LoadSerializer "U0/U1_deserializer_my_serial" {
        Length 7;
        ActiveScanChains load_group;
    }
    UnloadSerializer "U0/U1_serializer_my_serial" {
        Length 7;
        ActiveScanChains unload_group;
    }
}
```

This example shows that the leading edge occurs at the 12th cycle and the trailing edge at the 15th cycle. [Figure 415](#) shows the corresponding timing diagram.

Figure 415 Correspondence Between SerializerStructures and Clock Creation



Core-Level STIL Protocol File

For a core-level test protocol, the `InternalShiftStart` is used differently. This number is always one and specifies the number of generic shift procedures performed before the first internal shift. [Example 151](#) provides a core-level STIL protocol file example.

Example 151 Core-Level STIL Protocol File Example

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 1;
    UnloadDataStart 6;
    ExternalCyclesPerShift 4;
    LoadSerializer "bottom1_U_deserializer_ScanCompression_mode" {
        Length 4;
```

```

        ActiveScanChains load_group;
    }
    UnloadSerializer "bottom1_U_serializer_ScanCompression_mode" {
        Length 4;
        ActiveScanChains unload_group;
    }
}

```

Compressor Structures

[Figure 416](#) and [Figure 417](#) contrast the compressor structures of the serial and parallel modes.

Figure 416 Decompressor SPF

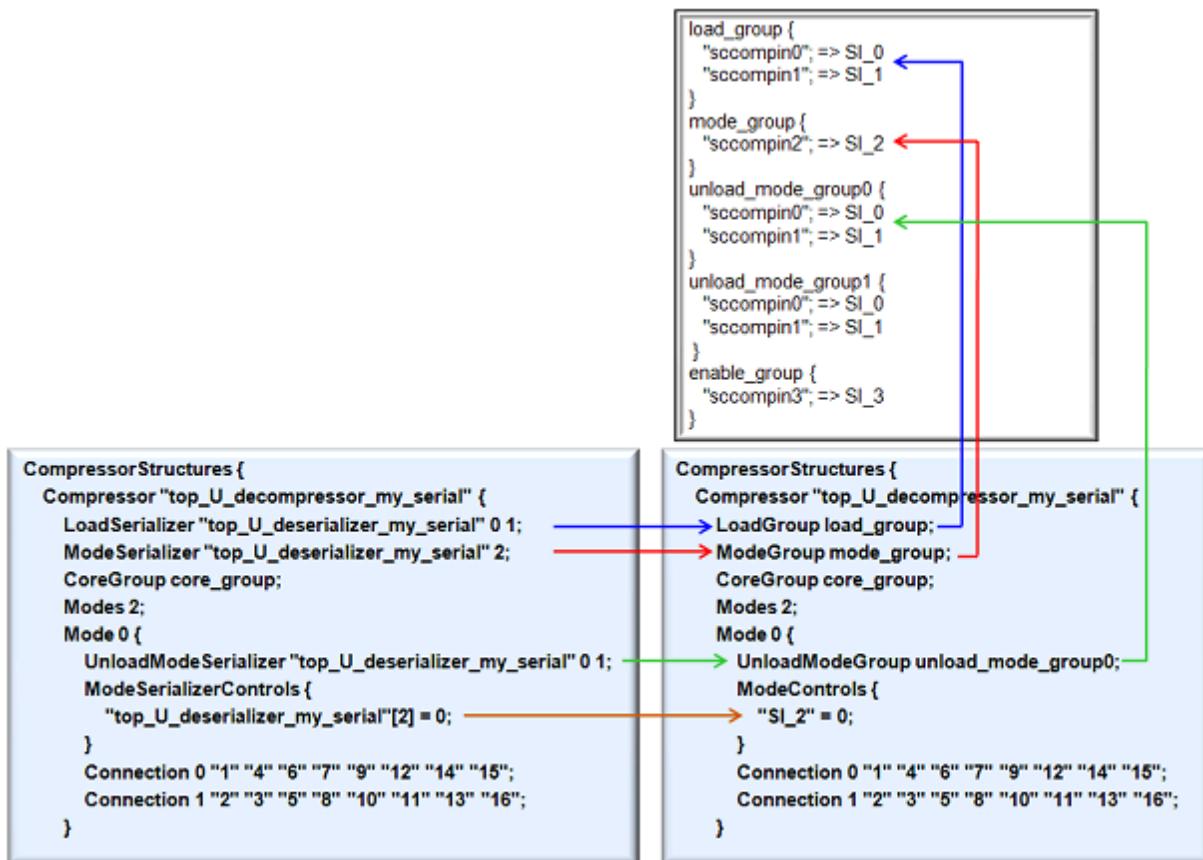
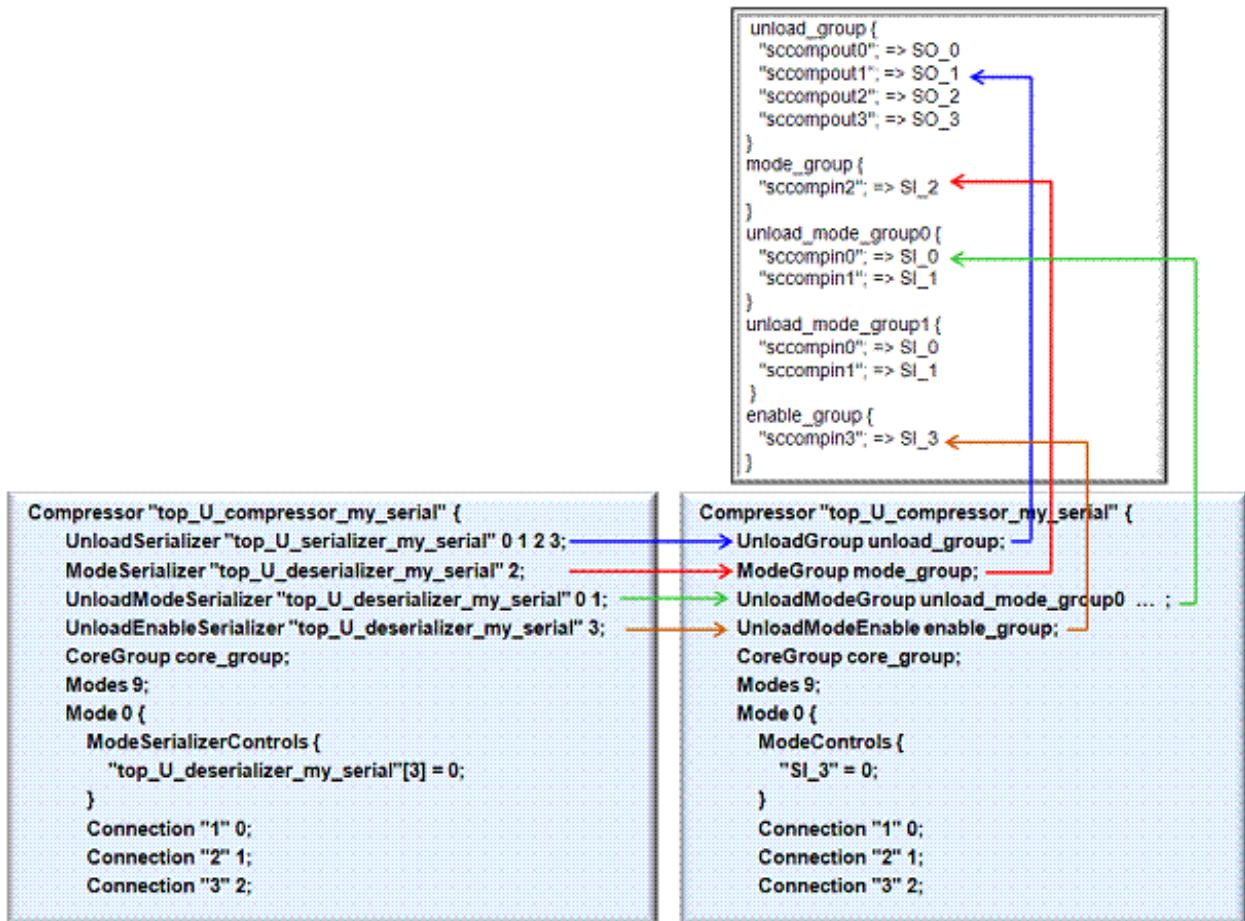
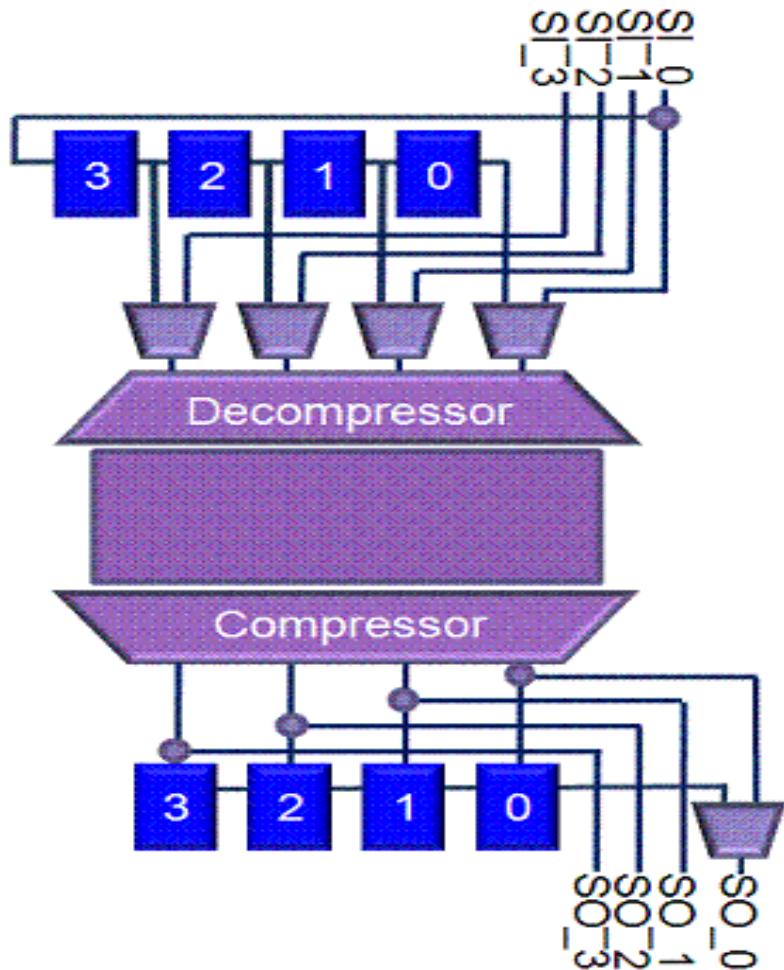


Figure 417 Compressor SPF



This comparison assumes the architecture represented in Figure 418. TestMAX ATPG assigns indexes 0 1 2 ... to the deserializer and serializer registers from the scan-out side. The scan-in port SI_0 in parallel mode corresponds to the deserializer register bit 0 in serial mode. The scan-out port SO_0 in parallel mode corresponds to the serializer register bit 0 in serial mode.

Figure 418 Correspondence Between Serial and Parallel Modes



ClockStructures

The ClockStructures section prints the output pins of the internally generated scan clocks, as shown in [Example 152](#):

Example 152 ClockStructures Example for a STIL Protocol File

```
UserKeywords DontSimulate;
ClockStructures {
    PLLStructures "serializer_init_shift_clocks" {
        Clocks {
            "u_clockcntrl/wide_clkgen/C75/U1/Z" PLLShift {
                OffState 0;
            }
            "u_clockcntrl/wide_clkgen/C75/U2/Z" PLLShift {
```

```
        OffState 0;  
    }  
}
```

This information about internally-generated clocks can help DRC in the TestMAX ATPG tool. You can prevent the `write_test_protocol` command from including this information in the SPF by setting the `test serialize put fsm clock output` variable to `false`.

Debugging TestMAX ATPG Serializer DRC Errors

When running TestMAX ATPG on designs that contain serializer blocks, you might encounter design rule violation (DRC) errors that are specific to the serializer flow. This topic provides debugging information for such DRC errors.

The following topics are discussed in this section:

- Debugging R33 Through R38 DRC Errors
 - Providing Guidance for R34 and R36 DRC Errors

Debugging R33 Through R38 DRC Errors

When R33 to R38 errors are issued by TestMAX ATPG, the following debug method might be helpful to isolate the issue.

For an R37 error:

Error: Scan cell 19806 was clocked during serializer nonshifting cycle.
(R37-1)

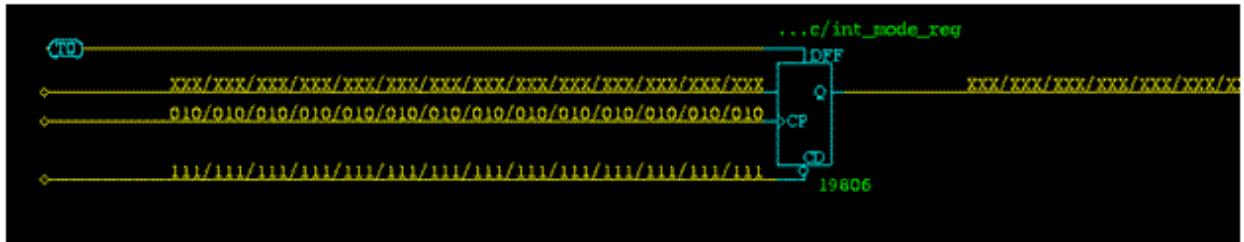
The scan cells of the compressed scan chains must be clocked as described in [Figure 374 on page 826](#). This error indicates that the scan cells are clocked at incorrect cycles.

To debug the issue, apply the following method:

```
DRC-T> set_drc -store_initial_shifts  
DRC-T> set_pindata -shift  
DRC-T> run_drc -patternexec my_serial  
DRC-T> (gui_start)
```

When you open the TestMAX ATPG GSV, look at the cell 19806. [Figure 419](#) shows the pin data.

Figure 419 Pin Data Example for a Scan Cell With an R37 Error



You can find the clock being pulsed in every shift cycle at the CP pin of the scan cell, which is not correct. Refer to the `SerializerStructures` section in your SPF to determine the correct clocking. If you see the following in the `SerializerStructures` section:

```

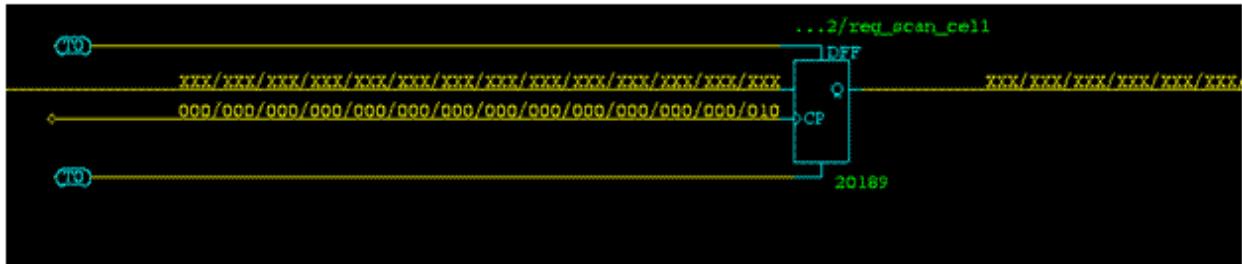
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 14;
    UnloadDataStart 17;
    ExternalCyclesPerShift 6;
    SerializerInputPipelineStages 1;
    SerializerOutputPipelineStages 2;
    LoadSerializer
"my_top_U_deserializer_ScanCompression_mode" {
    Length 4;
    ActiveScanChains load_group;
}
UnloadSerializer
"my_top_U_serializer_ScanCompression_mode" {
    Length 4;
    ActiveScanChains unload_group;
}
LoadSerializer "I_coreA/U_deserializer_ScanCompression_mode" {
    Length 6;
    ActiveScanChains "coreA_load_group";
}
UnloadSerializer "I_coreA/U_serializer_ScanCompression_mode" {
    Length 6;
    ActiveScanChains "coreA_unload_group";
}
}

```

then the first clocking for compressed scan chains is at the 14th cycle. You can check this with the value of the `InternalShiftStart`. If clocking exists in some other cycles, the clocking scheme is incorrect.

Figure 420 shows the pin data example on one of the scan cells with the correct clocking behavior.

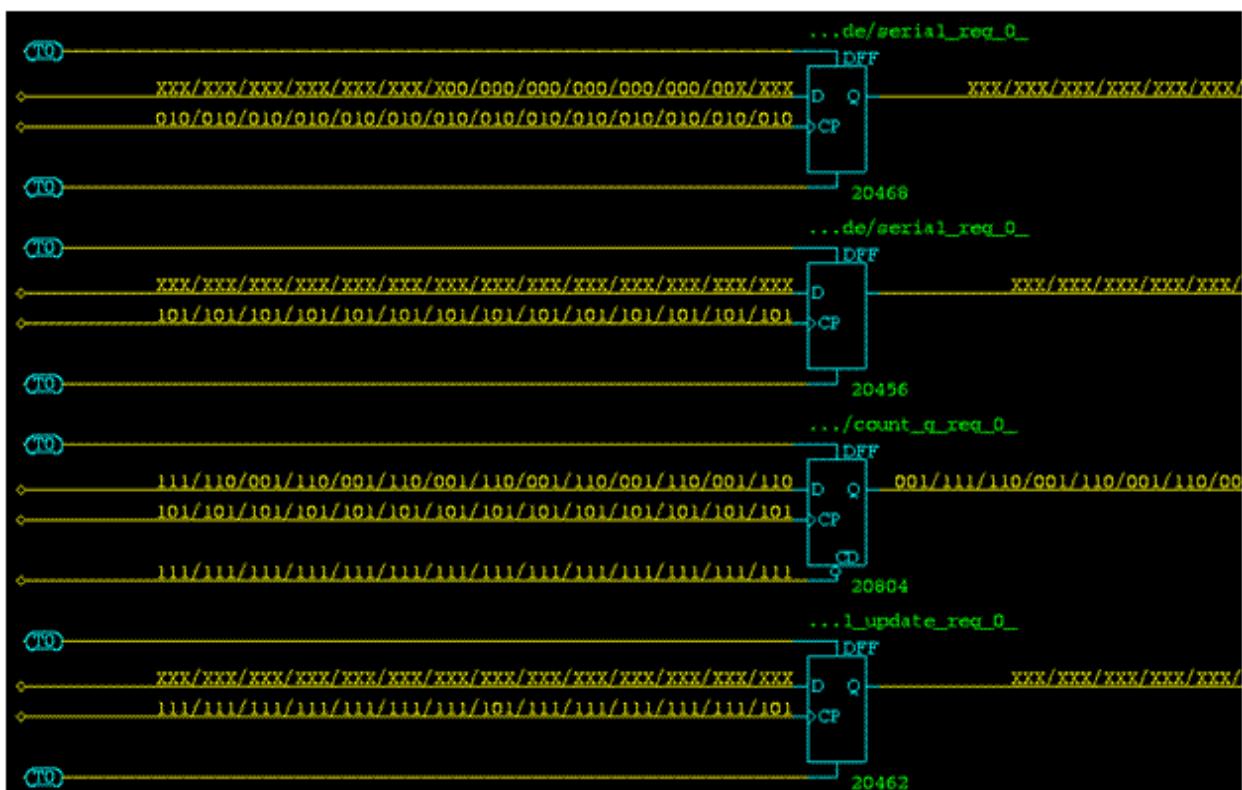
Figure 420 Pin Data Example for a Scan Cell With Correct Clocking



Verify that the clock pulse happens at 14th cycle, which is consistent with the InternalShiftStart value in the SPF.

As a reference, in the following pin data example shown in [Figure 421](#), notice what happens to one of the deserializer registers, the serializer registers, serializer FSM counter, and update stage registers, when you have the same `SerializerStructures` as the serializer structure just described.

Figure 421 Pin Data Example for Some Cells of the Serializer Logic



The first cell is one of the serializer registers at the output side. The second cell is one of the deserializer registers at the input side. The third cell is one of the serializer FSM counter registers. Those registers need to have the external clock pulses from the primary port without serializer clock gating. The fourth cell is one of the update stage registers. The clock pulses appear on the registers at the 8th cycle and 14th cycles, which can be explained in the following way:

The number of head pipeline registers is 1 and the maximum length of the serializer registers is 6. To load scan data fully to the deserializer registers, $6+1=7$ cycles are required. Then at the next cycle, which is 8th cycle, the scan data that has been loaded into the deserializer registers is transferred to the update stage registers. This is the reason why the 8th cycle on the update register has a clocking. The next scan data is also serially loaded through the pipeline register to the deserializer registers, consecutively. Since the length of the serializer registers is 6, $8+6=14$ is the next clocking for the update stage to obtain the second scan data. Also, at the 14th cycle, the data transfers from the update registers to the compressed scan chains.

The preceding examples show how to read the pin data, using the `set_drc -store_initial_shifts` command. After you apply this command, DRC in the TestMAX ATPG tool does not pass, but it does show you the stored shift data. You need to reset this setting by using the `set_drc -nostore_initial_shifts` command after you complete your debug to proceed in the same session.

Providing Guidance for R34 and R36 DRC Errors

In some cases, DRC in the TestMAX ATPG tool cannot identify the deserializer(serializer registers due to the presence of other topologically-connected nonscan cells. In these cases, DRC might issue R34 or R36 violations. The TestMAX ATPG tool provides a method to allow user guidance to be provided in the STIL protocol file to specify the correct deserializer(serializer registers.

For example, consider the following DRC violations in the TestMAX ATPG tool:

```
Error: Multiple candidates (192417,185880) for mode-port input with
serializer connection test_si1 (4). (R36-1)
Error: Multiple candidates (192418,185824) for mode-port input with
serializer connection test_si1 (3). (R36-2)
Error: Multiple candidates (192419,185825) for input 2 of load
compressor
abc_top_U_decompressor_abc. (R36-3)
Error: Multiple candidates (192420,185826) for input 1 of load
compressor
abc_top_U_decompressor_abc. (R36-4)
Error: Multiple candidates (192421,185827) for input 0 of load
compressor
abc_top_U_decompressor_abc. (R36-5)
```

To determine the cell names of the reported primitives for the first DRC error, use the `report_primitives` command on the two reported primitive IDs:

```
DRC-T> report_primitives 192417
abc_top_U_decompressor_abc/serial_reg_4_ (192417) DFF (DFF_X1)
--- I (TIE_0)
--- I (TIE_0)
!CK I 88837-u_clockcntrl/U14/X
--- I 159-
Q O 178252-/abc_top_U_decompressor_abc/serial_reg_3_/D
178253-/abc_top_U_decompressor_abc/U7/A ...
DRC-T> report_primitives 185880
xyz_dig_u/xyz_u/shiftreg_reg_47_ (185880) DFF (DFF_X2)
--- I (TIE_0)
!RD P I 143-xyz_dig_u/xyz_u/U83/X
CK I 156-xyz_dig_u/gpio_pads_ctrl_u svn_buf_s_16_u2/X
--- I 88745-
--- O 88743-
88745-
```

The reported instance names show that the primitive ID 192417 represents the correct serializer register. Next, you must identify the serializer index for this primitive. Since the R36 violations are issued on the deserializer side, which is a compressor, you must run the `report_serializers -load -verbose` command. For this example, the output is as follows:

```
DRC-T> report_serializers -load -verbose
-----
name type length
-----
abc_top_U_deserializer_abc load 5
-----
Scanin Index Serializer Index Parallel Outputs invert
-----
test_si1 0 xyz_dig_u/xyz_u/shiftreg_reg_43_ no
test_si1 1 xyz_dig_u/xyz_u/shiftreg_reg_44_ no
test_si1 2 xyz_dig_u/xyz_u/shiftreg_reg_45_ no
test_si1 3 xyz_dig_u/xyz_u/shiftreg_reg_46_ no
test_si1 4 xyz_dig_u/xyz_u/shiftreg_reg_47_ no
```

From this report, you can determine that primitive ID 192417 corresponds to the serializer index 4, as the other primitive ID 185880 matches the load register for serializer index 4. You can repeat this process to match the other serializer register names to their corresponding index values.

Next, open the STIL protocol file in a text editor. Locate the `SerializerStructures` section, and create a `ParallelOutputs` block as demonstrated in the following file:

```
UserKeywords SerializerStructures CompressorStructures;
SerializerStructures {
    InternalShiftStart 6;
    UnloadDataStart 7;
    ExternalCyclesPerShift 5;
    LoadSerializer "abc_top_U_deserializer_abc" {
        Length 5;
```

```

ActiveScanChains load_group;
ParallelOutputs {
    0 "abc_top_U_decompressor_abc/serial_reg_0_" no
    1 "abc_top_U_decompressor_abc/serial_reg_1_" no
    2 "abc_top_U_decompressor_abc/serial_reg_2_" no
    3 "abc_top_U_decompressor_abc/serial_reg_3_" no
    4 "abc_top_U_decompressor_abc/serial_reg_4_" no
}
}
UnloadSerializer "abc_top_U_serializer_abc" {
    Length 5;
    ActiveScanChains unload_group;
}
}

```

Note the following points when providing deserializer(serializer register guidance to DRC in the TestMAX ATPG tool:

- The numbers specified before the serializer register name on each line must match the serializer index values as reported by the `report_serializers -verbose` command.
- The serializer register names should be enclosed in double quotation marks.
- The `no` or `yes` value specified after the serializer register names specifies whether a logic inversion exists between the data input pin of each serializer register and the corresponding scan port.
- When you provide deserializer register (input side) guidance as with the previous example, you must supply it in a `ParallelOutputs` block inside the `LoadSerializer` section. When you provide serializer register (output side) guidance, you must supply it in a `ParallelInputs` block inside the `UnloadSerializer` section.
- Guidance is only needed for deserializer(serializer registers with R36 DRC violations. You do not need to supply guidance for other deserializer(serializer registers.

When DRC processes the updated STIL protocol file, it will verify that the specified register names and inversion flags are correct. If DRC determines that the register name is incorrect, it will issue an M873 warning message:

```

Warning: Possibly incorrect load serializer parallel output
specification: %d %s. (M873)
Warning: Possibly incorrect unload serializer parallel input
specification: %d %s. (M873)

```

If DRC determines that the register inversion flag is incorrect, it will issue an M874 warning message:

```

Warning: Possibly incorrect load serializer parallel output inversion
specification: %d %s %s. (M874)
Warning: Possibly incorrect unload serializer parallel output inversion
specification: %d %s %s. (M874)

```

When the register names and inversion flags are valid, TestMAX ATPG DRC honors the specified serializer register definitions and proceeds with the DRC process.

Pattern Translation

This topic describes the following types of serialized scan pattern translation:

- [Translating Parallel Mode Patterns to Serial Mode Patterns](#)
- [Translating Serial Mode Patterns to Standard Scan Mode Patterns](#)

Translating Parallel Mode Patterns to Serial Mode Patterns

Designs with serialized compressed scan can have both a serial scan mode, where the codec is connected to deserializer and serializer registers for I/O-limited operation, and a parallel scan mode, where the codec is connected directly to top-level scan I/O ports. If the same codec is used in both modes, you can create parallel mode scan patterns in TestMAX ATPG first, then translate them to serial mode scan patterns. This eliminates the need for a second pattern generation run.

To ensure that the same codec is used in both modes, you must use the `-parallel_mode` option of the `set_serialize_configuration` command to tie the parallel mode to the serial mode:

```
set_scan_compression_configuration \
  -base_mode my_base_mode \          # standard scan base mode
  -test_mode my_serial_mode \        # serial compressed scan mode
  -xtolerance ... \
  -chain_count ... \
  -inputs ... \
  -outputs ... \
  -serialize ...

set_serialize_configuration \
  -test_mode my_serial_mode \        # serial compressed scan mode
  -parallel_mode my_parallel_mode \  # parallel compressed scan mode
  -inputs ... \
  -outputs ...
```

Note:

The terms *serial* and *parallel* in this section refer to the type of scan compression being used, not to serial or parallel scan data loading in TestMAX ATPG testbenches.

Performing Pattern Translation for Matching Scan Data Pipeline Depths

Use this pattern translation flow if you are not using pipelined scan data, or if you are using pipelined scan data and the parallel mode has the same pipeline depth as the serial mode.

Note:

When you use the tool to perform automatic pipeline register insertion, it ensures that all test modes have the same pipeline depth.

To use this pattern translation flow, perform the following steps:

1. Run TestMAX ATPG in parallel scan mode.
2. Execute the `run_drc` command using the parallel mode SPF:

```
run_drc my_parallel_mode.spf
```

In the log file, you will see information reported during compressor rules checking:

```
Begin compressor rules checking...
Warning: Rule R11 (X on chain affects observe ability of other chains)
was violated 1008 times.
Compressor rules checking completed: #chains=200, #scanins=8,
#scanouts=8, #shifts=100, CPU time=0.13 sec.
```

Note the `#shifts=` value, which represents the number of shift cycles used in the parallel mode.

3. Perform ATPG in the parallel mode.
4. Write out the parallel mode pattern set with the `-format binary` and `-compressor_based` options:

```
write_patterns compressor_based.db \
  -format binary -replace -compressor_based
```

When you write out the pattern set with the `-compressor_based` option, the pattern set can be only read back into a serial scan mode TestMAX ATPG run.

5. Run TestMAX ATPG in serial scan mode.
6. Set the number of shift cycles to the `#shifts=` value obtained from the compressor rules checking log file entry from the parallel mode run:

```
set_drc -dftmax_shift_cycles 100
```

7. Execute the `run_drc` command using the serial mode SPF:

```
run_drc my_serial_mode.spf
add_nofaults ...
add_faults ...
```

8. Read the previously saved patterns into the TestMAX ATPG tool:

```
set_patterns -external compressor_based.db
```

9. Optionally, execute the `run_simulation` command, the `run_fault_sim` command, or an incremental ATPG step:

```
run_simulation
run_fault_sim
```

10. Write out the translated serial mode pattern set:

```
write_patterns translated_serial.stil -format stil -external
```

It is expected that you might see a small amount of coverage difference between the original parallel mode ATPG results and the `run_fault_sim` result using the translated serial mode patterns. This difference can be caused by different lock-up latch configurations, different scan chain MUXing, different primary input constraints, and other minor scan configuration differences. You should use options for the `set_build` and `set_drc` commands that are as similar as possible between the parallel mode and the serial mode runs.

Performing Pattern Translation Across Different Scan Data Pipeline Depths

Use this pattern translation flow if you are manually inserting pipelined scan data registers and the parallel mode has a different pipeline depth from the serial mode.

To use this pattern translation flow, perform the following steps:

1. Run TestMAX ATPG in parallel scan mode.
2. Execute the `run_drc` command using the parallel mode SPF:

```
run_drc my_parallel_mode.spf
```

In the log file, you will see information reported during compressor rules checking:

```
Begin compressor rules checking...
Warning: Rule R11 (X on chain affects observe ability of other
chains)
was violated 1008 times.
Compressor rules checking completed: #chains=200, #scanins=8,
#scanouts=8, #shifts=201, CPU time=0.13 sec.
```

Note the `#shifts`= value, which represents the number of shift cycles used in the parallel mode.

3. Run TestMAX ATPG in serial scan mode.
4. Execute the `run_drc` command using the serial mode SPF:

```
run_drc my_serial_mode.spf
```

Obtain the resulting #shifts= value from the serial mode:

```
Begin compressor rules checking...
Warning: Rule R11 (X on chain affects observe ability of other
chains)
was violated 1008 times.
Compressor rules checking completed: #chains=200, #scanins=8,
#scanouts=8, #shifts=202, CPU time=0.13 sec.
```

5. Take the larger #shifts= value from the two test modes. In this example, the larger value is 202.
6. Run TestMAX ATPG in parallel scan mode.
7. Set the number of shift cycles to the larger #shifts= value obtained from the parallel and serial modes:

```
set_drc -dftmax_shift_cycles 202
```

8. Follow the steps in “[Performing Pattern Translation for Matching Scan Data Pipeline Depths](#),” starting with the parallel mode ATPG performed in step 3. In the serial mode in step 6, use the larger #shifts= value determined from the two test modes.

Translating Serial Mode Patterns to Standard Scan Mode Patterns

To convert serial mode scan patterns to standard scan mode format, use the translation flow provided in “[Translating DFTMAX Compressed Patterns Into Normal Scan Patterns](#)” in TestMAX ATPG and TestMAX Diagnosis Online Help. This translation flow applies to both normal compressed scan patterns as well as serialized scan patterns.

Known Issues

The known issues for serializer designs in a TestMAX ATPG flow are described in the following topics:

- [C1 Violations](#)
- [Serializer Core-Level Flow With Pipelined Scan Data Insertion](#)

C1 Violations

A C1 violation might occur in parallel mode or regular scan mode when you use Synopsys automated pipeline scan data in which the clock is shared with the ATE clock of a DFT-inserted OCC controller. The violation is related to gate-level optimization and causes the clock-off state to be X on a lock-up-latch clock pin that has been inserted after the pipeline head registers. The violation can be reduced to a warning if the situation is the same as described earlier. You cannot expect to be able to downgrade all C1 violations.

Serializer Core-Level Flow With Pipelined Scan Data Insertion

Serialized compressed scan core creation with implemented pipeline stages could produce R rule violations during DRC in the TestMAX ATPG tool. One workaround is to implement the pipeline stage only at the top level in a HASS or Hybrid flow.

DFTMAX Compression With Serializer Limitations

The following functionalities are not supported:

- Integrating unserialized DFTMAX compression cores
 - If you integrate DFTMAX compression cores, you must use serializer IP insertion to serialize them.
- Having a different number of scan ports between a serial mode and a standard scan mode during HASS or Hybrid core integration
- Multiple serializer test modes
- Sparse scheduling

You cannot use the `-target` option of the `define_test_mode` command to target some cores but not others (also known as sparse targeting) in serializer and serializer IP insertion flows.

- Parallel mode support in top-down partition with concatenated serializer chain flow, HASS with concatenated serializer chain flow, and Hybrid flows
- DFT connectivity associations using the `set_dft_connect` or `set_dft_signal -connect_to` commands
- Pipeline scan data registers whose clock is shared with scan cells' clock

The tool inserts the serializer clock controller on the clock lines to provide internally generated clocks to the compressed scan chains. If it is inserted on the clock line feeding the pipelined scan data registers, the pipeline registers do not work properly.

- Pattern translation from serial to parallel
- Launch-on-shift (LOS) transition ATPG
- Internally generated scan-enable signals
- LSSD, scan-enabled LSSD, and clocked scan styles
- Lock-up flip-flops
- Retiming flip-flops

- External on-chip clocking (OCC) chains
- External shift power control (SPC) chains
- Terminal lock-up latches

When enabled, terminal lock-up latches are inserted at the end of the compressed scan chains (before the serializer compressor) instead of at the scan output ports. This might result in scan structures that do not shift into the compressor correctly.

- Mix of `-xtolerance high` and `-xtolerance default` codecs in top-down partition, HASS, and Hybrid flows
- Any case with core wrapping in which a dedicated wrapper clock is created when the `insert_dft` command is run

The dedicated wrapper clock is not gated by the serializer clock controller.

- Any case with core wrapping in which the DFTMAX Hybrid integration mode is also used
- Timing constraints for PrimeTime cannot be written from TestMAX ATPG using the `tmax2pt.tcl` utility.

Out-of-Scope Serializer Functionality

The following serializer functionalities are out of the current scope:

- Serialized standard scan mode
- Serialized asymmetrical I/O codec compression
- Serialized core with standard scan chains

This is supported only when all scan chains are compressed by the serializer codec

- Legacy Verilog testbench

DFTMAX Compression Error Messages

The following TEST error messages involve the serializer feature.

TEST-1093

Size of the deserializer and serializer are not equal.

TEST-1094

The number of deserializer inputs and serializer outputs are not equal.

TEST-1095

Scan compression mode chains that are outside the codec are not supported in the serializer flow.

TEST-1096

The head and tail pipeline flip-flops are not triggered by the same clock in the serializer flow.

TEST-1097

Pipeline clock is not dedicated to pipeline flip-flops in serializer flow.

Part 4: DFTMAX Ultra Compression

22

Introduction to DFTMAX Ultra

DFTMAX Ultra compression is an advanced test compression technology that is designed for hierarchical flows to deliver high quality results as measured by test time, data volume, design area and congestion, and time to implementation. The technology delivers very high compression, even with few scan I/O pins. It uses the same signal interface as standard scan with minimal impact to the clock tree. The technology is designed to deliver good results with few internal chains to minimize any impact on layout.

The following topics introduce DFTMAX Ultra compression:

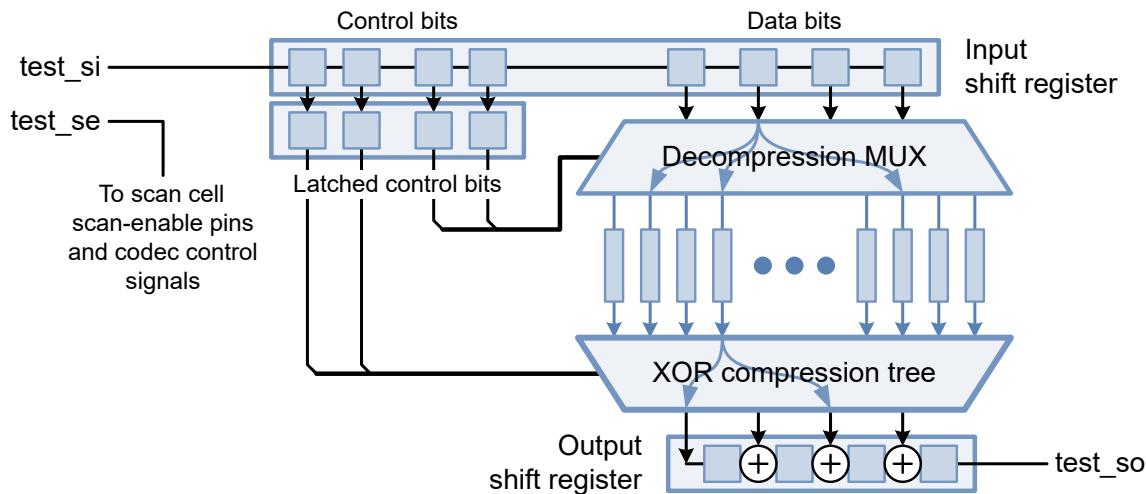
- [The DFTMAX Ultra Compression Architecture](#)
 - [Usage Flow](#)
 - [Hierarchical DFT Insertion](#)
 - [Test Pattern Creation Using TestMAX ATPG](#)
 - [Pattern Simulation](#)
-

The DFTMAX Ultra Compression Architecture

DFTMAX Ultra compression is an advanced method of scan compression that provides high levels of compression, high fault coverage, short scan chains, and low pin count. The scan architecture uses a shift register structure to shift in and shift out the scan data streams, allowing all test operations to occur at high frequencies. All scan circuits operate at the same frequency and no codec clock controller circuit is needed for scan operations.

[Figure 422](#) shows the basic decompression and compression (codec) architecture. The codec logic uses an existing scan clock; for simplicity, clock connections are not shown.

Figure 422 Basic DFTMAX Ultra Compression Architecture



The scan-in data port feeds an input shift register. Some bits in the shift register are used as control bits, while others are used as data bits. The control bits are latched once per pattern and the latched values configure the compression logic for the pattern. The data bits supply streaming data to the decompression multiplexer (MUX), which is a combinational logic block that distributes the data to the compressed scan chains.

At the scan chain outputs, a combinational XOR compression tree and a sequential XOR output shift register compress the data into a single stream. The compression tree is a multilevel combinational network of XOR gates that compresses the bits from the scan chains into a smaller number of bits. The output shift register compresses the data further to produce a single bit per shift cycle. Redundant connections to the output shift register help minimize the effect of X values.

The DFTMAX Ultra compression architecture allows test data to be streamed in through a single input port and to be read out through a single output port, using the normal shift clock. At the same time, the input and output shift registers allow high levels of compression to be achieved with very good fault coverage.

DFTMAX Ultra compression is an optional add-on to DFTMAX compression. Together, they synthesize the streaming scan compression circuitry. You specify the number of scan inputs, number of scan outputs, and the target number of chains. The tool then determines the optimum architecture for optimal compression and fault coverage possible with the available resources.

Usage Flow

To use DFTMAX Ultra compression, you specify the number of scan data inputs and outputs and the target number of scan chains. The tool determines the optimum

architecture to achieve the desired compression with the available resources and synthesizes the DFT compressor and decompressor (codec) circuitry. It also generates a STIL file to describe the test protocol and codec architecture.

The following example shows a typical DFTMAX Ultra compression script:

```
set_dft_configuration -streaming_compression enable
set_scan_configuration -chain_count 1
set_streaming_compression_configuration -chain_count 80
set_dft_signal -port SI1 -type ScanDataIn
set_dft_signal -port SO1 -type ScanDataOut
...

```

The `set_dft_configuration -streaming_compression enable` command enables DFTMAX Ultra compression.

The chip can be tested in two different modes: standard scan (uncompressed) mode and compressed scan mode. The `-chain_count` option is used with two different commands to specify the number of chains in these two modes:

- In the `set_scan_configuration` command, the `-chain_count` option specifies the number of scan chains for the standard scan chains, which are the scan chains used in standard scan mode. (This option value is also the default number of input and outputs ports used in compressed scan mode so that both modes share the same I/O characteristics.)
- In the `set_streaming_compression_configuration` command, the `-chain_count` option specifies the target number of compressed scan chains, which are the scan chains used in compressed scan mode.

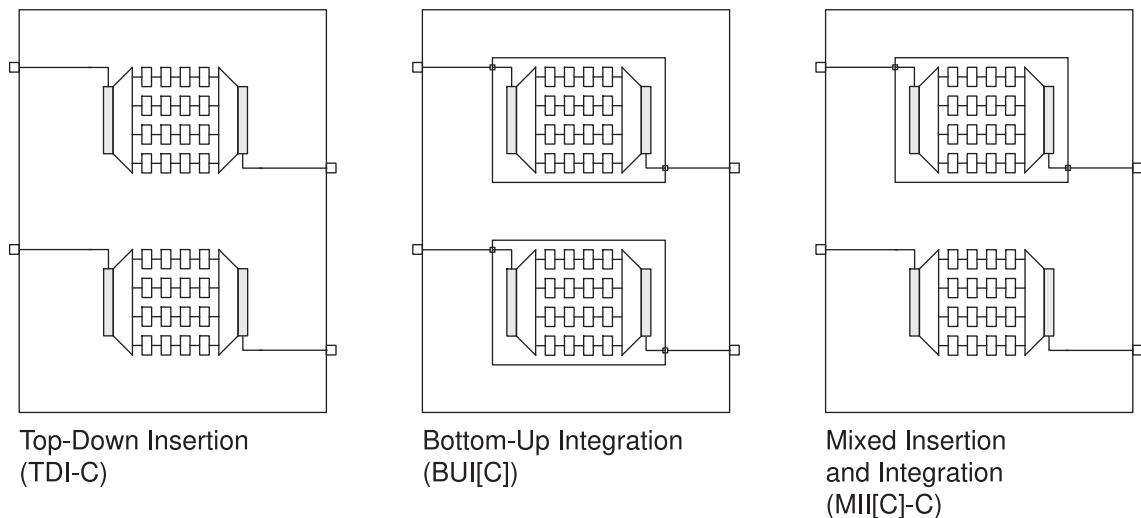
The tool synthesizes scan compression circuitry for the target number of compressed scan chains using the specified number of scan data inputs and outputs. In this example, the circuit has one scan input, one scan output, and 80 compressed scan chains.

Hierarchical DFT Insertion

DFTMAX Ultra compression supports hierarchical DFT insertion. You can perform scan synthesis independently for each lower-level block. When you use instances of these blocks at a higher level of hierarchy, the tool integrates the scan circuitry of the lower-level blocks at the higher level.

There are many ways to build and integrate lower-level blocks to create the top-level design. [Figure 423](#) shows a few examples.

Figure 423 Hierarchical DFT Examples



Several criteria can help you choose the best strategy for a hierarchical design:

- What is the target compression level? The need to reduce tester time and test data volume can determine the DFT flow and choice of block sizes.
- How many I/O pins at the top level are available for use as test I/O pins? Using a larger number of available pins can improve testing speed and quality of results.
- Is the chip layout congested? To reduce congestion and preserve routing resources, you can partition the design into smaller blocks, each having its own test circuitry.
- Are multiple test modes needed? Should the modes be implemented at the top level or at the block level? Which combinations of lower-level test modes need to be accessible at the top level? You could use different testing modes such as high compression, low compression, and standard scan, for different purposes.

When using hierarchical scan synthesis, it is important to consider the top-level scan architecture at the core level. Cores built with scan I/O counts that are a multiple of the top-level scan I/O count give maximum chain balancing flexibility during integration. Careful consideration of hierarchical block integration in the early stages of the design flow can have a significant impact on the final test coverage and pattern count.

Test Pattern Creation Using TestMAX ATPG

The `write_test_protocol` command writes out a STIL protocol file (SPF) containing a description of the test circuitry in a given test mode. For compression modes, the SPF contains information about the scan compression architecture. You use a separate

`write_test_protocol` command for each test mode that will be used for testing the device.

The TestMAX ATPG tool performs automatic test pattern generation for the DFTMAX Ultra compression designs. The tool has knowledge of the DFTMAX Ultra compression architecture and its pattern decompression and compression algorithms. Given the design netlist and an SPF, TestMAX ATPG generates a set of test patterns for that test mode. The tool attempts to get the best possible fault coverage using a reasonable number of patterns.

For more information about running TestMAX ATPG on DFTMAX Ultra designs, see “Using TestMAX ATPG and DFTMAX Ultra Compression” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Pattern Simulation

The test synthesis flow typically uses VCS simulation to validate the test protocol and test patterns. You can choose either serial or parallel loading of scan data patterns for simulation. Use serial loading to simulate the full scan-in and scan-out behavior of the test circuitry and test protocol. Use parallel loading of patterns to simulate just the launch and capture phases of test.

Parallel simulation of test patterns is much faster than serial simulation. However, only serial simulation can fully validate the scan circuitry and test protocol. You can use serial loading for the first few patterns for complete testing, and then use parallel loading for fast simulation of many patterns.

23

DFTMAX Ultra Compression Architecture

DFTMAX Ultra compression uses a shift-register scan-data architecture and a single scan clock to deliver very high compression without restriction on the number of I/O pins. The input shift register feeds the decompression logic that provides data to many internal scan chains. The output shift register compresses the scan-out data using XOR logic. This architecture delivers high scan compression levels while providing a scan-compatible interface that retains the simplicity of a basic scan design.

The following topics describe the DFTMAX Ultra compression architecture:

- [DFTMAX Ultra Compression Architecture](#)
 - [Multiple-Input, Multiple-Output Architecture](#)
 - [DFTMAX Ultra Architectures for On-Chip Clocking \(OCC\)](#)
-

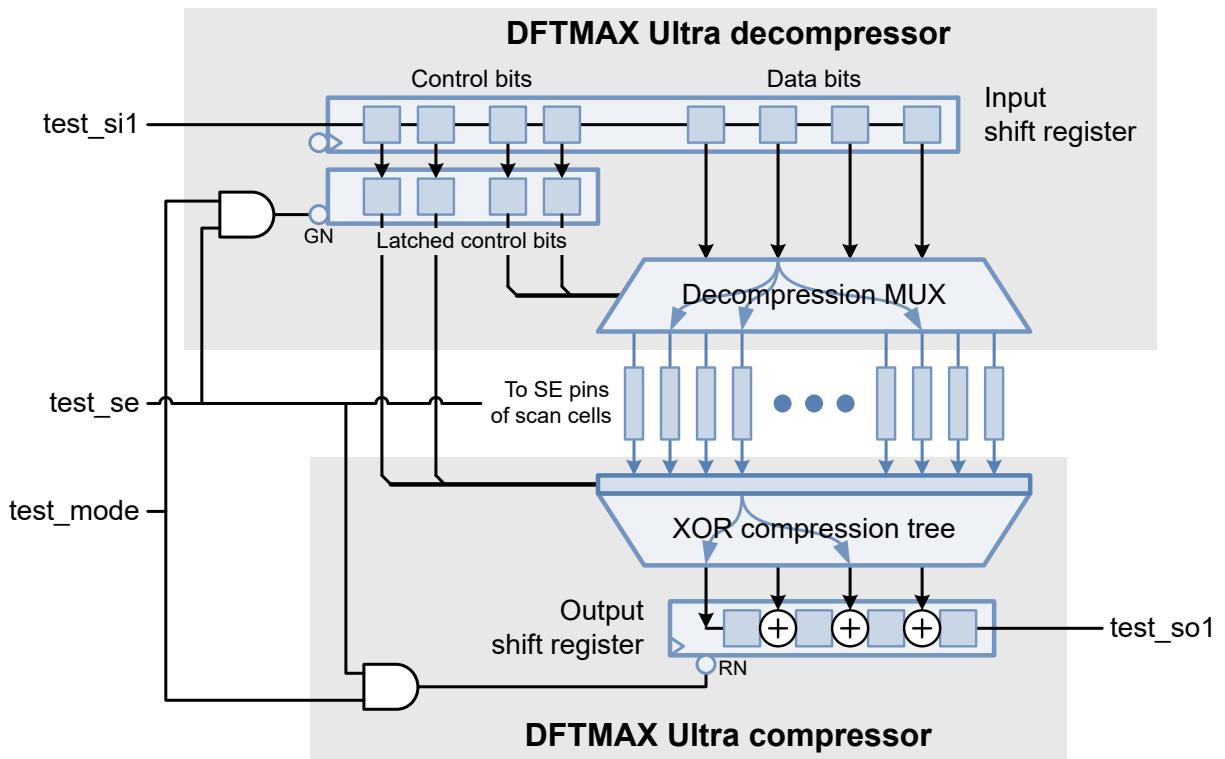
DFTMAX Ultra Compression Architecture

DFTMAX Ultra compression supports high levels of compression, high fault coverage, short scan chains, and low pin count. The scan architecture uses shift-register structures to feed in and read out the scan data streams, which enables high shift frequencies. All scan circuits operate at the same frequency; no codec clock controller circuit is needed for scan operations.

To insert DFTMAX Ultra scan compression, you specify the number of scan inputs and scan outputs, and the target number of compressed scan chains. The tool then implements an architecture based on your configuration. The DFTMAX Ultra architecture supports high compression levels using as few as one scan input and one scan output, even for a large number of internal scan chains.

[Figure 424](#) shows a block diagram for a single-input, single-output DFTMAX Ultra codec. Clock and control signals are active-high. The codec logic uses an existing scan clock; for simplicity, clock connections are not shown.

Figure 424 DFTMAX Ultra Compression Architecture, Single Scan-In and Scan-Out Pins



The features and function of the DFTMAX Ultra architecture are covered in the following topics:

- [Input Shift Register and Decompression MUX](#)
- [Control Register](#)
- [Output XOR Compression Tree and Shift Register](#)
- [Test Pattern Scan Procedure](#)
- [Scan-Enable Signal Requirements for Codec Operation](#)

Input Shift Register and Decompression MUX

The input decompressor circuit uses a shift register and a decompression multiplexer (MUX). The input scan data at the scan-in pin feeds into the shift register, which is clocked on the trailing clock edge at the normal scan clock rate. In this implementation example, the register has eight bits. The four register bits farthest from the scan-in pin feed into the decompression MUX.

The decompression MUX is a combinational logic block that causes each of the four scan data bits to fan out to multiple scan chains. The mapping of the four bits from the shift register to the scan chains remains constant for a particular pattern. However, the mapping can change from one pattern to the next. The mapping is controlled by bits in the control register. (For more details, see the next section, “[Control Register](#).”)

The shift-register structure that feeds into the decompression MUX causes the input data to stream into the scan chains multiple times. Therefore, the timing is shifted by one clock cycle from successive register bits in the shift register. This architecture allows the TestMAX ATPG tool to disperse the input data stream both in space and time – in space by fanning out to multiple chains under the control of the decompression MUX, and in time by controlling the stream of bits feeding the shift register. Although a single data stream enters the design, different chains receive different data by this time shifting.

The four last-arriving bits in the shift register are latched into the control register when the scan enable signal, `test_se`, is de-asserted. This de-assertion occurs exactly once per pattern. Other than this once-per-pattern latching function, the input shift register bits feeding the control register operate only as a time-delay pipeline for the input data stream.

Control Register

The control register is a bank of latch cells that stores the configuration of the scan circuit for a given pattern. Some of the register bits control the mapping of input shift-register bits to scan chains through the decompression MUX, while others control the X-masking logic at the ends of the scan chains. In this example, two bits control the decompression MUX and two bits control the X-masking logic. The control register latches remain constant during scan shifting, so the scan configuration stays the same within a given pattern.

To program the control register, TestMAX ATPG appends the desired string of control bits to the end of the previous pattern’s data stream. When scan-in completes, the control bits occupy the register positions that feed into the control register. The de-asserted scan-enable signal, `test_se`, latches these bits into the control register. Thus, the final bits of the scan-in pattern control X-masking for the current pattern to be scanned out and the decompression MUX mapping for the next pattern to be scanned in.

Output XOR Compression Tree and Shift Register

The output compressor circuit uses a combinational XOR compression tree and a sequential XOR output shift register.

The XOR compression tree is a multilevel combinational network of XOR gates that compresses the output bits from the scan chains into a smaller number of bits. Each scan chain feeds into multiple XOR tree outputs; this redundant logic helps to minimize the propagation of X values. In this example, the scan chain outputs are compressed into four bits that feed the output shift register.

If TestMAX ATPG determines that there are too many X values for the redundant XOR tree to isolate the X values, it invokes X-masking to block one or more scan chains during scan-out. The X-masking bits from the control register specify the chain or chains to mask for the current pattern and also specify the order of the signals feeding into the XOR compression logic.

The compressed bits feed into a chain of flip-flops that operate as an output shift register, which is clocked on the leading clock edge at the normal scan clock rate. During scan capture, the register is reset by the scan-enable signal. During scan shift, the XOR gate between each stage of the shift register merges scan data from an XOR compressor output into the scan data already moving through the output shift register. This architecture further compresses the scan data outputs from the XOR compression tree into a single data stream.

Test Pattern Scan Procedure

The ATE equipment performs the scan-in, scan-out procedure as specified by TestMAX ATPG. In this example, the scan procedure uses 15 extra clock cycles to flush out the extra bits from the input and output shift registers. For example, if the longest scan chain is 10,000 bits long, then the scan-in, scan-out procedure takes 10,015 scan clock cycles.

Consider two consecutive test patterns, 1 and 2, starting from the point at which pattern 1 has just been scanned in:

1. The first four bits of the input shift register (shifted in at the end of pattern 1) contain the desired scan control bits to be latched into the control register. The output XOR shift register contains leftover data from the previous pattern.
2. The scan-enable signal `test_se` changes from high to low, transitioning the device from scan shift mode to scan capture mode. The de-assertion of the `test_se` pin performs the following:
 - It latches the four control bits from the input shift register into the control register latches. This configures the X-masking circuit to scan out the data from pattern 1, and it configures the decompression MUX to decompress the data for incoming pattern 2.
 - It resets the output XOR shift register to known zero values.
3. The ATE equipment applies the test vector to the primary inputs of the device and reads the output vector from the primary outputs.
4. A clock pulse applied to the clock input causes the capture event, which changes the contents of the scan flip-flops.
5. The scan-enable signal `test_se` is asserted, which transitions the device from scan capture mode back into scan shift mode.

6. The ATE equipment applies a sequence of clock pulses at the scan clock rate. This reads out the captured scan data for pattern 1 through the test_so1 output and, at the same time, scans in the data for pattern 2 through the test_si1 input.
7. Scan-in and scan-out continue until the scan chains are filled with the data for pattern 2 (and the first four bits of the input shift register are filled with the control bits for the next pattern).

This same sequence is repeated for each pattern until the device is fully tested.

Before the initial scan-in of test pattern data, the MUX control bits of the control register must be programmed with values for proper decompression of the first test pattern. Therefore, the first test cycle uses an abbreviated “padding” pattern containing only the MUX control bits and no actual scan data. For more information about padding patterns, see “Optimizing Padding Patterns” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Scan-Enable Signal Requirements for Codec Operation

When the streaming codec scan-enable signal is de-asserted, the control register latches new control bit values, and the output shift register resets to a known state. Therefore, for proper operation, *this scan-enable signal must be held in the inactive state in all capture procedures*.

If you use the STIL protocol file created by the tool, the protocol already meets this requirement. In the capture procedures, the tool constrains all scan-enable signals that drive streaming codecs to the inactive state.

Note:

In some flows, streaming codecs cannot use signals defined with the `-usage` option of the `set_dft_signal` command. See [DFT Synthesis Limitations on page 981](#).

Note:

When OCC controllers are present, the tool uses different behavior that could constrain additional scan-enable signals. For more information, see [OCC Controllers and Streaming Codec Scan-Enable Constraints on page 947](#).

If you use a custom STIL protocol file, make sure that all scan-enable signals used by DFTMAX Ultra codecs are constrained to the inactive state in all capture procedures.

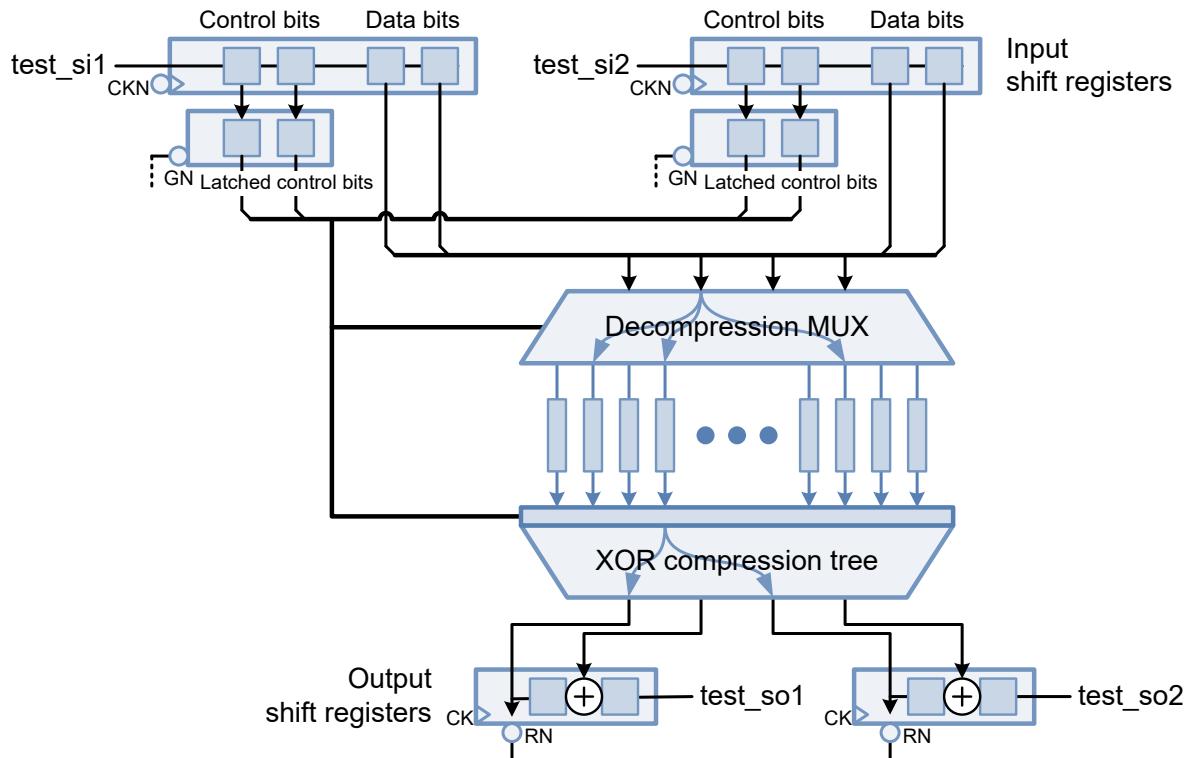
Multiple-Input, Multiple-Output Architecture

If the codec is configured to use multiple scan-in and scan-out connections, the tool synthesizes the scan circuitry in a manner similar to the single-pin circuit, but it splits the

input and output shift registers into smaller segments and connects them to the available input and output pins. By using more scan I/O pins, you get shorter shift registers, improved controllability, and improved observability into the design.

[Figure 425](#) shows a DFTMAX Ultra codec with two scan data inputs and two scan data outputs. The codec has four input shift-register scan data bits feeding into the decompression MUX, four control register bits that control the decompression and compression logic, and four output shift-register bits.

Figure 425 DFTMAX Ultra Compression Architecture, Multiple Scan-In and Scan-Out Pins



The tool determines the total input and output shift register lengths based on the number of compressed scan chains, then it then splits these shift register lengths across the scan inputs and outputs (rounding up shorter registers as needed). Therefore, as you increase the number of scan inputs and outputs, the scan shift overhead of the shift registers is reduced.

This bit distribution is more flexible than the single-input, single-output design because there are multiple independent data streams rather than one. This flexibility might allow the same fault coverage to be achieved with fewer patterns, but at the cost of using more device pins.

In this example, the distribution of bits in the input shift registers allows TestMAX ATPG to generate two independent data streams at the same time, one each for test_si1 and test_si2. Each input shift register provides its own scan data bits for multiplexing to the scan chains. For each compressed scan chain, TestMAX ATPG has a choice of up to four different bit streams: two from test_si1 and two from test_si2.

The control register bits are the last data bits shifted into the device for a pattern. Therefore, the bits of the shift register used for loading the control register are located closest to the input pin, whereas the bits of the shift register that are available to the decompression MUX are located farthest from the input pin.

On the output side, the output XOR shift register is divided into two smaller shift registers, one each for the output pins test_so1 and test_so2. This reduces the amount of data compression performed in the shift register, which reduces the propagation of X values and provides greater observability into the design.

DFTMAX Ultra Architectures for On-Chip Clocking (OCC)

On-chip clocking (OCC) controllers allow on-chip clock sources to be used for at-speed capture during device testing. In an OCC controller flow, the *clock chain* is a special scan segment that provides control over the at-speed capture pulse sequence generated by the OCC controller.

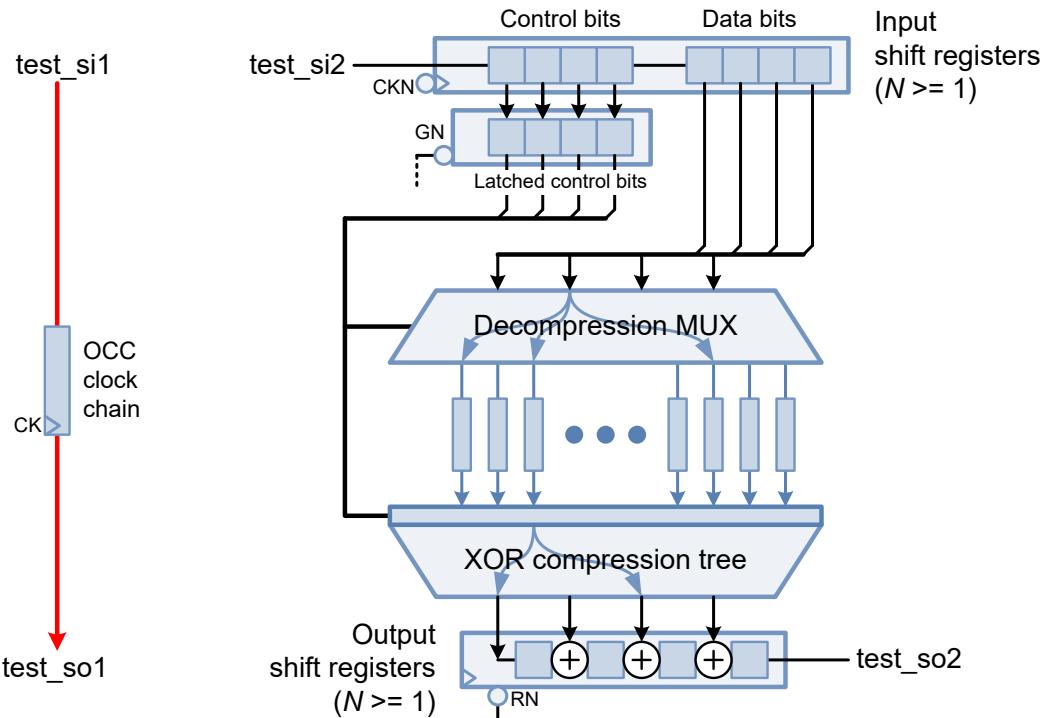
In a streaming compression flow, the clock chain can be external or compressed, as described in the following sections.

For more information on OCC controllers, see [Chapter 12, On-Chip Clocking Support.](#)

External Clock Chain

When you insert DFTMAX Ultra streaming compression along with a DFT-inserted or user-defined OCC controller, the clock chain is driven by dedicated scan-in and scan-out signals by default. This is known as an *external clock chain* because it exists outside the decompressor and compressor.

Figure 426 DFTMAX Ultra Codec With External Clock Chain



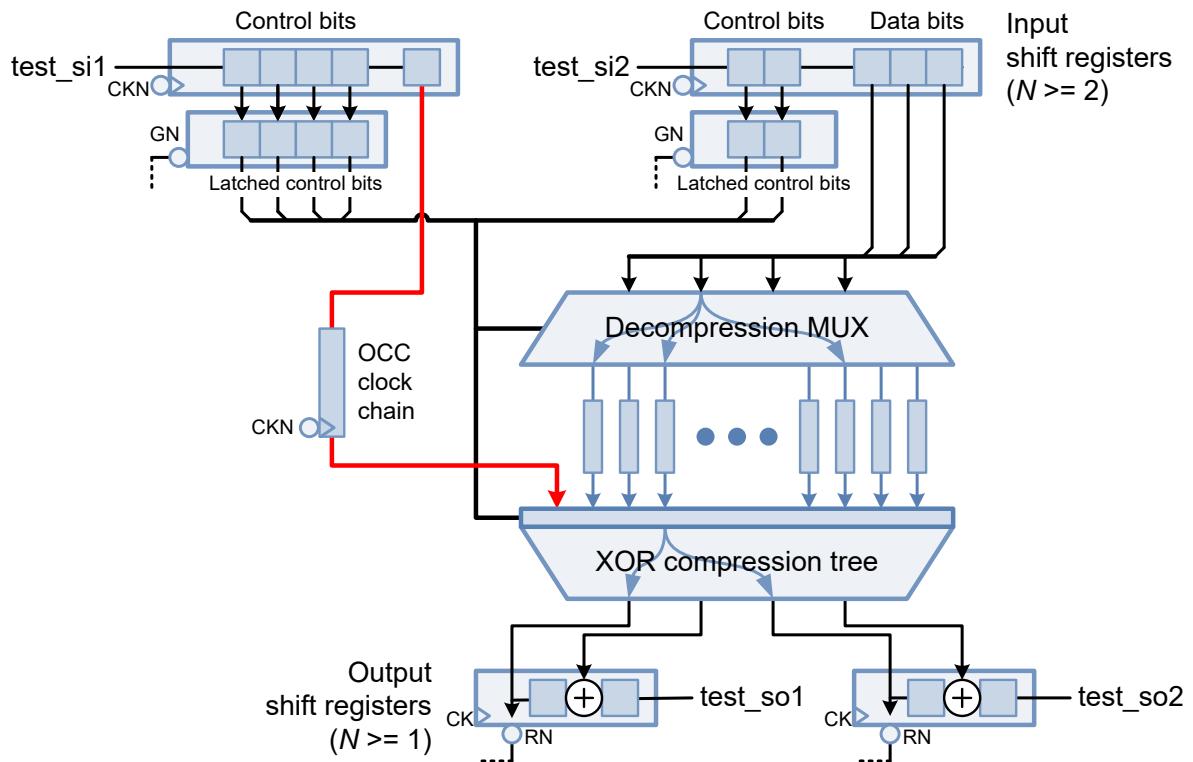
This is the default streaming compression clock chain architecture, even if you do not explicitly configure an external clock chain with the `set_scan_path` command.

In a DFTMAX Ultra design, DFT-inserted clock chains are clocked by the rising clock edge. User-defined clock chains can be clocked by the rising or falling edge.

Compressed Clock Chain

If you have at least two scan inputs, you can optionally include the clock chain in the codec scan paths. (For configuration details, see [Creating Compressed Clock Chains on page 946](#)).

Figure 427 DFTMAX Ultra Codec With Compressed Clock Chain



A dedicated scan input drives only control bits and the clock chain, but no data bits. This ensures that the clock chain does not impose ATPG constraints on other scan cells.

This clock chain architecture is called a *compressed clock chain* because it shares scan-in and scan-out pins with the codec, even though the scan data on the input side is uncompressed.

The data bits are allocated across the remaining input shift registers. At least two codec scan inputs are required for this architecture. There is no requirement for the number of scan outputs.

In this architecture, the clock chain is clocked by the input shift register clock.

Long clock chains (from many clocks or multiple OCC controllers) always use a single scan-in signal; they are not split up to balance them against other compressed chains.

24

Using DFTMAX Ultra Compression

To use DFTMAX Ultra compression, you specify the desired number of compressed chains and the number of I/O ports to be used for scan input and scan output. The tool synthesizes the scan circuitry and writes the architectural information to the SPF file. TestMAX ATPG then generates test patterns for simulation and device testing.

This chapter includes the following topics:

- [DFTMAX Ultra Compression Requirements](#)
- [Top-Down Insertion Compressed Scan Flow](#)
- [Top-Down Insertion Compressed Scan Flow With Partitions](#)
- [The Multiple-Input, Multiple-Output Codec Architecture](#)
- [DFTMAX Ultra Compression and Multiple Test Modes](#)
- [Using OCC Controllers With DFTMAX Ultra Compression](#)
- [Reducing Power Consumption in DFTMAX Ultra Designs](#)
- [Planning, Previewing, and Inserting DFTMAX Ultra Compression](#)
- [Library Cell Requirements for Codec Implementation](#)

DFTMAX Ultra Compression Requirements

To use DFTMAX Ultra scan compression,

- You must have the Design Compiler tool installed and licensed at your site.
- You must have the DFTMAX and DFTMAX Ultra tools, or the TestMAX DFT tool, licensed at your site.
- You must have an HDL-Compiler license for compressed scan insertion.

- You must have the following cell types available for mapping when using the `insert_dft` command:
 - Level-sensitive latch
 - Flip-flop with asynchronous reset
- For more information, see [Library Cell Requirements for Codec Implementation on page 967](#).
- You must use a preclock strobe (which is the default). If you set the `test_default_strobe` variable, ensure that the strobe occurs before the active edges of the test clock waveforms.

Note:

See [Chapter 28, DFTMAX Ultra Flow Naming Conventions](#), for information on the flow naming conventions used for DFTMAX Ultra flows.

Top-Down Insertion Compressed Scan Flow

This topic describes the top-down insertion (TDI-C) flow with DFTMAX Ultra compression. In this flow, you insert scan compression into a design that contains no existing scan compression logic.

This flow is covered in the following topics:

- [Enabling DFTMAX Ultra Compression](#)
- [Configuring the DFTMAX Ultra Codec](#)
- [Configuring the Codec Clock](#)

Enabling DFTMAX Ultra Compression

Commands and command options related to DFTMAX Ultra compression use the word “streaming.” To enable top-down insertion of DFTMAX Ultra compression (TDI-C), simply enable DFTMAX Ultra compression as follows:

```
dc_shell> set_dft_configuration -streaming_compression enable
```

The tool automatically performs TDI-C compression when all of the following are true:

- DFTMAX Ultra compression is enabled and a codec is configured.
- Scanned or scannable logic exists.
- No standard scan or compressed scan s exist.

When these criteria are met, the `preview_dft` and `insert_dft` commands issue the following messages:

```
Information: Detected scanned or scannable logic. (TEST-1462)
Information: Inferring the top-down scan insertion (TDI) flow.
(TEST-1436)
```

[Example 153](#) shows a script that implements DFTMAX Ultra compression using a top-down insertion flow.

Example 153 Script for Top-Down Insertion of DFTMAX Ultra Compression

```
# enable DFTMAX Ultra compression
set_dft_configuration -streaming_compression enable

# specify standard scan chain count
set_scan_configuration -chain_count 2

# configure the DFTMAX Ultra codec
set_streaming_compression_configuration \
    -chain_count 8 -inputs 2 -outputs 2

# configure required scan clock signals
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}

# configure optional placeholder DFT signal ports
set_dft_signal -view spec -type ScanDataIn -port SI
set_dft_signal -view spec -type ScanDataOut -port SO
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type TestMode -port TM
```

When DFTMAX Ultra compression is enabled, the `insert_dft` command inserts compressed scan logic into the design and defines the following two test modes:

- Compressed scan mode

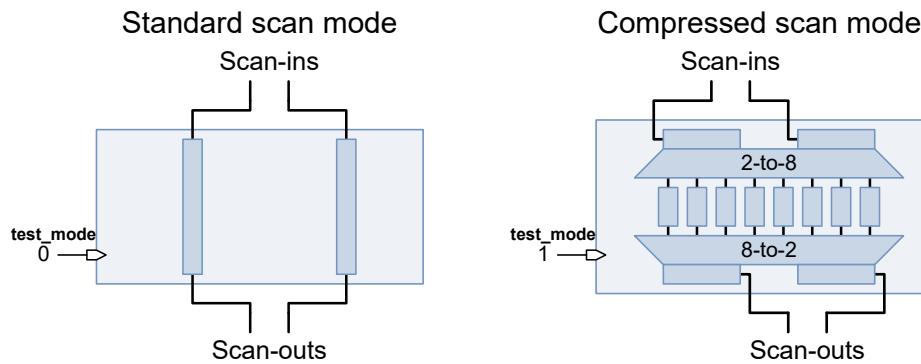
This mode configures the scan elements as short chains accessed through a DFTMAX Ultra codec. The default name for this test mode is `ScanCompression_mode`.

- Standard scan mode

This mode joins the short compressed scan chains to reconfigure them into longer standard scan chains. This is also known as standard scan mode. The default name for this test mode is `Internal_scan`.

These test modes are created automatically during compressed scan insertion; you do not need to create them or reference them. [Figure 428](#) shows the scan structures for the two test modes created by [Example 153](#). Three standard scan chains and eight compressed scan chains are created.

Figure 428 Standard Scan and Compressed Scan Modes



At least one test-mode signal is required to select between standard scan mode and compressed scan mode. If a `TestMode` signal is defined with the `set_dft_signal` command, it is used for mode selection. If no test-mode signals are defined, a test-mode port is created and used. Test-mode encodings are created that map the test-mode signal values to each scan mode.

Note:

For more information about working with multiple test modes in DFT Compiler, including information on specifying test-mode encodings, see [Multiple Test Modes on page 357](#).

A compressed scan mode is always associated with a corresponding standard scan mode. The standard scan mode associated with a compressed scan mode is known as its *base mode*. The base mode controls aspects of scan configuration that are common to both modes, such as scan I/O port definitions, scan signal hookup pin definitions, and top-level test access structures.

Configuring the DFTMAX Ultra Codec

The `set_streaming_compression_configuration` command configures aspects of the DFTMAX Ultra compressed scan mode, just as the `set_scan_configuration` command configures aspects of the standard scan mode.

Use one of the following options of the `set_streaming_compression_configuration` command to configure the compression architecture of the codec:

- `-compressed_max_length chain_length`

The `-compressed_max_length` option specifies the maximum number of *shift cycles* of the entire scan compression path (from decompressor inputs to compressor outputs). The tool adjusts the shift register and compressed chain lengths together to find an optimal compression architecture that meets this constraint.

Note:

The shift cycle count is *not* simply the sum of the input shift register, compressed chain, and output shift register lengths. See [SolvNet article 2151939, "How Do I Determine the Shift Cycle Count of My Scan, DFTMAX, or DFTMAX Ultra Design?"](#).

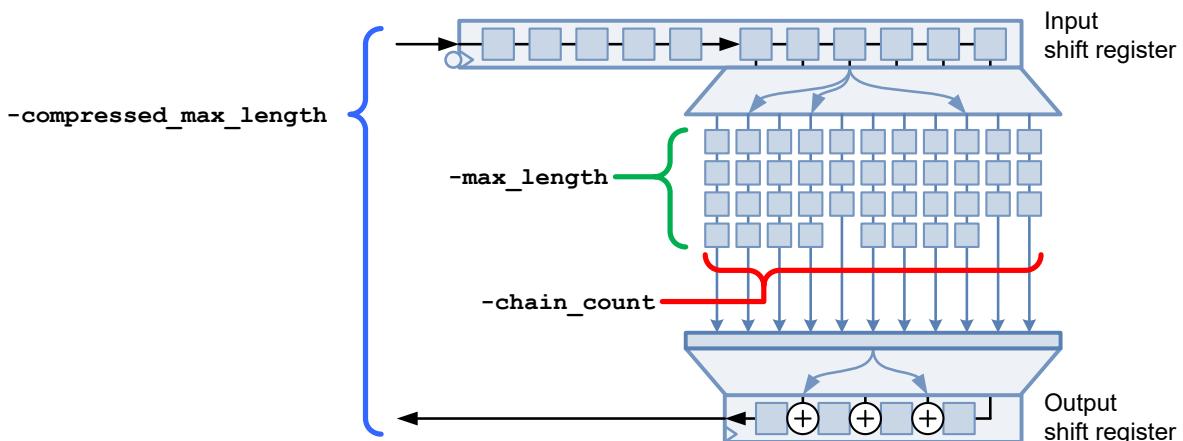
- `-max_length chain_length`

The `-max_length` option specifies the maximum allowed length of the compressed scan chains only (not including the codec shift registers). The tool creates the number of compressed scan chains needed to meet this requirement.

- `-chain_count chain_count`

The `-chain_count` option specifies the number of compressed scan chains. The tool adjusts the compressed scan chain lengths to meet this requirement.

Figure 429 Streaming Compression Codec Configuration Options



Specify only one of these options. If no option is specified, the DFT architect aborts with an error. If multiple options are specified, precedence applies as described in the man page.

DFTMAX Ultra compression automatically determines the following aspects of codec architecture:

- It computes the input and output shift register lengths that are optimal for the number of scan inputs and outputs and the number of compressed scan chains.
- If multiple scan clocks exist, it uses the clock that minimizes the number of lock-up latches. For more information, see [Configuring the Codec Clock on page 932](#).

By default, a compressed scan mode inherits and uses the same scan I/Os as its base mode. To specify the number of scan I/Os to use for the codec in compression mode, use the `-inputs` and `-outputs` options of the `set_streaming_compression_configuration` command.

Configuring the Codec Clock

By default, the tool selects clocks for the decompressor and compressor that minimize the number of lock-up latches at the decompressor outputs and compressor inputs. The selection rules are as follows:

- The decompressor uses a clock whose trailing edge is as late or later than all head scan elements.
- The compressor uses a clock whose leading edge is as early or earlier than all tail scan elements.
- If multiple clocks meet these criteria for the decompressor or compressor, the dominant clock across the head or tail scan elements is used, respectively.

You can determine which clocks are selected for the decompressor and compressor by looking at the codec information reported by the `preview_dft` or `insert_dft` commands. For example,

```
Architecting Streaming Decompressor
Number of inputs = 1
Maximum size per input = 80
Decompressor Clock = CLK2
Architecting Streaming Compressor
Number of outputs = 1
Maximum size per output = 67
Compressor Clock = CLK2
Architecting Load Decompressor (version 5.8)
Number of inputs/chains/internal modes = 80/70/4
Architecting Unload compressor (version 5.8)
Number of outputs/chains = 67/70
Information: Compressor will have 100% x-tolerance
```

In most designs, the decompressor and compressor clocks are the same. If they differ, the tool issues the following information message:

Information: Different clocks are chosen for the streaming codec decompressor and compressor clocks. (TEST-1480)

To exclude one or more clocks from automatic clock selection, use the `-exclude_clocks` option of the `set_streaming_compression_configuration` command:

```
dc_shell> set_streaming_compression_configuration -exclude_clocks {IPCLK}
```

To specify a particular scan clock for the codec (both decompressor and compressor), use the `-clock` option of the `set_streaming_compression_configuration` command:

```
dc_shell> set_streaming_compression_configuration -clock CLK2
```

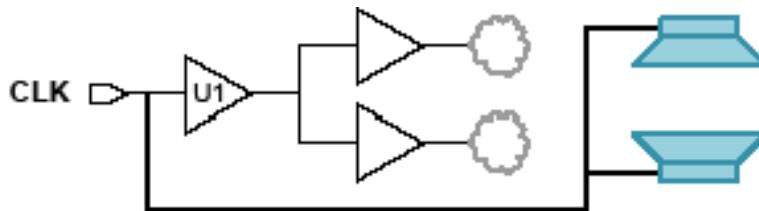
To specify a particular scan clock for the decompressor or compressor, use the `-decompressor_clock` or `-compressor_clock` option, respectively:

```
dc_shell> set_streaming_compression_configuration \
    -decompressor_clock CLK1 \
    -compressor_clock CLK3
```

A referenced scan clock must be previously defined as a scan clock using the `set_dft_signal -type ScanClock` command.

By default, DFT Compiler makes the codec clock connections at the source port specified in the `-view existing_dft` signal definition, as shown in [Figure 430](#).

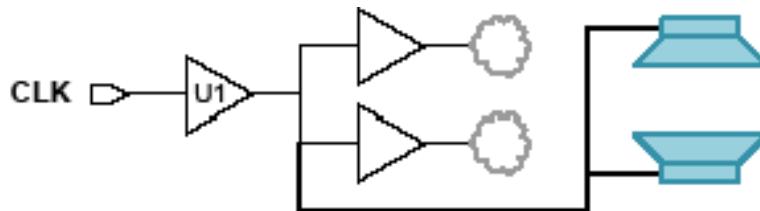
Figure 430 Default Codec Clock Connection



```
set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port CLK
```

However, if you want DFT Compiler to make the clock connection at an internal pin, such as a pad cell or clock buffer output, you can specify it with the `-hookup_pin` option in a subsequent `-view spec` signal definition, as shown in [Figure 431](#).

Figure 431 User-Defined Codec Clock Connection



```
set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port CLK
set_dft_signal -view spec -type ScanClock \
    -port CLK -hookup_pin U1/Z
```

For more information on specifying a clock hookup pin, see [Specifying a Hookup Pin for DFT-Inserted Clock Connections on page 246](#).

Top-Down Insertion Compressed Scan Flow With Partitions

In the DFTMAX Ultra compression top-down insertion flow, you can use the DFT partitions feature to divide the design into multiple partitions. The tool inserts a separate codec for each partition. This is known as the TDI-C-P flow.

This flow is described in the following topics:

- [Using Dedicated Scan Data Connections for Each Partition](#)
- [Using Serial Scan Data Connections Between Partitions](#)
- [Per-Partition Streaming Configuration Commands](#)

See Also

- [Partitioning a Scan Design With DFT Partitions on page 285](#) for general information about DFT partitions

Using Dedicated Scan Data Connections for Each Partition

When creating DFT partitions, dedicated scan data connections are created for each partition by defining the scan data DFT signals within each partition. In [Example 154](#), scan-in and scan-out signals are created inside each DFT partition.

Example 154 Script for Top-Down Insertion Flow With Partitions and Dedicated Scan Data Connections

```
# enable streaming compression in global configuration
# (before any DFT partitions are defined)
set_dft_configuration -streaming_compression enable

# define and configure a user-defined DFT partition
define_dft_partition my_part -include {...}
current_dft_partition my_part
set_scan_configuration -chain_count 1
set_streaming_compression_configuration -chain_count 4
set_dft_signal -port SI1 -type ScanDataIn
set_dft_signal -port SO1 -type ScanDataOut

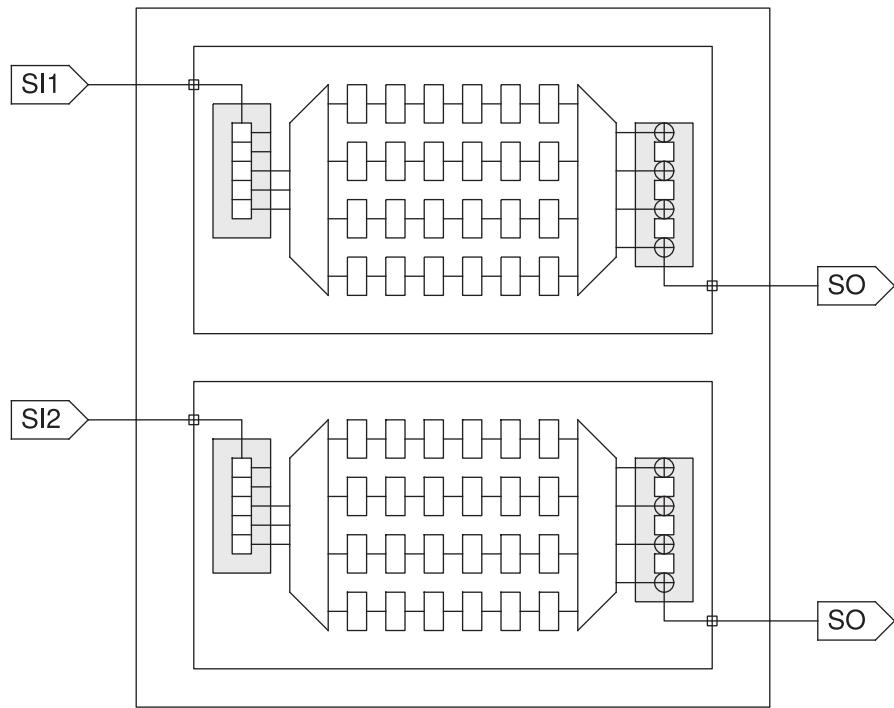
# define and configure the default DFT partition
current_dft_partition default_partition
set_scan_configuration -chain_count 1
set_streaming_compression_configuration -chain_count 4
set_dft_signal -port SI2 -type ScanDataIn
set_dft_signal -port SO2 -type ScanDataOut
```

When the `set_dft_signal` command is used inside a partition definition, it includes the partition name in the resulting message:

Accepted dft signal specification for **partition 'my_part'** and modes:
`all_dft`

The script in [Example 154](#) results in the scan data connections shown in [Figure 432](#). Each DFT partition has dedicated scan-in and scan-out connections.

Figure 432 Top-Down Insertion Flow With Partitions and Dedicated Scan Data Connections



This is the default method for scan data connections. If you do not explicitly define scan data signals with the `set_dft_signal` command, dedicated scan data signals are created as needed for each partition.

Using Serial Scan Data Connections Between Partitions

When creating DFT partitions, serial scan data connections between the partition codecs are created by defining the scan data DFT signals with the `define_dft_signal -partition all` command. In [Example 154](#), global scan-in and scan-out signals are created for all DFT partitions.

Example 155 Script for Top-Down Insertion Flow With Partitions and Serial Scan Data Connections

```
# enable streaming compression in global configuration
# (before any DFT partitions are defined)
set_dft_configuration -streaming_compression enable

set_dft_signal -port SI1 -type ScanDataIn -partition all
set_dft_signal -port SO1 -type ScanDataOut -partition all
```

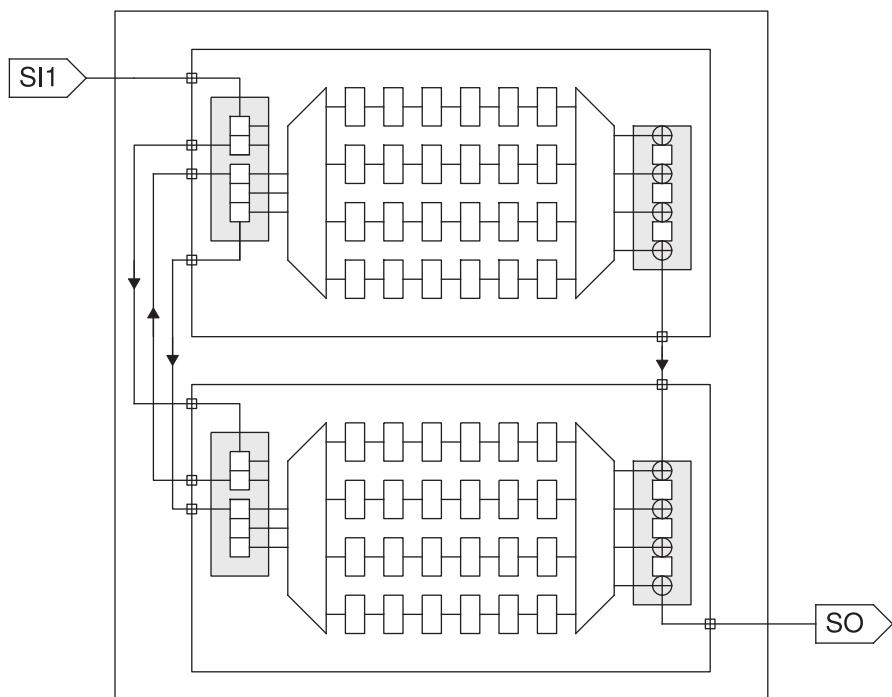
Chapter 24: Using DFTMAX Ultra Compression
Top-Down Insertion Compressed Scan Flow With Partitions

```
# define and configure a user-defined DFT partition
define_dft_partition my_part -include {...}
current_dft_partition my_part
set_scan_configuration -chain_count 1
set_streaming_compression_configuration -chain_count 4

# define and configure the default DFT partition
current_dft_partition default_partition
set_scan_configuration -chain_count 1
set_streaming_compression_configuration -chain_count 4
```

The script in [Example 155](#) results in the scan data connections shown in [Figure 433](#). The partition codecs are connected using serial scan data connections.

Figure 433 Top-Down Insertion Flow With Partitions and Serial Scan Data Connections



All static control shift registers are stitched into the scan chain first, followed by all dynamic input data shift registers. This ensures that all control register bits are shifted in last. The output shift registers are stitched together, with the second and subsequent output shift registers incorporating the output of the previous register into their XOR shift chain.

This connection method has the following requirements:

- You must use only one scan-in and one scan-out signal. You cannot use multiple scan-in or scan-out signals.
- You must explicitly define the scan data signals with the `set_dft_signal -partition all` command. You cannot use this connection method with automatically created scan data signals.

Per-Partition Streaming Configuration Commands

This topic lists the commands you can use to configure DFTMAX Ultra streaming compression on a per-partition basis. Streaming compression commands and options not listed in this section should be applied as part of the global DFT configuration settings.

See Also

- [Per-Partition Scan Configuration Commands on page 289](#) for the per-partition commands that are not specific to the streaming compression flow.

set_streaming_compression_configuration

The following `set_streaming_compression_configuration` options can be specified on a per-partition basis:

- `-inputs`
- `-outputs`
- `-compressed_max_length`
- `-max_length`
- `-chain_count`
- `-clock`
- `-min_power`
- `-shift_power_chain_length`
- `-shift_power_chain_ratio`
- `-shift_power_clock`
- `-shift_power_disable`

set_dft_signal

The following `set_dft_signal` options can be specified on a per-partition basis:

- `-type ScanEnable -usage streaming_codec`

Note:

The `streaming_codec` usage is the only scan-enable usage that supports per-partition specification. Do not include other usages when defining per-partition scan-enable signals.

The Multiple-Input, Multiple-Output Codec Architecture

The multiple-input, multiple-output codec architecture uses multiple scan data signals for a single codec to increase the scan data shifting throughput.

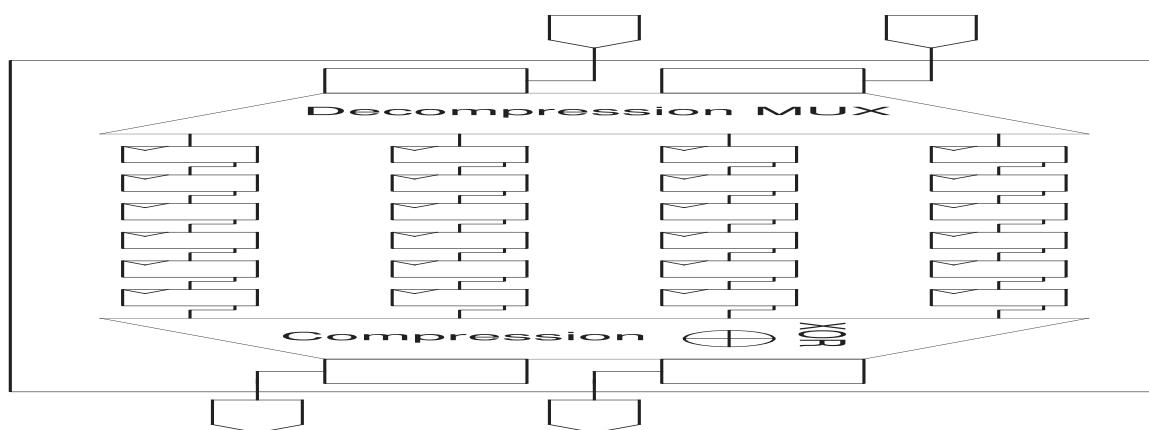
To implement this architecture, specify a standard chain count value larger than one. For example,

```
# use two scan-ins/scan-outs in standard scan and compressed scan modes
set_scan_configuration -chain_count 2

# create four compressed scan chains
set_streaming_compression_configuration -chain_count 4
```

This creates two scan chains in standard scan mode, but it also provides two scan-in and scan-out signals in the compressed scan mode. The tool splits the codec input and output shift registers across the available scan-ins and scan-outs, as shown in [Figure 434](#).

Figure 434 The Multiple-Input, Multiple-Output Codec Architecture



If you want to specify a different number of scan connections for standard scan mode and compressed scan, use the `-inputs` and `-outputs` options of the `set_streaming_compression_configuration` command to specify the number of scan connections in compressed scan mode:

```
# use eight scan-ins/scan-outs in standard scan mode
set_scan_configuration -chain_count 8

# use two scan-ins/scan-outs in compressed scan mode
set_streaming_compression_configuration \
    -inputs 2 -outputs 2 -chain_count 4
```

You can use the multiple-input, multiple-output architecture with DFT partitions. Apply the scan configuration commands within each partition definition.

See Also

- [Multiple-Input, Multiple-Output Architecture on page 922](#) for more information about this architecture
- [DFT Synthesis Limitations on page 981](#) for a list of requirements and limitations of this architecture

DFTMAX Ultra Compression and Multiple Test Modes

You invoke DFTMAX Ultra compression by setting the DFT configuration to scan compression and specifying the streaming scan compression configuration:

```
set_dft_configuration -streaming_compression enable
set_streaming_compression_configuration ...
```

When you insert DFTMAX Ultra compressed scan into your design, the tool creates two test modes by default:

- A standard scan mode

The default name for this test mode is `Internal_scan`.

- A DFTMAX Ultra compressed scan mode

The default name for this test mode is `ScanCompression_mode`.

Just as you can create multiple standard scan modes with standard scan, you can also create multiple compressed scan modes with DFTMAX Ultra compression. This capability uses the same multiple test-mode creation, configuration, and reporting commands as used with multiple standard scan modes.

Usage of multiple compressed scan modes is described in the following topics:

- [Defining Multiple DFTMAX Ultra Compressed Scan Modes](#)
- [Mixing DFTMAX and DFTMAX Ultra Compression Modes](#)
- [Per-Test-Mode Streaming Configuration Options](#)

See Also

- [Multiple Test Modes on page 357](#) for more information about defining multiple test modes

Defining Multiple DFTMAX Ultra Compressed Scan Modes

You can define user-defined test modes with the `define_test_mode` command. Define streaming compression modes with the `streaming_compression` usage. For example,

```
dc_shell> define_test_mode COMP -usage streaming_compression
```

Note:

For backward compatibility, you can also define streaming compression modes with the `scan_compression` usage, but only if no DFTMAX compression modes also exist in the design. This behavior will be obsoleted in a future release.

You can define and use more than one compressed scan mode for a device. For example, you might create one compressed scan mode to use for wafer sort testing and another for class testing of finished devices. When you define multiple compressed scan modes, the tool creates circuitry to support all such modes in the device.

TestMAX ATPG can select the testing mode by forcing one or more control inputs to specified values. [Example 156](#) shows a script that defines two compressed scan modes and one standard scan mode, together with the encoding to select the three test modes.

Example 156 Defining Two Compressed Scan Modes and One Standard Scan Mode

```
set_dft_configuration -streaming_compression enable

set_dft_signal -type TestMode -port {TM0 TM1}
set_dft_signal -port SI -type ScanDataIn
set_dft_signal -port SO -type ScanDataOut
set_dft_signal -view spec -port SE -type ScanEnable

define_test_mode SCAN -usage scan \
    -encoding {TM0 1 TM1 1}
define_test_mode COMP1 -usage streaming_compression \
    -encoding {TM0 0 TM1 1}
define_test_mode COMP2 -usage streaming_compression \
    -encoding {TM0 1 TM1 0}
```

```
set_scan_configuration -test_mode SCAN -chain_count 1

set_streaming_compression_configuration \
    -test_mode COMP1 -base_mode SCAN \
    -chain_count 80

set_streaming_compression_configuration \
    -test_mode COMP2 -base_mode SCAN \
    -chain_count 40
```

Subsequent commands in the DFT synthesis flow use the `-test_mode` option to specify the test mode to which the command applies. For example,

```
set_streaming_compression_configuration -test_mode COMP1 ...
set_streaming_compression_configuration -test_mode COMP2 ...
set_scan_path -test_mode COMP1 ...
set_scan_path -test_mode COMP2 ...
```

For information on how to order global and mode-specific configuration commands in your scripts, see [Recommended Ordering of Global and Mode-Specific Commands on page 365](#).

In a hierarchical design, each lower-level block can itself have multiple test modes defined. By default, all combinations of lower-level test modes blocks are selected from the top level. However, if only certain combinations of lower-level test modes are needed, you can specify which combinations of lower-level test modes are selected from the top level.

See Also

- [Using Multiple Test Modes in Hierarchical Flows on page 978](#) for more information about defining multiple test modes in hierarchical DFTMAX Ultra flows

Mixing DFTMAX and DFTMAX Ultra Compression Modes

You can mix DFTMAX and DFTMAX Ultra compression modes in the same design. However, note the following requirements:

- Both compression types cannot be active in the same test mode.
- User-defined DFTMAX Ultra test modes must be defined with the `streaming_compression` usage so that the tool can differentiate them from DFTMAX test modes defined with the `scan_compression` usage.

[Example 157](#) shows a top-down insertion (TDI) flow that configures three compression modes: a standard scan mode, a DFTMAX compression mode, and a DFTMAX Ultra compression mode.

Example 157 Script Example With DFTMAX and DFTMAX Ultra Compression

```
# enable the DFTMAX and DFTMAX Ultra compression clients
set_dft_configuration \
    -scan_compression enable \
    -streaming_compression enable

# apply global DFT configuration
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}
set_dft_signal -view spec -type ScanEnable -port SE
set_scan_configuration -clock_mixing mix_clocks

# define the test modes
define_test_mode SCAN -usage scan
define_test_mode DFTMAX -usage scan_compression
define_test_mode DFTMAX_ULTRA -usage streaming_compression ;# note usage

# configure each test mode
set_scan_configuration -test_mode SCAN -chain_count 4
set_scan_compression_configuration \
    -test_mode DFTMAX -base_mode SCAN \
    -chain_count 20
set_streaming_compression_configuration \
    -test_mode DFTMAX_ULTRA -base_mode SCAN \
    -chain_count 80
```

For information on integrating cores with DFTMAX and DFTMAX Ultra compression modes, see [Mixing DFTMAX and DFTMAX Ultra Compression Core Modes](#).

Per-Test-Mode Streaming Configuration Options

The following `set_streaming_compression_configuration` options can be applied to specific test modes:

- `-inputs`
- `-outputs`
- `-compressed_max_length`
- `-max_length`
- `-chain_count`
- `-base_mode`
- `-clock`
- `-min_power`
- `-shift_power_chain_length`

- `-shift_power_chain_ratio`
- `-shift_power_clock`
- `-shift_power_disable`

Note:

Although the `set_streaming_compression_configuration` command applies to the current test mode by default, the `-test_mode` option is typically used together with the `-base_mode` option so that the relationship between the test mode and base mode is explicitly highlighted.

Using OCC Controllers With DFTMAX Ultra Compression

The following topics describe how to use OCC controllers with DFTMAX Ultra compression:

- [Creating External Clock Chains](#)
- [Creating Compressed Clock Chains](#)
- [OCC Controllers and Streaming Codec Scan-Enable Constraints](#)

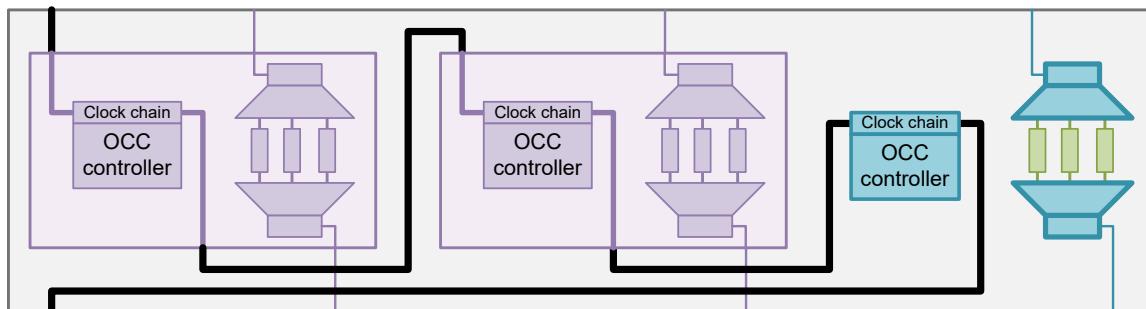
See Also

- [DFTMAX Ultra Architectures for On-Chip Clocking \(OCC\) on page 924](#) for architecture details

Creating External Clock Chains

By default, DFTMAX Ultra compression creates external clock chains so that the control bits are directly controllable without conflict by ATPG. [Figure 435](#) shows an external clock chain structure for two core-level OCC controllers and a top-level OCC controller.

Figure 435 External Clock Chain in a DFTMAX Ultra Design



In a DFTMAX Ultra design, DFT-inserted external clock chains are clocked by the rising clock edge. User-defined clock chains can be clocked by the rising or falling edge.

You can create external clock chains automatically or manually, as described in the following topics:

- [Automatically Creating External Clock Chains](#)
- [Manually Specifying External Clock Chains](#)

Automatically Creating External Clock Chains

By default, DFTMAX Ultra builds a single external (uncompressed) clock chain for all available core-level and top-level clock chains. The tool uses existing scan-in and scan-out signals, creating new signals as needed.

For top-level user-defined OCC controllers, define the existing clock chain segments with the `set_scan_group` command; the tool automatically includes these segments in its clock chain.

Note the following:

- To build multiple clock chains, you must manually specify the external clock chains.
- Automatic creation of external clock chains applies to all test modes created for the design, not just DFTMAX Ultra compression modes.

If you have DFTMAX-only cores with compressed clock chains, the tool does not include these compressed clock chains in the automatically created external clock chain. These compressed clock chains operate normally when the core is active in its DFTMAX mode.

Manually Specifying External Clock Chains

To manually define the complete external clock chain for special cases, you can use the `set_scan_path` command with the `-class occ` option. This method allows you to use specific scan-in and scan-out signals for the clock chain. It also allows you to concatenate multiple clock chains in a specific order.

For example,

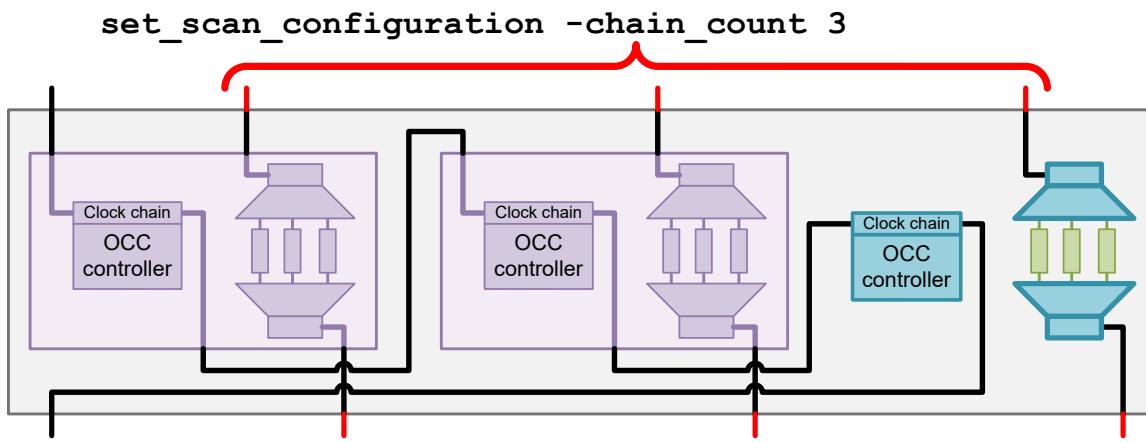
```
set_dft_signal -view spec -type ScanDataIn -port OCC_SI
set_dft_signal -view spec -type ScanDataOut -port OCC_SO
set_scan_path \
  MY_clock_chain -class occ \
  -include_elements {\ \
    snps_clk_chain_2/clock_chain \
    CORE1/clock_chain_name \
    CORE2/clock_chain_name} \
  -complete true \
  -scan_data_in OCC_SI -scan_data_out OCC_SO \
  -test_mode all
```

For details on defining external clock chains, see [Defining External Clock Chains on page 689](#).

Budgeting Scan I/Os and External Clock Chains

When you use OCC controllers in a DFTMAX Ultra flow, the tool does not include external clock chains in the `set_scan_configuration -chain_count` value. [Figure 436](#) shows an example.

Figure 436 External Clock Chain Excluded From `set_scan_configuration -chain_count` Value



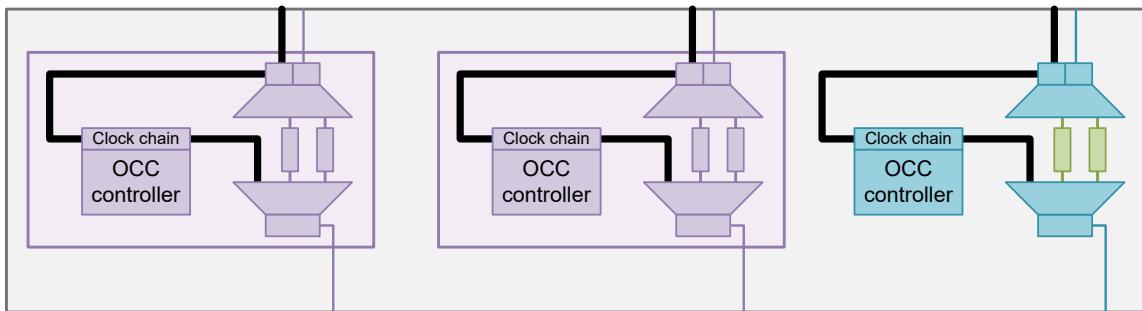
Note the following:

- This behavior is different than non-DFTMAX Ultra flows, which include the clock chains in the value.
- This behavior affects all test modes created for the design, not just the DFTMAX Ultra compression modes.

Creating Compressed Clock Chains

You can optionally configure the tool to implement compressed (decompressor-driven) clock chains. [Figure 437](#) shows compressed clock chain structures for two core-level OCC controllers and a top-level OCC controller.

Figure 437 Compressed Clock Chains in a DFTMAX Ultra Design



To implement a compressed clock chain, specify the following option when configuring the codec:

```
dc_shell> set_streaming_compression_configuration \
           -external_clock_chain false
```

In addition, you must also provide at least two codec scan inputs:

```
dc_shell> set_streaming_compression_configuration -inputs 2 ;# or more
```

There is no requirement for the number of scan outputs.

Additional limitations apply when using compressed clock chains in DFTMAX Ultra designs. See [Chapter 26, DFTMAX Ultra Limitations and Known Issues](#).

See Also

- [Compressed Clock Chain on page 925](#) for architecture details

OCC Controllers and Streaming Codec Scan-Enable Constraints

For proper operation, scan-enable signals that drive streaming codecs must be de-asserted during capture.

In a DFTMAX Ultra design without OCC controllers, the tool creates a test protocol that constrains only streaming codec scan-enable signals, as described in [Scan-Enable Signal Requirements for Codec Operation on page 922](#).

OCC controllers also require that their scan-enable signals be de-asserted during capture. When OCC controllers are present in a DFTMAX Ultra design, the tool constrains all scan-enable signals except those defined with a usage of `clock_gating`, as described in [Scan-Enable Signal Requirements for OCC Controller Operation on page 530](#).

Reducing Power Consumption in DFTMAX Ultra Designs

You can reduce the power consumption of designs with streaming compression by using the following features:

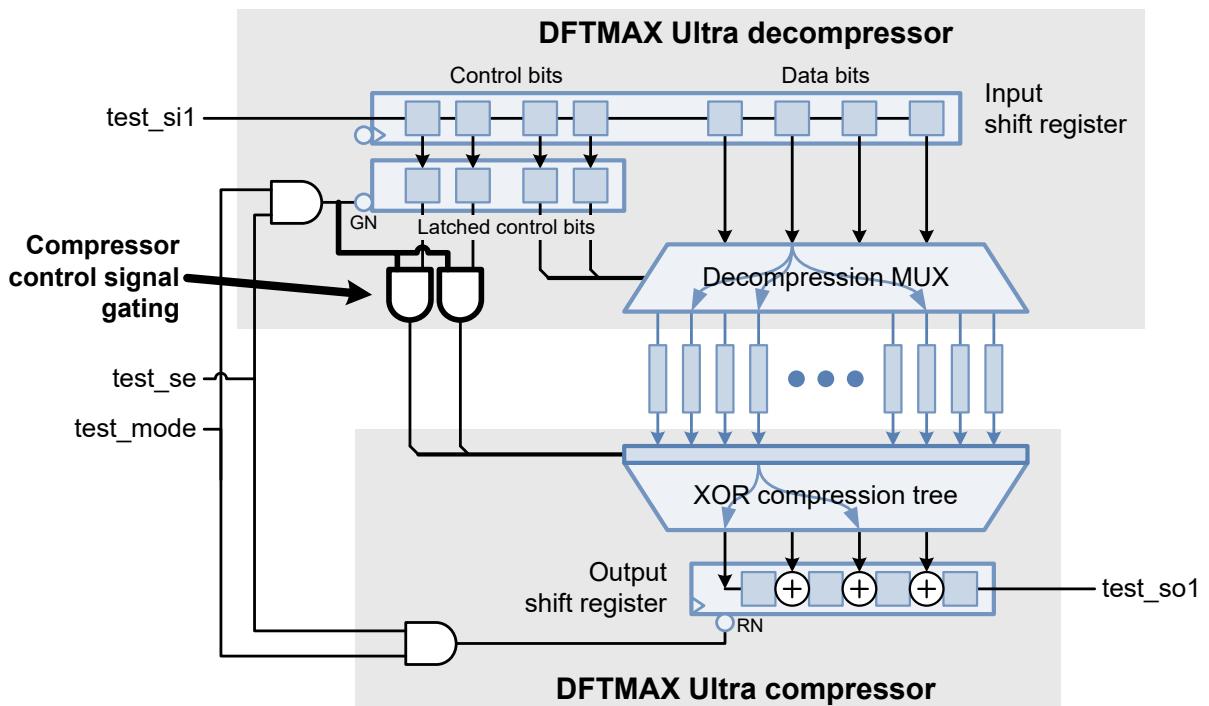
- Reducing Compressor Power When Codec Is Inactive
- Reducing Scan Shift Power Using Shift Power Groups

Reducing Compressor Power When Codec Is Inactive

In a streaming compression architecture, an XOR compression tree combines the shift outputs from all compressed chains into a reduced set of scan data that is captured by the output register. This XOR compression tree is needed only during scan shifting in that codec's compressed scan mode. At other times, the compression logic is not needed, but it will still toggle when the tail scan flip-flops of the compressed chains toggle. This is a particular concern during mission mode, when the flip-flops are clocked at their full operating frequency.

To address this, the tool can insert logic that enables the streaming compressor only when needed, as shown in [Figure 438](#).

Figure 438 Example of a Streaming Compressor With Compressor Gating



The control register signals to the compressor are enabled only when the codec shifts data in its test mode. At all other times, the control signals are held at logic 0, which causes the X-blocking logic in the compressor to block the toggle activity of the tail scan cells from propagating into the compressor logic.

To enable compressor XOR gating, specify the following option:

```
dc_shell> set_streaming_compression_configuration -min_power true
```

When enabled, this feature adds one AND gate for each compressor control signal (but not the decompressor control signals).

Reducing Scan Shift Power Using Shift Power Groups

You can use shift power groups to reduce power consumption during scan shift. This feature is described in the following topics:

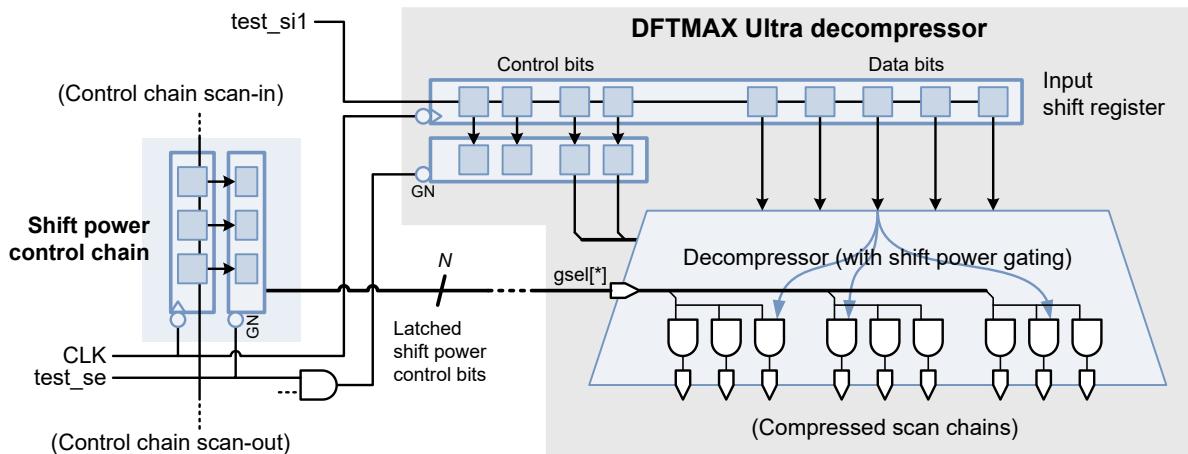
- [The Shift Power Groups Architecture](#)
- [Configuring Shift Power Groups](#)
- [Integrating Cores With Shift Power Groups in Hierarchical Flows](#)
- [Configuring Shift Power Groups in TestMAX ATPG](#)
- [Using Shift Power Groups With Other DFT Features](#)
- [Limitations of Shift Power Groups](#)

The Shift Power Groups Architecture

During scan shift, there is significant toggle activity in the scan chains. At high scan shift frequencies, this can result in higher-than-desired shift power consumption.

The shift power groups feature helps reduce power consumption during scan shift in DFTMAX Ultra compressed scan modes. This feature inserts AND gates at the decompressor outputs before each compressed scan chain. The chains are gated in groups that are controlled by a shift power control (SPC) chain, as shown in [Figure 439](#).

Figure 439 Shift Power Groups Decompressor Architecture



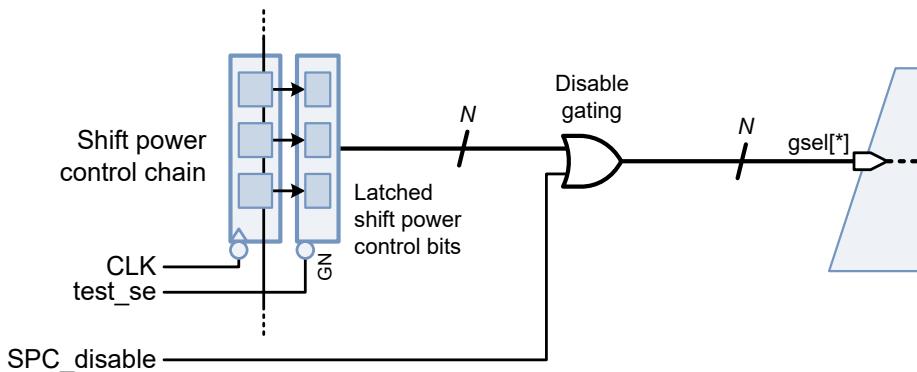
The SPC chain is not part of the control chain. Instead, it is an external (uncompressed) chain outside the DFTMAX Ultra codec. When scan-in completes, the SPC registers contain the group mask values for the next pattern. The de-asserted scan-enable signal, test_se, latches these bits into shadow latches that retain the mask values for scan-in of the next pattern.

TestMAX ATPG configures the group masking in each pattern, depending on the power constraints and the number of care bits in each chain group. The larger number of short chains inherent to scan compression provide finer granularity for this control. Masked groups load constant values into their chains, which reduces overall toggle activity.

SPC chains cannot be compressed because a compressed SPC chain would gate itself, preventing it from reliably loading in each pattern.

The shift power logic also includes a hardware disable signal that, when asserted, disables the shift power logic by enabling all compressed chains, as shown in Figure 440. This signal must be de-asserted or asserted prior to DRC, depending on whether the shift power groups feature is enabled in TestMAX ATPG or not, respectively.

Figure 440 Shift Power Disabling Logic



Configuring Shift Power Groups

To configure the shift power groups feature, do the following:

1. Enable the shift power groups feature.

```
dc_shell> set_streaming_compression_configuration \
           -shift_power_groups true
```

2. Specify the configuration of the compressed chain groups.

- To directly specify the number of compressed chain groups, and therefore the length of the SPC chain, use the `-shift_power_chain_length` option:

```
dc_shell> set_streaming_compression_configuration \
           -shift_power_chain_length 16
```

- To specify the number of compressed chains in each group, which makes the SPC chain length a function of the compressed chain count, use the `-shift_power_chain_ratio` option:

```
dc_shell> set_streaming_compression_configuration \
           -shift_power_chain_ratio 12
```

These options are mutually exclusive.

The default is to include three compressed chains in each group, while still limiting the SPC chain length to the maximum chain length in the design.

3. Define the shift power groups disable signal.

```
dc_shell> set_dft_signal -view spec -type TestControl \
           -port SPC_DISABLE
```

```
dc_shell> set_streaming_compression_configuration \
           -shift_power_disable SPC_DISABLE
```

You can define the disable signal using the `-port` and/or `-hookup_pin` options of the `set_dft_signal` command. For an “internal pins” hookup pin, you must use a test_setup protocol that de-asserts the disable signal.

4. Configure the shift power control chain.

- If no OCC controllers (DFT-inserted or user-defined) are configured in the current design, you must configure an external SPC chain.

Specify the scan-in and scan-out signals to use for the SPC chain:

```
dc_shell> set_scan_path SPC -class spc \
           -scan_data_in SPC_IN \
           -scan_data_out SPC_OUT \
           -test_mode all
```

You do not need to specify SPC scan path elements; the SPC chain is automatically included in the specification.

- If OCC controllers (DFT-inserted or user-defined) are configured in the current design, you must explicitly configure an external clock chain:

```
dc_shell> set_scan_path OCC -class occ \
           -scan_data_in OCC_IN \
           -scan_data_out OCC_OUT \
           -test_mode all ;# includes the SPC chain too
```

You cannot use the default tool-created external clock chain when using SPC.

By default, the tool automatically includes the SPC chain in the clock chain. It is clocked by the ATE clock unless specified otherwise with the `-shift_power_clock` option of the `set_streaming_compression_configuration` command.

Alternately, you can explicitly define a separate external SPC chain as previously described, which provides independent access to the OCC and SPC chains when the core is integrated.

5. (Optional) Configure the shift power clock.

To specify a particular clock for the SPC chain, use the `-shift_power_clock` option:

```
dc_shell> set_streaming_compression_configuration \
           -shift_power_clock CLK
```

The default is to use the ATE clock in OCC flows and the decompressor clock in non-OCC flows.

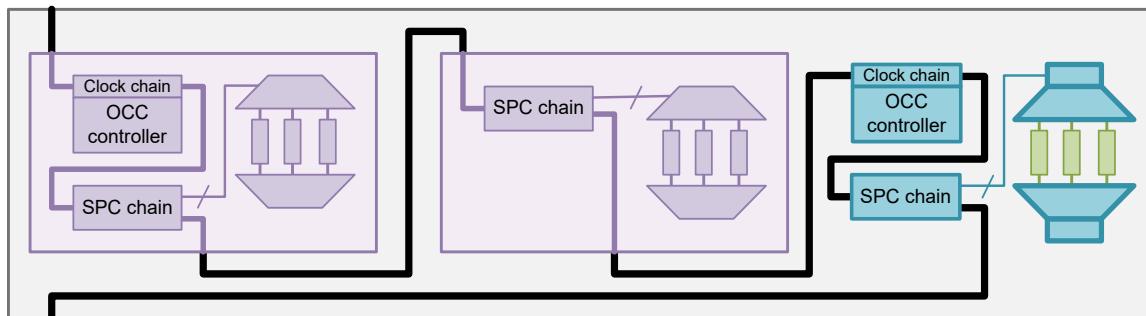
Integrating Cores With Shift Power Groups in Hierarchical Flows

This topic describes how to integrate cores with shift power groups.

Configuring the Control Chain for Shift Power Groups Cores

When you integrate cores that use shift power groups, you must define a top-level external control chain that includes all core-level and top-level clock chains and/or SPC chains, as shown in [Figure 441](#).

Figure 441 External Control Chain in a Shift Power Groups Design



Use the `set_scan_path` command to define the top-level external control chain as follows:

- If any core-level or top-level clock chains exist or will be inserted, then define the external chain using the `-class occ` option.
- If only core-level or top-level SPC chains exist or will be inserted, then define the external chain using the `-class spc` option.
- All core-level clock chains and SPC chains *must be explicitly included in the specification* using the `-include_elements` option. They are not automatically included.
- All top-level clock chains *must be explicitly included in the specification* using the `-include_elements` option. They are not automatically included.
- Top-level SPC chains *are automatically included* in the external chain.

The following example includes core-level clock chains and SPC chains along with top-level clock chains and SPC chains:

```
set_scan_path clock_chain -class occ \
    -include_elements { \
        core1/SPC \
        core2/SPC \
        coreOCC1/OCC \
        coreOCC2/OCC \
        snps_clk_chain_2/clock_chain} \
    -complete_true \
    -scan_data_in OCC_SI \
    -scan_data_out OCC_SO \
    -test_mode all
# (the top-level SPC chain is automatically included)
```

If you concatenate external control chains from pipelined cores, those cores must be created with beginning and ending retiming registers to avoid edge-related concatenation issues at the top level. See the retiming register information in [Using Shift Power Groups With Other DFT Features on page 955](#).

Connecting Core-Level Shift Power Disable Signals

When integrating cores that contain shift power groups, you must manually connect core-level shift power disable signals to a top-level disable signal.

You can use one of the following methods:

- Include preexisting connections to the cores in your top-level RTL.
- Use ECO commands, such as `disconnect_net` and `connect_pin`, to make the connections to the cores.

You can share a single disable signal or use multiple disable signals.

All shift power disable signals must be *de-asserted* (set to logic 0) to *enable* the shift power logic. The DFT-created disable signal for a top-level codec is already de-asserted in the SPF. Additional disable signals must be manually de-asserted by defining constant signals on them. For example,

```
dc_shell> set_dft_signal -view existing_dft -type Constant \
    -port SPC_CORE_DISABLE* -active_state 0
```

Configuring Shift Power Groups for a Top-Level Codec

If you are implementing a top-level codec, you must configure shift power groups for that codec using the pertinent options of the `set_streaming_compression_configuration` command. For more information, see [Configuring Shift Power Groups on page 951](#).

Configuring Shift Power Groups in TestMAX ATPG

Use the following commands in TestMAX ATPG to configure ATPG use of the shift power groups hardware:

```
DRC_T> set_drc -spc_chain SPC_chain_name
DRC_T> set_atpg -shift_controller_peak probability_value
```

SPC_chain_name is the name of the scan path that contains the SPC chain.

probability_value is the maximum percentage of scan cells that can switch in a shift cycle. TestMAX ATPG rejects patterns that exceed this switching percentage.

The STIL protocol file (SPF) created by the TestMAX DFT tool enables shift power groups by default. When enabled, you must configure the feature with the preceding commands, otherwise the compressed scan chains will fail DRC due to chain blockages.

Alternatively, you can assert the shift power disable signal, in which case the DFTMAX Ultra codec degenerates to a non-shift-power codec and no shift power configuration commands are needed.

Using Shift Power Groups With Other DFT Features

The shift power groups feature interacts with other DFT features as follows:

- Multiple test modes

You can use shift power groups with multiple test modes, including multiple DFTMAX Ultra compression modes. Configure the SPC chain in each DFTMAX Ultra compression mode. See [Per-Test-Mode Streaming Configuration Options on page 943](#) for supported options.

The control chain must be external only in DFTMAX Ultra compression modes. You can use the `-test_mode` option of the `set_scan_path` specification to limit the external chain specification to those modes (instead of `all`); the control chains are incorporated into regular scan chains in other modes.

If shift power groups are used, they must be used in all DFTMAX Ultra test modes. You cannot mix codecs with and without shift power groups across test modes.

- DFT partitions

You can use shift power groups with DFT partitions. Configure the SPC chain in each partition that contains a DFTMAX Ultra codec. See [Per-Partition Streaming Configuration Commands on page 938](#) for supported options.

Although SPC chains can be created for multiple partitions, they are all stitched into the single external control chain specified by the `set_scan_path` command.

If shift power groups are used, they must be used in all partitions that contain a DFTMAX Ultra codec. You cannot mix codecs with and without shift power groups across partitions.

- Retiming registers

When you enable beginning and/or ending retiming registers, SPC chains are clocked on the leading clock edge instead of the trailing clock edge. This facilitates control chain concatenation at the top level. See [Retiming Scan-Ins and Scan-Outs to the Leading Clock Edge on page 211](#).

Limitations of Shift Power Groups

Shift power groups have the following limitations:

- This feature applies only to the compressed scan chains it is configured for. Standard scan modes are unaffected.
- When shift power groups are used, they must be used
 - In all DFTMAX Ultra test modes
 - In all codecs in the design (across both cores and DFT partitions)

You cannot mix DFTMAX Ultra codecs with and without shift power groups within the same design.

- The shift power control (SPC) chain must be an external (uncompressed) chain that you explicitly define using the `set_scan_path` command.
- When integrating cores that contain shift power groups, you must manually connect the core-level shift power disable signal to a top-level shift power disable signal.
- The `report_scan_path` command does not report SPC chain information.

In TestMAX Diagnosis, the following requirements apply:

- Diagnosis capability is limited. High-resolution diagnostics are not supported when shift power groups are used. Assert the shift power disable signal to generate patterns for high-resolution diagnostics.

In TestMAX ATPG, the following tasks are not supported when using shift power groups:

- Analyzing X effects or X sources performed during a TestMAX ATPG simulation
- Comparing simulation results from a VCD simulation file, the internal patterns from the fast-sequential simulator, or the internal patterns from the full-sequential simulator
- Reporting total (cumulative) power data with the `report_power` command after multiple incremental ATPG runs
- Saving patterns and fault lists to files at a specified checkpoint interval during ATPG pattern generation
- Saving a GZIP-compressed parallel pattern set that can be simulated during the ATPG process
- Assigning ATPG constraints during an IDDQ measure strobe when the IDDQ fault model is selected

Planning, Previewing, and Inserting DFTMAX Ultra Compression

The following topics describe how to plan, preview, and insert DFTMAX Ultra compression in your design:

- [Planning the Streaming DFT Architecture](#)
 - [Previewing and Inserting DFT Logic](#)
 - [Writing Out Test Protocols for TestMAX ATPG](#)
-

Planning the Streaming DFT Architecture

As you configure the streaming DFT architecture, you can use the streaming DFT planner to visualize the currently configured architecture:

```
streaming_dft_planner [-show flow | elements | all]
```

The flow report (the default) focuses on the overall DFT architecture structure, scan chain lengths, and compression ratios. The elements report focuses on the elements within the scan chains, such as clock and polarity information, lock-up latches, retiming registers, and test clock waveform information.

The output is ASCII so you can capture it in log files. You can modify the DFT configuration and rerun the `streaming_dft_planner` command as many times as needed until you are satisfied with the architecture.

The following topics provide more information about the streaming DFT planner:

- [DFT Planner Flow Report](#)
- [DFT Planner Elements Report](#)
- [DFT Planner Limitations](#)

See Also

- [SolvNet article 2150838, “Understanding the Streaming DFT Planner Report](#) for more information on the conventions and information fields shown in the DFT planner report

DFT Planner Flow Report

The DFT planner flow report focuses on scan chain lengths and compression ratios. This report summarizes the DFT architecture. It is useful for length-balancing multiple codecs.

The flow report shows many DFT architecture details, including

- Scan chain counts and lengths, including external chains
- Codec clock, scan-enable, scan-in, and scan-out signal information
- Codec input and output shift register lengths
- Pipeline stages (core-level and top-level)
- Clock chains (core-level and top-level)
- DFT partitions
- The contents of streaming compression cores, including external chains
- The total shift latency of the design architecture
- The streaming compression overhead for each codec
- The streaming compression overhead of the codec that determines the maximum total shift latency
- The overall input and output target compression of the design architecture

To generate a flow report, use the following command:

```
dc_shell> streaming_dft_planner -show {flow}
```

The flow report is the default report type, so you can omit the `-show` option.

[Example 158](#) shows the planner flow report for a streaming compression core and a top-level codec.

Example 158 DFT Planner Flow Report Example

```
Information: Detected compressed scan core(s): core (TEST-1463)
Information: Detected scanned or scannable logic. (TEST-1462)
Information: Inferring the mixed scan insertion and core integration (MII) flow
(TEST-1438)
Information: Using test design rule information from previous dft_drc run.
Architecting Scan Compression structures
Integrating Streaming Decompressor core/U_deserializer_ScanCompression_mode
Number of inputs = 2
Maximum size per input = 8
Decompressor Clock = CLK
Integrating Streaming Compressor core/U_serializer_ScanCompression_mode
Number of outputs = 2
Maximum size per output = 4
Compressor Clock = CLK
Architecting Scan Chains
Architecting Pipeline Structures
Information: For incremental pipeline balancing, 0 head stages will be inserted at the
top, along with 1 existing head stages in core core. (TEST-1433)
Information: For incremental pipeline balancing, 0 tail stages will be inserted at the
top, along with 1 existing tail stages in core core. (TEST-1434)
Number of Head Pipeline Stages = 1
```

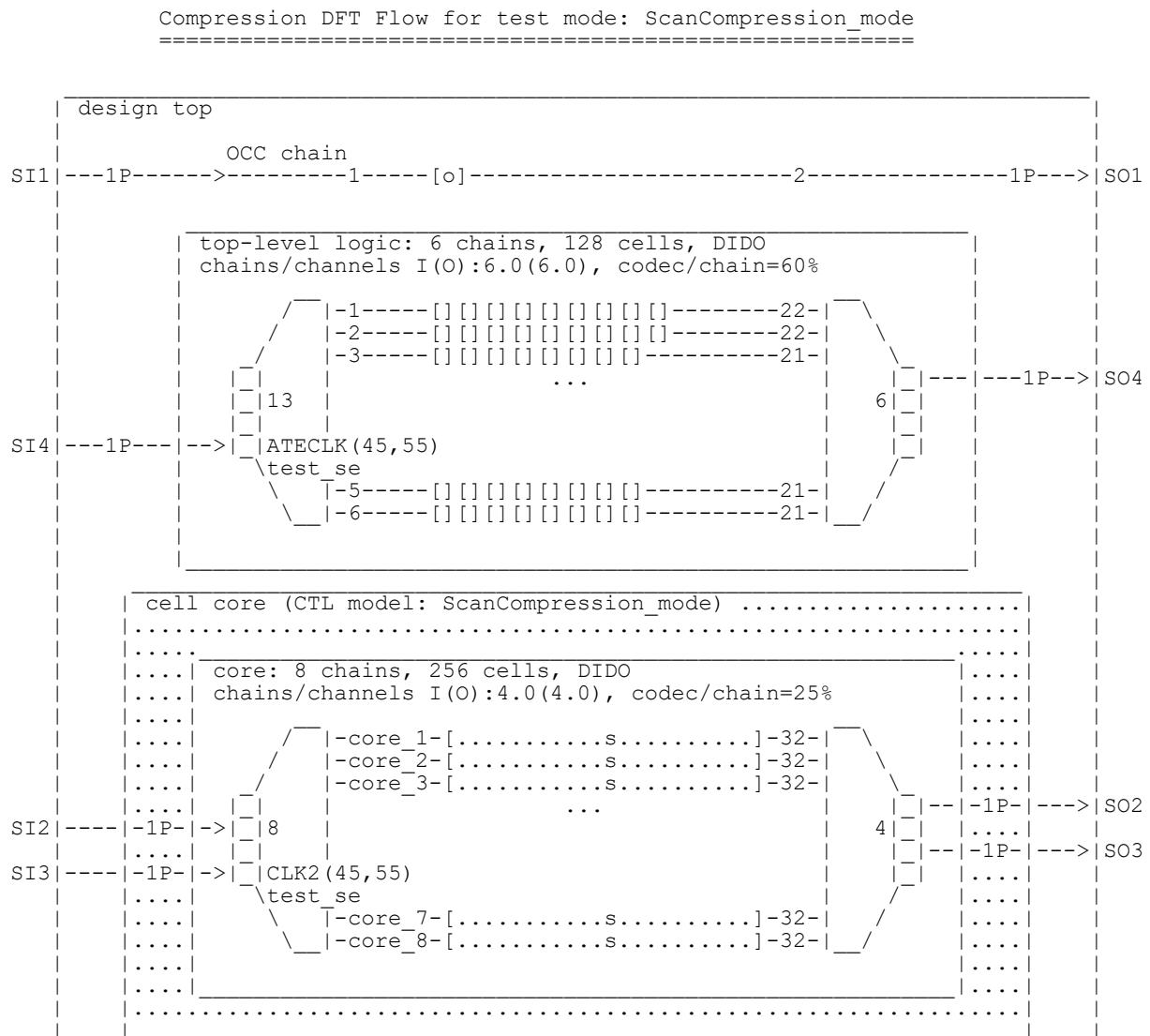
Chapter 24: Using DFTMAX Ultra Compression Planning, Previewing, and Inserting DFTMAX Ultra Compression

```

Number of Tail Pipeline Stages = 1
Architecting scan compression mode ScanCompression_mode with base mode Internal_scan
Architecting Streaming Decompressor
    Number of inputs = 1
    Maximum size per input = 6
    Decompressor Clock = ATECLK
Architecting Streaming Compressor
    Number of outputs = 1
    Maximum size per output = 6
    Compressor Clock = ATECLK
Architecting Load Decompressor (version 5.8)
    Number of inputs/chains/internal modes = 13/6/2
Architecting Unload compressor (version 5.8)
    Number of outputs/chains = 6/6
    Information: Compressor will have 100% x-tolerance

```

Running Streaming Compression DFT Planner (version 1.0)



```

| |
=====

Streaming Compression DFT Flow Information

DFT Flow: MII
Base scan mode: Internal_scan
Base scan mode chains: 4
Base scan mode maximum shift length: 129

Compression mode maximum shift length: 41
Codec with maximum shift length: core
Codec shift penalty (codec/chain): 25%
Target input compression: 3.15X (w.r.t. Internal_scan)
Target output compression: 3.15X (w.r.t. Internal_scan)

=====

```

Note the following tips for effective use of the DFT planner flow report:

- The scan chains in the diagrams are sized relative to their length. This allows you to adjust the architecture to balance the shift latency across codecs, cores, and external chains to maximize ATPG efficiency.
- Use the report to adjust the architecture to not exceed a codec shift penalty of 30%.

DFT Planner Elements Report

The DFT planner elements report uses the same architectural structure as the flow report for showing cores, top-level logic, and external scan chains. However, it also provides more information about elements within the scan chains, such as

- Clock timing and waveforms
- OCC controllers
- Scan element clocks and edges
- Pipeline register clocks and edges
- Clock and edge mixing transitions within scan chains
- Lock-up latches

To generate an elements report, use the following command:

```
dc_shell> streaming_dft_planner -show {elements}
```

[Example 159](#) shows the planner elements report for a streaming compression core and a top-level codec.

Example 159 DFT Planner Elements Report Example

```

Information: Detected compressed scan core(s): core (TEST-1463)
Information: Detected scanned or scannable logic. (TEST-1462)
Information: Inferring the mixed scan insertion and core integration (MII) flow
w (TEST-1438)
    Information: Using test design rule information from previous dft_drc run.
Architecting Scan Compression structures
Integrating Streaming Decompressor core/U_deserializer_ScanCompression_mode
    Number of inputs = 2
    Maximum size per input = 8
    Decompressor Clock = CLK
Integrating Streaming Compressor core/U_serializer_ScanCompression_mode
    Number of outputs = 2
    Maximum size per output = 4
    Compressor Clock = CLK
    Architecting Scan Chains
Architecting Pipeline Structures
Information: For incremental pipeline balancing, 0 head stages will be inserted at the
top, along with 1 existing head stages in core core. (TEST-1433)
Information: For incremental pipeline balancing, 0 tail stages will be inserted at the
top, along with 1 existing tail stages in core core. (TEST-1434)
    Number of Head Pipeline Stages = 1
    Number of Tail Pipeline Stages = 1
Architecting scan compression mode ScanCompression_mode with base mode Internal_scan
Architecting Streaming Decompressor
    Number of inputs = 1
    Maximum size per input = 6
    Decompressor Clock = ATECLK
Architecting Streaming Compressor
    Number of outputs = 1
    Maximum size per output = 6
    Compressor Clock = ATECLK
Architecting Load Decompressor (version 5.8)
    Number of inputs/chains/internal modes = 13/6/2
Architecting Unload compressor (version 5.8)
    Number of outputs/chains = 6/6
Information: Compressor will have 100% x-tolerance

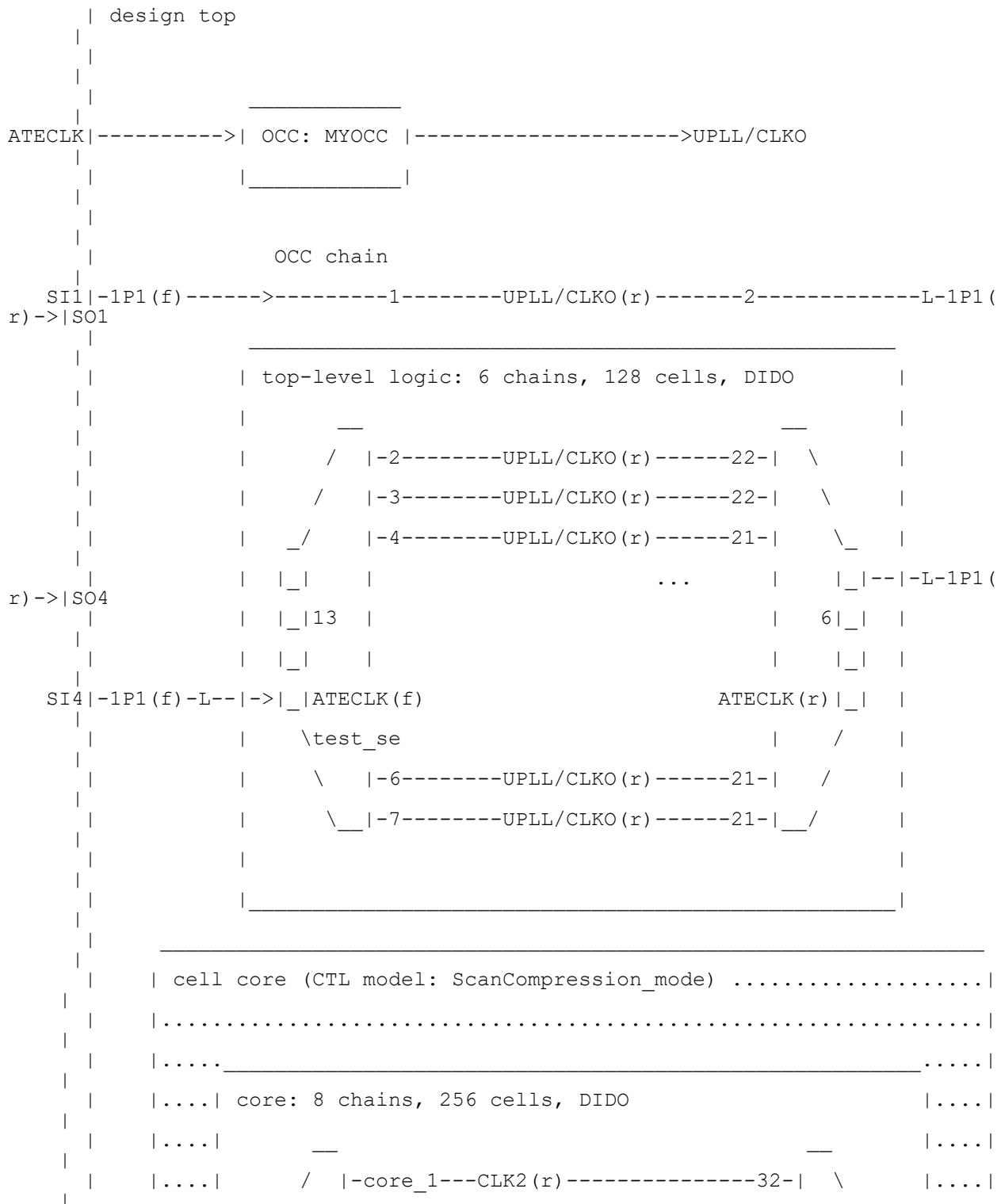
        Running Streaming Compression DFT Planner (version 1.0)
=====
=====

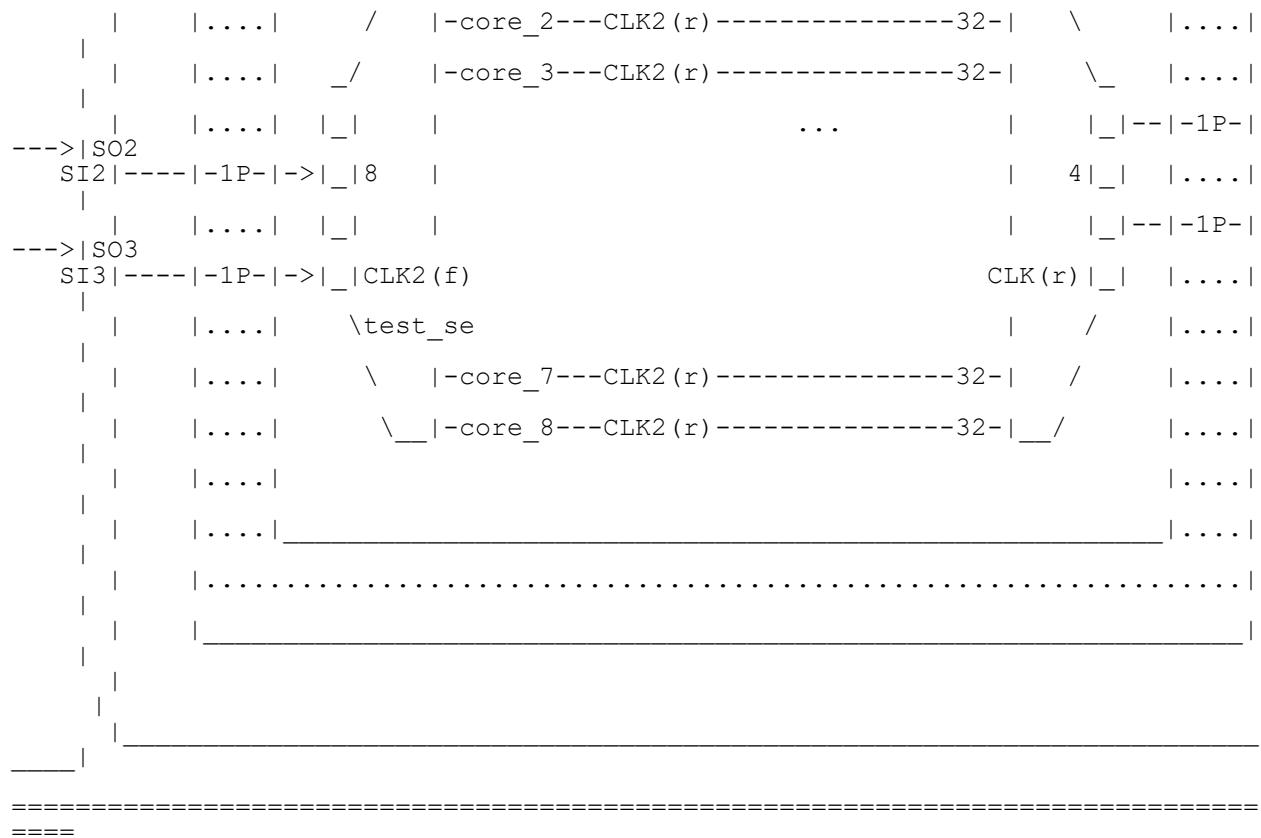
        Compression DFT Elements for test mode: ScanCompression_mode
=====
=====

Rising edge      : r          Falling edge      : f          Lockup latch      : L
Clocks          : ATECLK, CLK2
Pipeline clocks : CLK2(P1)
Timing          :

ATECLK(45,55)   : _____ |^-| _____
CLK2(45,55)     : _____ |^-| _____

```





Streaming Compression DFT Elements Information

DFT Flow:	MII
Base scan mode:	Internal_scan
Clock with maximum number of flops:	CLK2 (r, 256 FFs)
Clock with minimum number of flops:	UPLL/CLKO (r, 128 FFs)
Total number of lockup latches:	3

Note the following tip for effective use of the DFT planner elements report:

- Scan elements clocked by OCC-controlled clocks show the fast clock names for identification, but the scan architecture (such as ordering and lock-up latches) is determined by the ATE clock.

DFT Planner Limitations

Note the following limitations of the DFT planner:

- When using OCC controllers,
 - The top-level connections between core-level and top-level clock chain segments are not shown.
 - In the elements report, compressed scan chains (core-level or top-level) clocked by an OCC controller inside a core show the top-level ATE clock name, not the OCC-controlled clock name.
- When using streaming compressed scan cores,
 - The codec clock name shown might be the core-level clock name instead of the name of the top-level clock that reaches it.
 - In the elements report, pipeline stages inside cores only show depth information, not clock information.
 - In the elements report, clock edge information for scan chains inside cores might be incorrect.
 - In the elements report, scan chains inside cores do not show full element details.
 - In mixed insertion and integration (MII) flows, when external chains with pipeline registers inside cores are included in a top-level external chain, the tail pipeline depth might not be shown properly.
 - All CTL-modeled cores must be created with the K-2015.06 release or later. Cores created with earlier releases are not supported.
- In the elements report,
 - The correct maximum chain length might not be shown.
 - Warnings about invalid lock-up latch structures might be incorrect.
- Nested integration flows are not supported.
- Core wrapping is not supported.
- The internal pins flow is not supported.
- Other compression technologies, such as DFTMAX compression, are not supported.

Previewing and Inserting DFT Logic

After you complete your DFT configuration, you can use the `preview_dft` command to review the scan architecture and test-mode signal details before scan insertion is performed, as shown in [Example 160](#).

Example 160 Output From the preview_dft Command for a Compressed Scan Configuration

```
*****
Current mode: Internal_scan
*****  
  
Number of chains: 3
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks  
  
Scan chain '1' (test_si1 --> test_so1) contains 22 cells
  Active in modes: Internal_scan  
  
Scan chain '2' (test_si2 --> test_so2) contains 21 cells
  Active in modes: Internal_scan  
  
Scan chain '3' (test_si3 --> test_so3) contains 21 cells
  Active in modes: Internal_scan  
  
*****
Current mode: ScanCompression_mode
*****  
  
Number of chains: 8
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix  
  
Scan chain '1' contains 8 cells
  Active in modes: ScanCompression_mode  
  
(...omitted...)  
  
Scan chain '8' contains 8 cells
  Active in modes: ScanCompression_mode  
  
=====  
Test Mode Controller Information
=====  
  
Test Mode Controller Ports
-----  
test_mode: test_mode  
  
Test Mode Controller Index (MSB --> LSB)
```

```
-----
test_mode
Control signal value - Test Mode
-----
0 Internal_scan - InternalTest
1 ScanCompression_mode - InternalTest
```

During scan insertion, the `insert_dft` command creates and instantiates two scan compression designs: one for the compressor and one for the decompressor. By default, the `insert_dft` command instantiates these blocks at the top level of the current design. To insert the codec logic into a lower-level hierarchical block, use the `set_dft_location -include {CODEC}` command. For more information, see the man page.

After DFT insertion, you can use the `report_scan_path` command to review the scan chain structures that now exist in the standard scan and compressed scan modes:

```
dc_shell> report_scan_path -view existing_dft -test_mode all
```

See Also

- [Chapter 14, Previewing, Inserting, and Checking DFT Logic](#) for more information about previewing and inserting DFT logic

Writing Out Test Protocols for TestMAX ATPG

You can write out test protocols for both test modes using the `write_test_protocol -test_mode` command:

```
dc_shell> write_test_protocol -output scan.spf \
           -test_mode Internal_scan
dc_shell> write_test_protocol -output scancompress.spf \
           -test_mode ScanCompression_mode
```

TestMAX ATPG uses these protocol files, along with the design netlist, for pattern generation.

For more information about running TestMAX ATPG on DFTMAX Ultra designs, see “Using TestMAX ATPG and DFTMAX Ultra Compression” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Library Cell Requirements for Codec Implementation

The DFTMAX Ultra codec architecture requires that the target libraries have the following cell types available for mapping when using the `insert_dft` command:

- Level-sensitive latch
- Flip-flop with asynchronous reset

If applied, the `dont_use` attribute prevents these cell types from being available for mapping. This attribute might be present in the original library. More commonly, it is applied by `set_dont_use` commands in the synthesis script to prevent particular cell types from being used in the design.

Use the `report_lib` command to check for library cells that have the `dont_use` attribute applied, indicated by the “`u`” attribute annotation. For example,

```
dc_shell> report_lib lsi_10k
...
      Attributes:
Cell      Attributes
-----
...
LD1      s, u
LD2      s, u
LD3      s, u
LD4      s, u
...
...
```

Use the `remove_attribute` command to remove the `dont_use` attribute from any required library cells before DFT insertion. For example,

```
dc_shell> remove_attribute lsi_10k/LD* dont_use
lsi_10k/LD1 lsi_10k/LD2 lsi_10k/LD3 lsi_10k/LD4
dc_shell> insert_dft
```

If needed, use the `set_dont_use` command to reapply the attributes after the `insert_dft` command completes.

25

Hierarchical DFTMAX Ultra Compression

DFTMAX Ultra compression supports hierarchical scan synthesis. You can perform scan synthesis independently for each lower-level DFT-inserted core. When you use instances of these cores at a higher level of hierarchy, the tool automatically incorporates their scan structures at the higher level.

Hierarchical DFT using DFTMAX Ultra compression is described in the following topics:

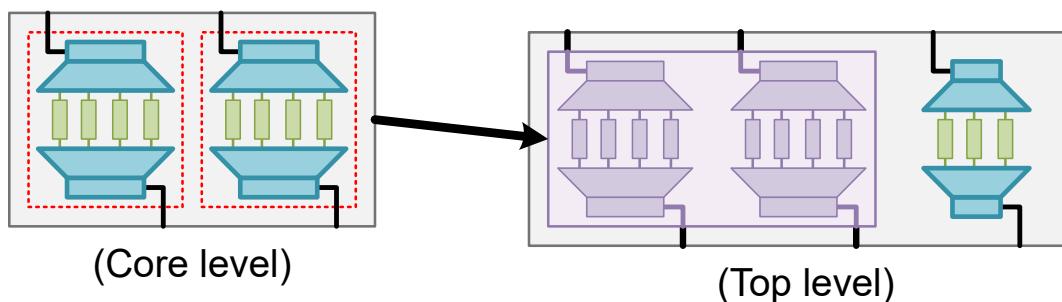
- [Overview of Hierarchical DFTMAX Ultra Compression](#)
- [Creating Cores for Integration](#)
- [Performing Core Integration](#)
- [Using DFT Partitions During Core Integration](#)
- [Using Multiple Test Modes in Hierarchical Flows](#)

Overview of Hierarchical DFTMAX Ultra Compression

DFTMAX Ultra compression supports hierarchical scan synthesis. You can perform scan synthesis independently for each lower-level DFT-inserted block, called a *core*. When you use instances of these cores at a higher level of hierarchy, the tool automatically incorporates their scan structures at the higher level. This is called *core integration*.

[Figure 442](#) shows a core that is created with two DFT partitions, then integrated at a higher level of hierarchy.

Figure 442 Integrating a Core in a Hierarchical Flow



Note:

The resulting integrated design can itself become a core for use at even higher levels of hierarchy. However, for simplicity, this section refers to the integration level as the “top level.”

Creating Cores for Integration

To create a standard scan or compressed scan core, insert scan at the core level. No special options are needed for core creation. [Example 161](#) shows a simple top-down insertion compressed scan core creation script.

Example 161 Script for Creation of DFTMAX Ultra Compression Core

```
# enable scan compression
set_dft_configuration -streaming_compression enable

# specify standard scan chain count
set_scan_configuration -chain_count 1

# enable core-level DFTMAX Ultra compression insertion
# and specify compressed chain count
set_streaming_compression_configuration -chain_count 4

# configure required scan clock signals
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}

# configure core-level DFT signal ports (if they exist)
set_dft_signal -view spec -type ScanDataIn -port SI
set_dft_signal -view spec -type ScanDataOut -port SO
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type TestMode -port TM

# insert DFT
create_test_protocol
dft_drc
preview_dft
insert_dft

# write out design in multiple formats
write -format ddc -hierarchy -output core1.ddc
write_test_model -format ddc -output core1.ctlddc
change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output core1.v
```

Each core must have CTL test model information so that the tool can perform top-level integration. If the block fits in memory during top-level integration, you can use the `write` command to write a design .ddc file that contains the full design netlist as well as the CTL test model information:

```
dc_shell> write -format ddc -hierarchy -output design_name.ddc
```

If the block is large, you can use the `write_test_model` command to write out a test-model-only .ddc file that contains the CTL test model along with an interface-only representation of the core that allows the test model to be linked at the top level:

```
dc_shell> write_test_model -format ddc -output design_name.ctlddc
```

You can use either format for standard scan and compressed scan cores in core integration flows.

Performing Core Integration

In the DFTMAX Ultra flow, you can perform DFT insertion with any combination of the following logic types present anywhere in the logical hierarchy of the design:

- DFTMAX Ultra compressed scan cores
- Standard scan cores
- Scanned or scannable logic

If only scanned or scannable logic exists, the tool performs top-down scan insertion, as described in [Chapter 24, Using DFTMAX Ultra Compression](#)." However, if one or more cores are present, the tool performs core integration, as described in the following topics:

- [Automatic Detection of Existing Logic Types](#)
- [Configuring Core Integration](#)
- [Core Integration Script Examples](#)

Automatic Detection of Existing Logic Types

You do not need to specify what types of logic exist at the top level; the tool automatically detects and prints information messages about what scanned or scannable logic, standard scan cores, or compressed scan cores exist.

When you run the `preview_dft` or `insert_dft` command, the tool detects the existing logic types and prints one or more of the following information messages:

Information: Detected standard scan core(s): *core_list*. (TEST-1463)

Information: Detected compressed scan core(s): *core_list*. (TEST-1463)

Information: Detected scanned or scannable logic. (TEST-1462)

After logic detection, the tool infers and reports the flow type as follows:

- If only noncore scanned or scannable logic is detected (TEST-1462), the tool prints the following message:

Information: Inferring the top-down scan insertion (TDI) flow.
(TEST-1436)

- If only cores are detected (TEST-1463), the tool prints the following information message:

Information: Inferring the bottom-up core integration (BUI) flow.
(TEST-1437)

- If noncore scanned or scannable logic and cores are both detected (TEST-1462 and TEST-1463), the tool prints the following message:

Information: Inferring the mixed scan insertion and core integration (MII) flow.
(TEST-1438)

For more information about logic types and flow names, see [Chapter 28, DFTMAX Ultra Flow Naming Conventions.](#)"

Configuring Core Integration

To perform core integration, simply enable DFTMAX Ultra compression:

```
dc_shell> set_dft_configuration -streaming_compression enable
```

Then, configure the standard scan and compressed scan modes to be created, as described in the following topics.

- [Configuring the Standard Scan Mode](#)
- [Configuring the Compressed Scan Mode](#)

Configuring the Standard Scan Mode

In standard scan mode, all core and noncore logic operates in uncompressed mode. DFTMAX Ultra compression uses the following core integration rules to create scan chains from the existing logic in standard scan mode:

- Any scanned or scannable logic is stitched into scan chains.
- Scan chains inside standard scan cores become scan segments that are incorporated into scan chains, as needed, for length-balancing purposes.
- Scan chains from the standard scan mode of compressed scan cores become scan segments that are incorporated into scan chains, as needed, for length-balancing purposes.

Use the `-chain_count` or `-max_length` option of the `set_scan_configuration` command to specify the target scan chain configuration for the standard scan mode. For example,

```
dc_shell> set_scan_configuration -chain_count 3
```

Configuring the Compressed Scan Mode

By default, the tool reuses the base mode scan I/Os for the compressed mode. The number of base mode I/Os is determined by the base mode chain count, which is typically set with the `set_scan_configuration -chain_count` command.

Note:

For simplicity, this section assumes that the `set_scan_configuration -chain_count` option is used to specify the base mode chain count.

However, the base mode chain count can also be specified indirectly with the `-max_length` option. In addition, either specification can be altered due to requirements such as clock mixing.

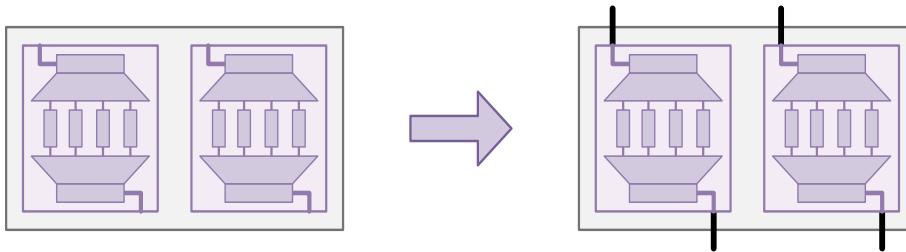
DFTMAX Ultra compression uses the following core integration rules to create scan structures from the existing logic in compressed scan mode:

- Scan connections of compressed scan cores are promoted to top-level scan connections.
- Scan chains inside standard scan cores become scan segments that are incorporated into top-level scan structures as described in the following rule.
- For uncompressed logic (scanned or scannable logic or standard scan core segments),
 - If no such uncompressed logic exists, then only compressed scan cores exist. See [Integrating Compressed Scan Cores With No Uncompressed Logic on page 972](#).
 - If a top-level codec is configured, the tool inserts a top-level codec to compress the uncompressed logic. See [Compressing Uncompressed Logic on page 973](#).
 - If no top-level codec is configured, the tool creates standard scan chains from the uncompressed logic. See [Building Standard Scan Chains From Uncompressed Logic on page 974](#).

Integrating Compressed Scan Cores With No Uncompressed Logic

If only compressed scan cores exist, all compressed scan core connections are promoted to top-level connections, regardless of the I/O configuration of the base mode. See [Figure 443](#).

Figure 443 Integrating Only Compressed Scan Cores



In this case, you should set your base mode chain count to match the I/Os used by the cores in compression mode so that the I/O resources are equally and fully used in both modes.

Compressing Uncompressed Logic

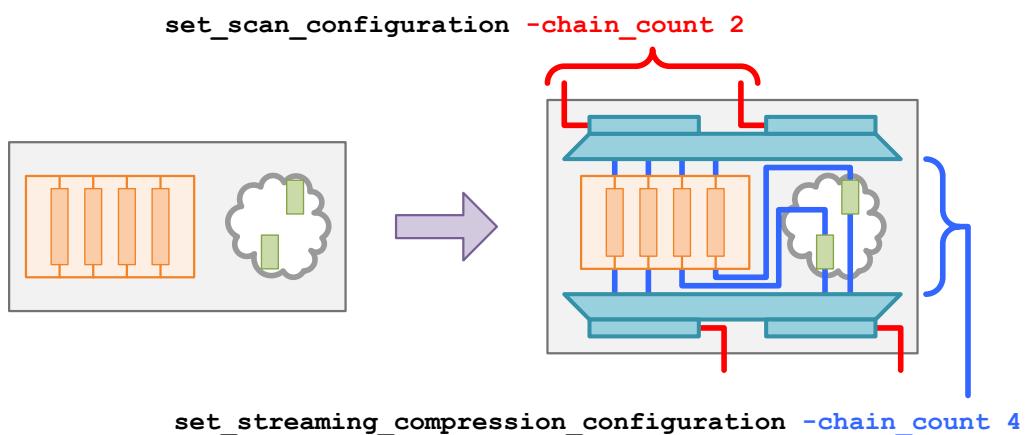
If you have uncompressed logic in your design, you can compress the logic by configuring a top-level codec with the `-chain_count` or `-max_length` option of the `set_streaming_compression_configuration` command. For example,

```
dc_shell> set_streaming_compression_configuration -chain_count 4
```

The term “top-level” differentiates this codec from codecs in compressed scan cores.

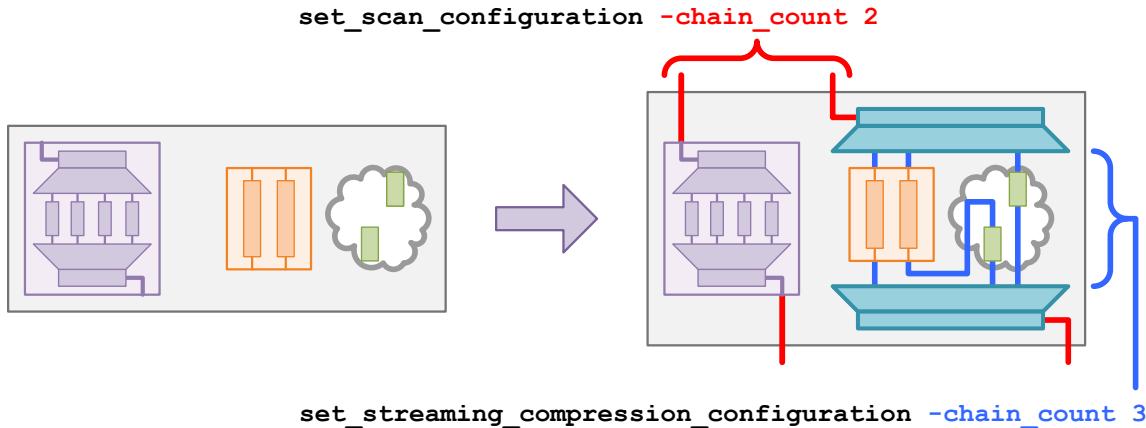
By default, the codec uses the scan I/Os inherited from the base mode. [Figure 444](#) shows an example with only uncompressed logic.

Figure 444 Compressing Uncompressed Logic, Including a Standard Scan Core



If compressed scan cores exist, their promoted connections reduce the number of scan I/Os available to the top-level codec. [Figure 445](#) shows an example with a compressed scan core along with uncompressed logic.

Figure 445 Integrating Compressed Scan Cores and Compressing Uncompressed Logic

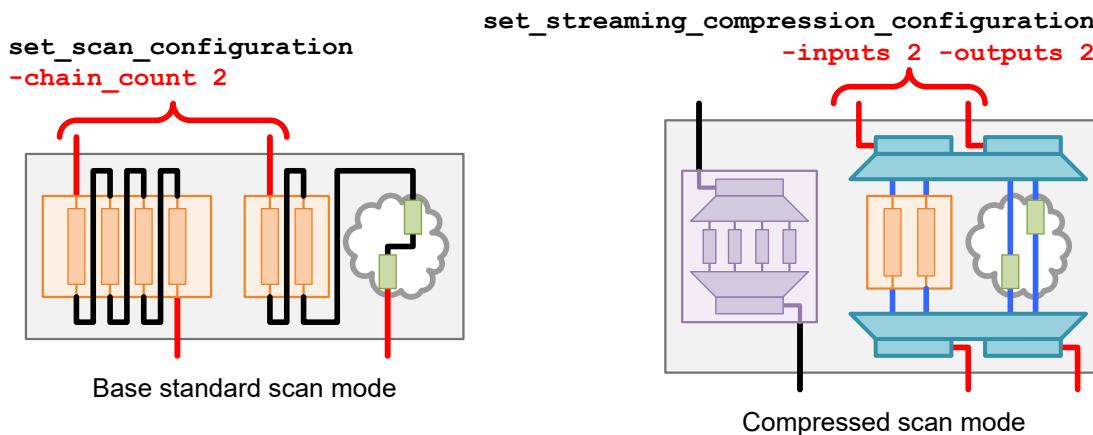


If you want to force a specific number of scan-in or scan-out connections to be used for the top-level codec, use the `-inputs` or `-outputs` option, respectively. For example,

```
dc_shell> set_scan_configuration -chain_count 2
dc_shell> set_streaming_compression_configuration \
           -chain_count 4 -inputs 2 -outputs 2
```

Figure 446 shows the previous design example with these commands applied. Note that the `-inputs` and `-outputs` options might cause the scan I/O requirements of the compressed scan mode to differ from the base standard scan mode.

Figure 446 Specifying The Number of Codec Scan I/Os for a Top-Level Codec

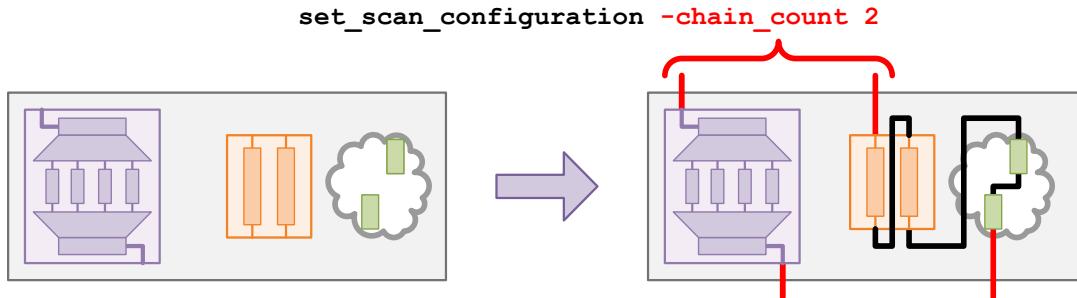


Building Standard Scan Chains From Uncompressed Logic

If you have uncompressed logic in your design and you do not define a codec with the `set_streaming_compression_configuration` command, the tool builds standard scan

chains from the uncompressed logic. The target chain count is determined by the available I/Os inherited from the base mode, less any I/Os needed for promoted compressed scan core connections. See [Figure 447](#).

Figure 447 Integrating Compressed Scan Cores and Scan-Stitching Uncompressed Logic



Core Integration Script Examples

This topic provides script examples of core integration.

Integrating Only Compressed Scan Cores

[Example 162](#) shows a script that performs BUI[C] integration of DFTMAX Ultra cores.

Example 162 Script for Bottom-Up Integration of Compressed Scan Cores

```
# enable DFTMAX Ultra compression
# BUI[C] flow is used automatically when only compressed scan cores exist
set_dft_configuration -streaming_compression enable

# configure top-level chain count
# (set to number of I/Os required by compressed scan cores in
# compressed scan mode)
set_scan_configuration -chain_count 2

# configure top-level DFT clocks
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}

# configure top-level DFT signal
set_dft_signal -view spec -type ScanDataIn -port {SI1 SI2}
set_dft_signal -view spec -type ScanDataOut -port {SO1 SO2}
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type TestMode -port TM

# insert DFT
create_test_protocol
dft_drc
preview_dft
insert_dft
```

Integrating Compressed Scan Cores With Uncompressed Logic

[Example 163](#) shows a script that performs MII[C]-C or MII[S][C]-C integration of DFTMAX Ultra cores and inserts a top-level codec to compress the uncompressed logic. The only difference from the previous example is the configuration of a top-level codec to compress the uncompressed logic.

Example 163 Script for Mixed Insertion and Integration of Compressed Scan Cores

```
# enable and configure DFTMAX Ultra compression;
# MII[C]-C flow is used when compressed scan cores exist along with
# additional scanned or scannable logic
set_dft_configuration -streaming_compression enable

# configure scan, including top-level codec
set_scan_configuration -chain_count 2 ;# includes compressed core I/Os
set_streaming_compression_configuration -chain_count 4

# configure top-level DFT clocks
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}

# configure top-level DFT signal
set_dft_signal -view spec -type ScanDataIn -port {SI1 SI2}
set_dft_signal -view spec -type ScanDataOut -port {SO1 SO2}
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type TestMode -port TM

# insert DFT
create_test_protocol
dft_drc
preview_dft
insert_dft
```

Using DFT Partitions During Core Integration

In the MII[C]-C flow, if the amount of top-level logic is large, you can optionally create multiple top-level DFT partitions, each having its own codec. This is known as the MII[C]-C-P flow. For more information about creating DFT partitions in the DFTMAX Ultra flow, see [Top-Down Insertion Compressed Scan Flow With Partitions on page 934](#).

Note:

In the MII[C]-C-P flow, all compressed scan cores must remain in the default partition. You cannot reassign them to user-defined partitions.

[Example 164](#) shows a script that performs MII[C]-C-P integration of DFTMAX Ultra cores, along with the insertion of two codecs to compress top-level scan logic.

Example 164 Script for Mixed Insertion and Integration of DFTMAX Ultra Compression Cores With Partitions

```
# enable and configure DFTMAX Ultra compression MII[C]-C flow
set_dft_configuration -streaming_compression enable

# configure scan in each partition
define_dft_partition my_part -include {...}
current_dft_partition my_part
set_scan_configuration -chain_count 1
set_streaming_compression_configuration -chain_count 4
set_dft_signal -port SI1 -type ScanDataIn
set_dft_signal -port SO1 -type ScanDataOut

current_dft_partition default_partition
set_scan_configuration -chain_count 2 ;# includes core scan I/Os
set_streaming_compression_configuration -chain_count 4
set_dft_signal -port SI2 -type ScanDataIn
set_dft_signal -port SO2 -type ScanDataOut

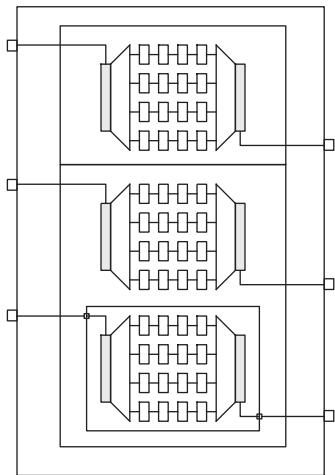
# configure top-level DFT clocks
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}

# configure top-level DFT signal
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type TestMode -port TM

# insert DFT
create_test_protocol
dft_drc
preview_dft
insert_dft
```

For a design with a single compressed scan core, the script in [Example 164](#) results in the top-level scan logic shown in [Figure 448](#).

Figure 448 The Mixed Insertion and Integration With Partitions (MII[C]-C-P) Flow



Using Multiple Test Modes in Hierarchical Flows

When you invoke DFTMAX Ultra compression, by default the tool creates two test operating modes: a standard (uncompressed) scan mode and a compressed scan mode. To create a larger set of test modes, use the `define_test_mode` command. The tool creates logic to support each defined mode. For details, see [DFTMAX Ultra Compression and Multiple Test Modes on page 940](#).

In hierarchical flows that perform core integration, each lower-level core can have multiple test modes defined. By default, the tool groups together identically named modes in different cores and at the top level and selects those core-level modes as a single mode at the top level. If the modes have different names, they are grouped by usage and then combined into all name combinations.

You can also explicitly specify which combinations of lower-level test modes are to be targeted by top-level test modes by using the `-target` option of the `define_test_mode` command. This is known as *test mode scheduling*.

The default test mode assignment and test mode scheduling behaviors for integrating DFTMAX Ultra cores are the same as for integrating DFTMAX cores. For more information, see [Using Multiple Test Modes in Hierarchical Flows on page 725](#).

Note:

The `-target` option has a limitation when used in the MII compressed scan core integration mode. See [DFT Synthesis Limitations on page 981](#).

Mixing DFTMAX and DFTMAX Ultra Compression Core Modes

You can integrate cores that have both DFTMAX and DFTMAX Ultra test modes. This includes any mix of

- Cores that have only DFTMAX modes
- Cores that have only DFTMAX Ultra modes
- Cores that have both DFTMAX and DFTMAX Ultra modes

The requirements described in [Mixing DFTMAX and DFTMAX Ultra Compression Modes on page 942](#) also apply to core integration modes; see that section for more information.

By default, the relationship between core-level and top-level test modes is determined by test mode name. If your core-level DFTMAX modes do not share any names with your core-level DFTMAX Ultra modes, you can use this default name-based core test mode assignment.

Alternatively, you can override the default name-based association of core-level test modes. This is known as *test mode scheduling*. To do this, use the `-target` option of the `define_test_mode` command.

[Example 165](#) shows the integration of two wrapped DFTMAX Ultra cores (CORE1 and CORE2) and an unwrapped DFTMAX legacy IP core (COREIP). Note that HASS integration must be enabled for DFTMAX compression, while the integration mode is automatically detected for DFTMAX Ultra.

Example 165 Core Integration Example With DFTMAX and DFTMAX Ultra Compression Cores

```
# enable the DFTMAX and DFTMAX Ultra compression clients
set_dft_configuration \
    -scan_compression enable \
    -streaming_compression enable

# configure DFTMAX core integration mode
# (DFTMAX Ultra configures itself automatically)
set_scan_compression_configuration -hybrid true

# apply global DFT configuration
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}
set_dft_signal -view spec -type ScanEnable -port SE
set_scan_configuration -clock_mixing mix_clocks

# define the test modes, including core test-mode scheduling
define_test_mode CORES_SCAN -usage scan \
    -target {CORE1:wrp_if CORE2:wrp_if}
define_test_mode CORES_DFTMAX_ULTRA -usage streaming_compression \
    -target {CORE1:ScanCompression_mode CORE2:ScanCompression_mode}
define_test_mode TOP_SCAN -usage scan \
```

```
-target {COREIP:Internal_scan top}
define_test_mode TOP_DFTMAX -usage scan_compression \
    -target {COREIP:ScanCompression_mode top}

# configure each test mode
# (modes with inward-facing cores use their default configuration)
set_scan_configuration -test_mode TOP_SCAN -chain_count 8
set_scan_compression_configuration \
    -test_mode TOP_DFTMAX -base_mode TOP_SCAN \
    -chain_count 10
```

See Also

- [Using Multiple Test Modes in Hierarchical Flows on page 725](#) for more information about default test-mode assignment and test mode scheduling

26

DFTMAX Ultra Limitations and Known Issues

This chapter contains the limitations and known issues that apply to DFTMAX Ultra compression.

This chapter contains the following topics:

- [DFT Synthesis Limitations](#)
 - [Supported DFT Insertion Flows](#)
-

DFT Synthesis Limitations

The following DFT synthesis requirements and limitations apply to DFTMAX Ultra compression:

- The minimum compression ratio for DFTMAX Ultra compression is 3. For example, with five scan inputs and five scan outputs, a minimum of 15 internal chains must be used.
- The target libraries must have the following cell types available for codec implementation:
 - Level-sensitive latch
 - Flip-flop with asynchronous reset

If applied, the `dont_use` attribute prevents these cell types from being used for mapping. Use the `remove_attribute` command to remove the `dont_use` attribute from any required library cells before running the `insert_dft` command. See [Library Cell Requirements for Codec Implementation on page 967](#).

- In core integration (BUI and MII) flows, post-DFT DRC of test modes that contain active compressed scan cores is not supported.

- When DFT partitions are used,
 - All partitions must use streaming DFT compression. Any partitions that contain uncompressed logic (top-level logic or standard scan cores) must have a codec configured using the `set_streaming_compression_configuration` command.
You can also keep scan logic or standard scan cores uncompressed by defining external chains.
 - All compressed scan cores must remain in the default partition. You cannot reassign them to user-defined partitions.
- When you use the `-target` option of the `define_test_mode` command in the MII core integration flow, a top-level codec is inserted in a test mode only when you target the top-level logic by including the name of the current design in the target list. You cannot insert a codec for targeted cores without also compressing the top-level logic, which includes any untargeted standard scan cores and any wrapped cores in outward-facing mode.
- When OCC controllers are used with compressed (decompressor-driven) clock chains,
 - At least two codec inputs are required. There is no requirement for the codec outputs. For more information, see [Using OCC Controllers With DFTMAX Ultra Compression on page 944](#).
 - The `-chain_count` option of the `set_dft_clock_controller` command cannot be set to a value other than its default of 1.
- You can mix DFTMAX and DFTMAX Ultra compression modes in the same design. However, both compression types cannot be active in the same test mode. Compression types other than DFTMAX and DFTMAX Ultra are not supported.
- When integrating pipelined cores, the top-level depths reported by the TEST-1433 and TEST-1434 information messages are incorrect, and DFT insertion does not abort if the total pipeline depth is smaller than the largest core-level depth.
- When using wrapped cores, top-down flat testing with transparent wrapped cores is not supported.
- Synthesis of static-X chains is not supported.
- Scan groups, defined with the `set_scan_group` command, are not supported.
- Scan-through-TAP, which provides access to internal scan chains through the TDI and TDO ports of the IEEE Std 1149.1 test access ports (TAP), is not supported.
- End-of-cycle measures are not supported.
- For flows other than top-down insertion (TDI) flows, streaming codecs cannot use scan-enable signals defined with the `-usage` option of the `set_dft_signal` command.

You should define at least one scan-enable signal without a usage for proper codec insertion and operation.

- Modifications to the streaming compression IP blocks, such as adding inversions in codec scan paths, are not supported.

Supported DFT Insertion Flows

The following DFT insertion flows are supported by DFTMAX Ultra compression:

- TDI-S/C-DIDO
- TDI-S/C-P-DIDO
- BUI[C]-DIDO
- BUI[C-P]-DIDO
- MII[S/C]-S/C-DIDO
- MII[S/C]-S/C-P-DIDO
- MII[S/C-P]-S/C-DIDO
- MII[S/C-P]-S/C-P-DIDO

See [Chapter 28, DFTMAX Ultra Flow Naming Conventions](#), for information on the flow naming conventions used in this list.

DFTMAX Ultra STIL Protocol File Syntax

The `write_test_protocol` command writes out a STIL protocol file containing a complete description of the DFTMAX Ultra compression architecture. TestMAX ATPG uses this information to generate test patterns for the compressed scan design.

This chapter contains the following topics:

- [STIL Protocol File Contents](#)
 - [STIL Protocol File Example](#)
-

STIL Protocol File Contents

The `write_test_protocol` command writes out a STIL protocol file containing a description of the DFTMAX Ultra compression architecture. TestMAX ATPG uses the information from the file to determine what types of patterns to generate for the specific scan compression architecture. In general, you do not need to be concerned about the contents of this file. However, for debugging purposes, you might want to know the meanings of the statements in the file.

The `CompressorStructures` section specifies the architecture of the decompression and compression logic of the device. Within that section, a `Compressor` section represents a decompression or compression structure in the design and provides detailed information about the logic connections in that structure. TestMAX ATPG uses this information to configure the test logic for each pattern and generate pattern data to target specific faults.

STIL Protocol File Example

[Example 166](#) shows the sections in a STIL procedure that describe the DFTMAX Ultra compression architecture implemented for a particular device. The file was written by the `write_test_protocol` command for the design shown in [Figure 449](#). Note that the line numbers shown along the left of the example are not included in the file.

Example 166 STIL Protocol File and DFTMAX Ultra Compression Architecture

```

1 UserKeywords StreamingStructures CompressorStructures;
2 StreamingStructures {
3     ExternalCyclesPerShift 59;
```

Chapter 27: DFTMAX Ultra STIL Protocol File Syntax

STIL Protocol File Example

```

4      LoadSerializer "top_U_deserializer_ScanCompression_mode" {
5          Length 59;
6          ActiveScanChains load_group;
7      }
8      UnloadSerializer "top_U_serializer_ScanCompression_mode" {
9          Length 49;
10         ActiveScanChains unload_group;
11     }
12 }
13 CompressorStructures {
14     Compressor "top_U_decompressor_ScanCompression_mode" {
15         LoadSerializer "top_U_deserializer_ScanCompression_mode" 0 1 2 3 4
16         5 6 7
17         8 9;
18         ModeSerializer "top_U_deserializer_ScanCompression_mode" 56 57;
19         LoadSerializerDir "top_U_deserializer_ScanCompression_mode" 55;
20         CoreGroup core_group;
21         Modes 4;
22         Mode 0 {
23             UnloadModeSerializer "top_U_deserializer_ScanCompression_mode"
24             10 11
25             12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
26             33 34
27             35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53;
28             ModeSerializerControls {
29                 "top_U_deserializer_ScanCompression_mode"[56] = 0;
30                 "top_U_deserializer_ScanCompression_mode"[57] = 0;
31             }
32             Connection 0 "1" "11" "21" "31" "41";
33             Connection 1 "2" "12" "22" "32" "42";
34             Connection 2 "3" "13" "23" "33" "43";
35             Connection 3 "4" "14" "24" "34" "44";
36             Connection 4 "5" "15" "25" "35" "45";
37             Connection 5 "6" "16" "26" "36" "46";
38             Connection 6 "7" "17" "27" "37" "47";
39             Connection 7 "8" "18" "28" "38" "48";
40             Connection 8 "9" "19" "29" "39" "49";
41             Connection 9 "10" "20" "30" "40" "50";
42         }
43         Mode 1 {
44             UnloadModeSerializer "top_U_deserializer_ScanCompression_mode"
45             10 11
46             12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
47             33 34
48             35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53;
49             ModeSerializerControls {
50                 "top_U_deserializer_ScanCompression_mode"[56] = 0;
51                 "top_U_deserializer_ScanCompression_mode"[57] = 1;
52             }
53             Connection 0 "1" "18" "25" "32" "49";
54             Connection 1 "2" "19" "26" "33" "50";
55             Connection 2 "3" "20" "27" "34" "41";
56             Connection 3 "4" "11" "28" "35" "42";
57             Connection 4 "5" "12" "29" "36" "43";
58             Connection 5 "6" "13" "30" "37" "44";
59             Connection 6 "7" "14" "21" "38" "45";
60             Connection 7 "8" "15" "22" "39" "46";
61             Connection 8 "9" "16" "23" "40" "47";
62             Connection 9 "10" "17" "24" "31" "48";
63         }
64         Mode 2 {

```

Chapter 27: DFTMAX Ultra STIL Protocol File Syntax

STIL Protocol File Example

```

60          UnloadModeSerializer "top_U_deserializer_ScanCompression_mode"
10 11
61          12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
63          33 34
62          35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53;
63          ModeSerializerControls {
64              "top_U_deserializer_ScanCompression_mode"[56] = 1;
65              "top_U_deserializer_ScanCompression_mode"[57] = 0;
66          }
67          Connection 0 "1" "15" "29" "33" "47";
68          Connection 1 "2" "16" "30" "34" "48";
69          Connection 2 "3" "17" "21" "35" "49";
70          Connection 3 "4" "18" "22" "36" "50";
71          Connection 4 "5" "19" "23" "37" "41";
72          Connection 5 "6" "20" "24" "38" "42";
73          Connection 6 "7" "11" "25" "39" "43";
74          Connection 7 "8" "12" "26" "40" "44";
75          Connection 8 "9" "13" "27" "31" "45";
76          Connection 9 "10" "14" "28" "32" "46";
77      }
78      Mode 3 {
79          UnloadModeSerializer "top_U_deserializer_ScanCompression_mode"
10 11
80          12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
83          33 34
81          35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53;
82          ModeSerializerControls {
83              "top_U_deserializer_ScanCompression_mode"[56] = 1;
84              "top_U_deserializer_ScanCompression_mode"[57] = 1;
85          }
86          Connection 0 "1" "12" "23" "34" "45";
87          Connection 1 "2" "13" "24" "35" "46";
88          Connection 2 "3" "14" "25" "36" "47";
89          Connection 3 "4" "15" "26" "37" "48";
90          Connection 4 "5" "16" "27" "38" "49";
91          Connection 5 "6" "17" "28" "39" "50";
92          Connection 6 "7" "18" "29" "40" "41";
93          Connection 7 "8" "19" "30" "31" "42";
94          Connection 8 "9" "20" "21" "32" "43";
95          Connection 9 "10" "11" "22" "33" "44";
96      }
97  }
98  Compressor "top_U_compressor_ScanCompression_mode" {
99      UnloadSerializer "top_U_serializer_ScanCompression_mode" 0 1 2 3 4
5 6 7
100     8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 3
0 31 32
101     33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48;
102     ModeSerializer "top_U_deserializer_ScanCompression_mode" 56 57;
103     UnloadModeSerializer "top_U_deserializer_ScanCompression_mode" 10
11 12
104     13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36
105     37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53;
106     UnloadModeControl "top_U_deserializer_ScanCompression_mode" 10;
107     UnloadEnableSerializer "top_U_deserializer_ScanCompression_mode"
58;
108     UnloadSerializerDir "top_U_deserializer_ScanCompression_mode" 54;
109     CoreGroup core_group;
110     Modes 5;
111     Mode 0 {

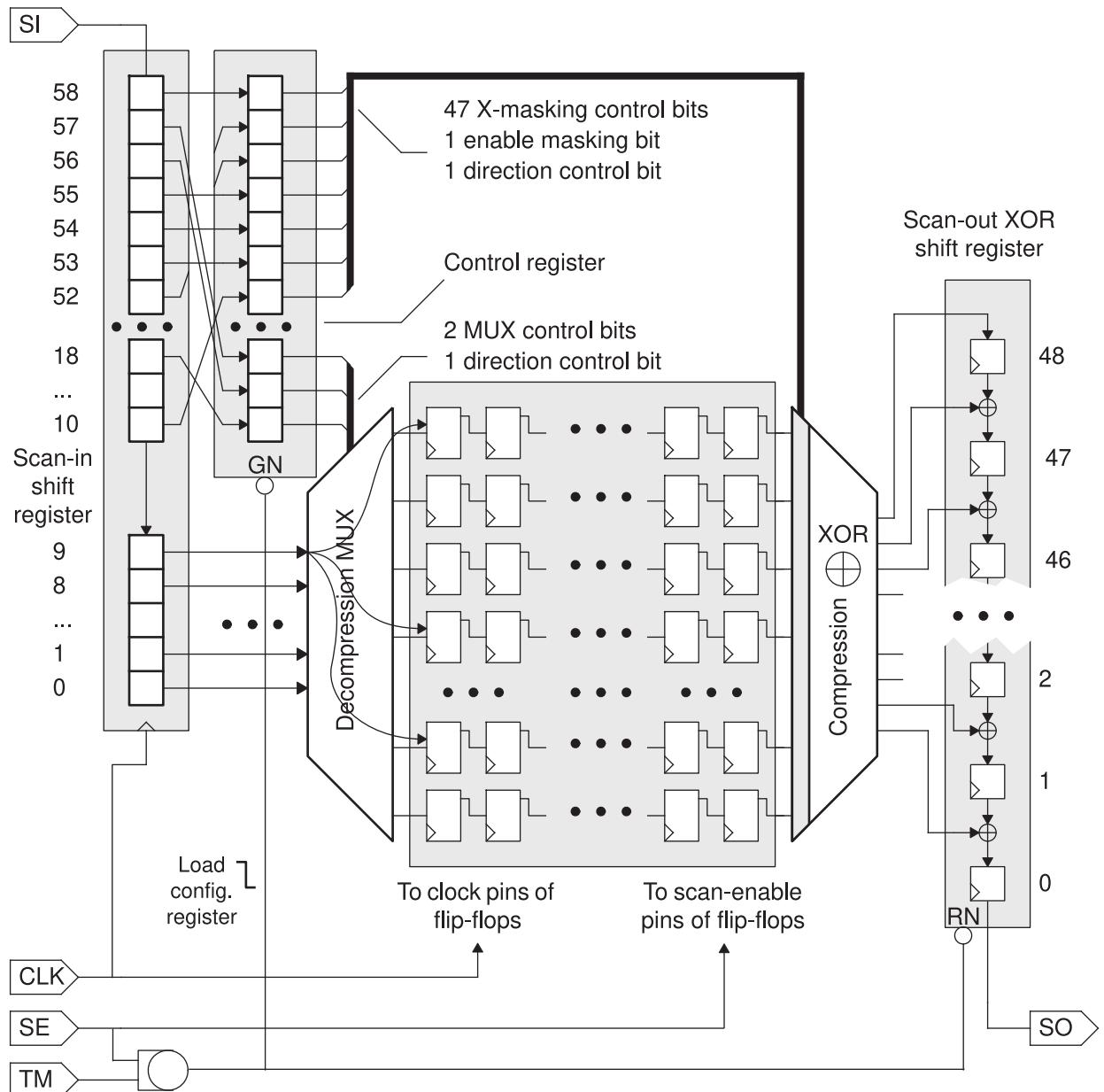
```

Chapter 27: DFTMAX Ultra STIL Protocol File Syntax

STIL Protocol File Example

```
112     ModeSerializerControls {
113         "top_U_deserializer_ScanCompression_mode"[58] = 0;
114     }
115     Connection "1" 0 17 37;
116     Connection "2" 1 18 38;
117     Connection "3" 2 19 39;
118     Connection "4" 3 20 40;
119     Connection "5" 4 21 41;
120     Connection "6" 5 22 42;
121     Connection "7" 6 23 43;
122     Connection "8" 7 24 44;
123     Connection "9" 8 25 45;
124     Connection "10" 9 26 46;
125     Connection "11" 10 27 47;
126     Connection "12" 11 28 48;
127     Connection "13" 0 12 29;
128     Connection "14" 1 13 30;
129     Connection "15" 2 14 31;
130     Connection "16" 3 15 32;
131     Connection "17" 4 16 33;
132     Connection "18" 5 17 34;
133     Connection "19" 6 18 35;
134     Connection "20" 7 19 36;
...
...
```

Figure 449 DFTMAX Ultra Compression Architecture for SPF Example



Line 1, `StreamingStructures`, starts the section of the STIL protocol file that describes the DFTMAX Ultra compression architecture for the device. Line 3, `ExternalCyclesPerShift`, contains the number of external shift cycles. This number is added to the maximum length of the internal chains to get the number of shifts per load operation. Line 5, `Length 59` under `LoadSerializer`, declares that the length of

the input shift register on the input side is 59 bits, whereas Line 9, `Length 49` under `UnloadSerializer`, declares that the length of the output shift register on the output side is 49.

Line 14, `Compressor ...`, starts the section that describes the architecture of the input decompression MUX. Lines 15 through 18 describe the functions of the scan-in shift-register bits used for scan-in data and for configuration of the decompression MUX. Bits 0 through 9, `LoadSerializer` bits, are the ten data bits that feed into the decompression MUX. Bits 56 and 57, `ModeSerializer` bits, control the mapping configuration of these ten bits to the scan chains. Line 18, the `LoadSerializerDir` bit, is a direction control bit that selects one of two ways to order the mapping, either from least to most significant bit or vice versa.

The DFT logic can be configured into multiple modes, so the STIL protocol file describes the characteristics of each mode in a separate section. Line 20, `Modes 4`, declares that there are four operating modes for the decompression MUX. Line 21, `Mode 0`, starts the section that describes Mode number 0.

Line 22, `UnloadModeSerializer ...`, declares that bits 10 through 53 of the scan-in shift register are used for configuring the output compression logic in Mode 0. Line 25, `ModeSerializerControls`, starts the section that describes the decompression MUX control bits. Lines 29 through 38, `Connection ...`, describe the mapping of the scan-in shift-register data bits to the scan chains.

Line 98, `Compressor ...`, starts the section that describes the architecture of the output compression XOR logic. Line 99, `UnloadSerializer`, declares that 49 bits are used in the scan-out XOR shift-register stages, designated bits 0 through 48. Lines 102 and 103, which are identical to lines 17 and 22, specify the usage of control bits in the scan-in shift register. Line 106, `UnloadModeControl`, and Line 107, `UnloadEnableSerializer`, specify that bit 10 and bit 58 control the different modes used for the application of output masking. Line 108, `UnloadSerializerDir`, specifies that bit 54 of the scan-in shift register is a direction control bit, which selects one of two ways to order the mapping of scan chains to the XOR compression logic.

Each successive `Mode` section in the compression section describes the usage of masking control bits from the scan-in shift register and the mapping of scan chains to the XOR compression logic for that particular mode.

28

DFTMAX Ultra Flow Naming Conventions

DFTMAX Ultra compression uses a structured naming convention for describing DFT flows. This naming convention uses a structured set of abbreviations that describes what type of logic exists before DFT insertion and what type of scan, compression, and DFT structures are created by DFT insertion. This chapter describes the naming convention. It also provides examples of how traditional DFT Compiler and DFTMAX flow names map to the new flow names.

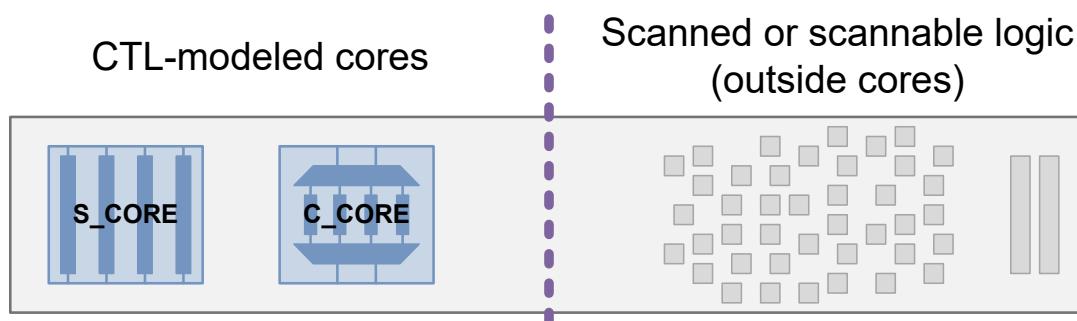
This chapter contains the following topics:

- [Describing Existing Logic](#)
- [Describing DFT Logic To Be Inserted](#)
- [Describing Additional DFT Features](#)
- [Scan Flow Mapping](#)

Describing Existing Logic

In a design, the existing sequential logic can be divided into two types, shown in [Figure 450](#).

Figure 450 Two Types of Existing Sequential Logic



CTL-modeled cores include any standard scan or compressed scan DFT-inserted blocks represented by a CTL model during DFT insertion, such as

- Full .ddc block netlists with CTL information
- Binary .ctlddc models created by the `write_test_model -format ddc` command
- ASCII .ctl models created by the `write_test_model -format ctl` command

Scanned or scannable logic includes any noncore logic to be included in scan chains after DFT insertion, such as

- Nonscan (but scannable) cells
- Test-ready scan cells
- Scan segments or complete scan chains defined with the `set_scan_path` command

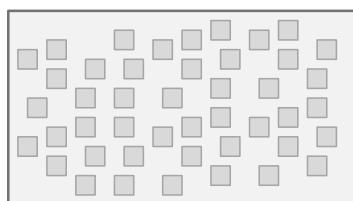
[Table 56](#) shows the flow name associated with each combination of sequential logic types.

Table 56 Flow Names Indicating Existing Sequential Logic Types

Scanned or scannable logic?	CTL-modeled cores?	Flow name	Description
			(Nothing to do)
X		TDI	Top-down insertion
	X	BUI	Bottom-up integration
X	X	MII	Mixed insertion and integration

The *top-down insertion* (TDI) flow type describes the case where no scan-inserted cores exist, but valid scanned or scannable cells exist to be stitched into newly inserted scan structures. [Figure 451](#) shows an example of the existing logic in a TDI flow.

[Figure 451 Existing Logic in a Top-Down Insertion \(TDI\) Flow](#)



Scannable logic

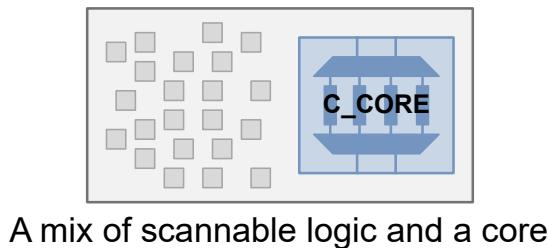
The *bottom-up integration* (BUI) flow type describes the case where scan-inserted cores exist to be integrated, and no valid scannable cells remain outside the cores. [Figure 452](#) shows examples of the existing logic in a BUI flow. The cores to be integrated are shown in blue.

Figure 452 Existing Logic in a Bottom-Up Integration (BUI) Flow



The *mixed insertion and integration* (MII) flow type describes the case where scan-inserted cores exist to be integrated, and valid scannable cells exist outside the cores to be stitched into newly inserted scan structures. [Figure 453](#) shows an example of the existing logic in an MII flow.

Figure 453 Existing Logic in a Mixed Insertion and Integration (MII) Flow



In the flows that contain cores, which are the BUI and MII flows, add square-bracket suffixes to indicate what types of cores exist:

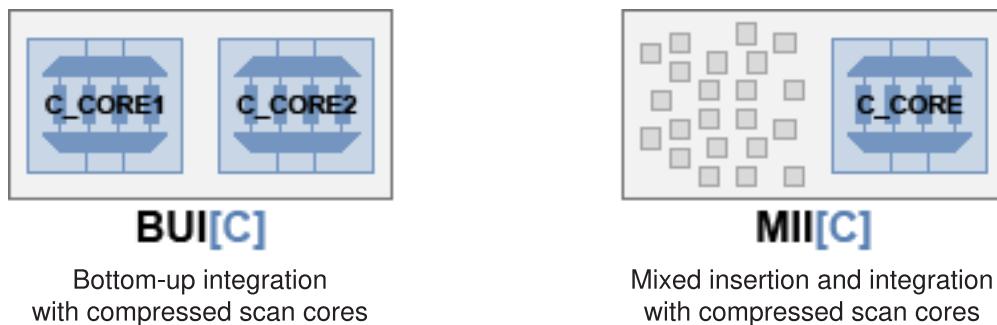
- If standard scan cores exist, add [S]:

Figure 454 [S] Flow Suffix for Existing Standard Scan Cores



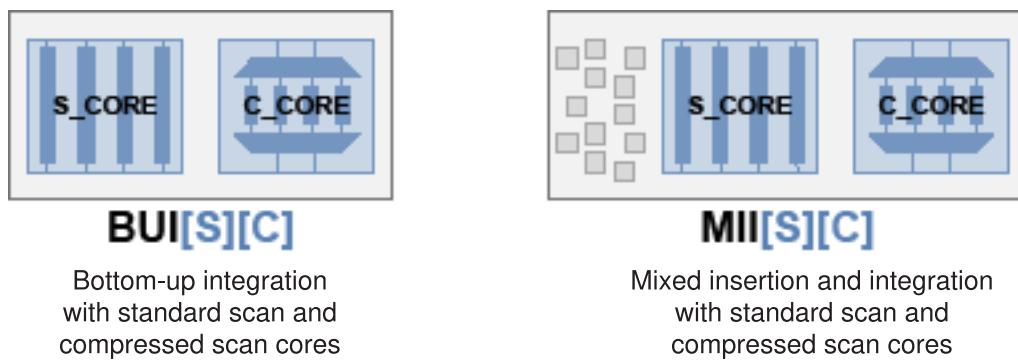
- If compressed scan cores exist, add [C]:

Figure 455 [C] Flow Suffix for Existing Compressed Scan Cores



- If both standard scan and compressed scan cores exist, add [S][C]:

Figure 456 [S][C] Flow Suffix for Existing Standard Scan and Compressed Scan Cores



The square brackets visually indicate that the suffixes represent cores.

Describing DFT Logic To Be Inserted

The previous section shows how the base flow describes what types of sequential logic already exist. This topic shows how dashed suffixes, appended to the base flow name, indicate what scan structures are to be inserted. Dashed suffixes represent scan characteristics applied to the design by DFT configuration commands and options.

If standard scan chains are created from scanned or scannable logic, add the -S suffix:

Figure 457 Inserting Standard Scan Chains in the TDI Flow

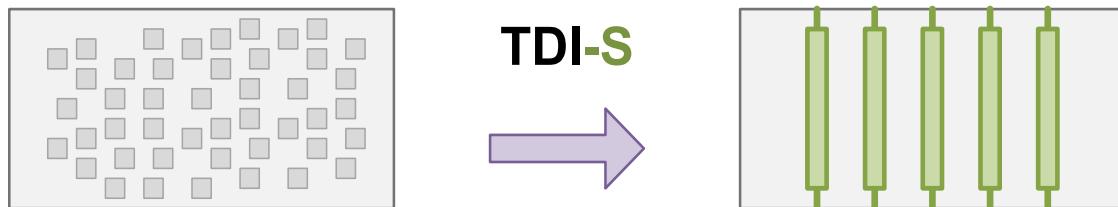


Figure 458 Inserting Standard Scan Chains in the MII[S] Flow



Figure 459 Inserting Standard Scan Chains in the MII[C] Flow



If a codec is inserted to compress scanned or scannable logic or standard scan cores, add the -C suffix:

Figure 460 Inserting Scan Compression in the TDI Flow

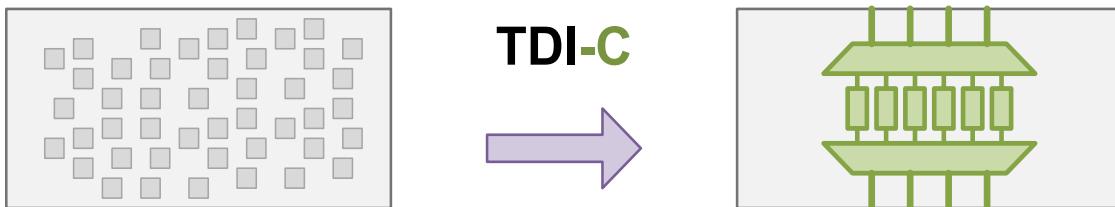


Figure 461 Inserting Scan Compression in the BUI[S] Flow

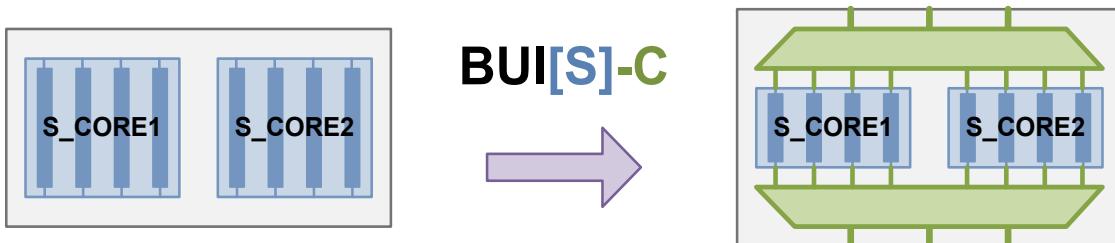


Figure 462 Inserting Scan Compression in the MII[S] Flow



Figure 463 Inserting Scan Compression in the MII[C] Flow



If only core scan connections are made in a BUI flow, no suffix is needed:

Figure 464 Making Core Scan Connections in the BUI[S] Flow

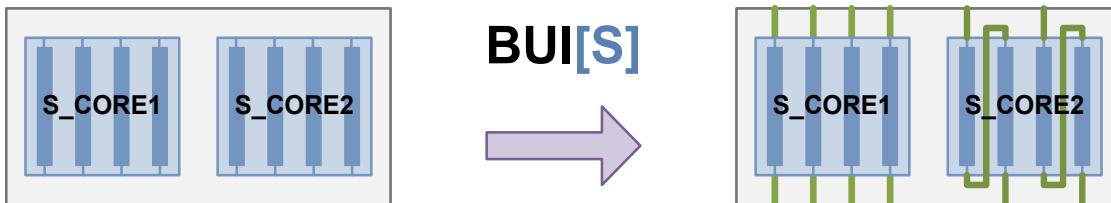
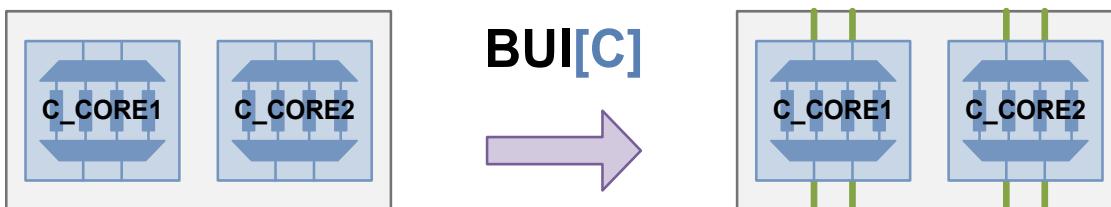


Figure 465 Making Core Scan Connections in the BUI[C] Flow



Describing Additional DFT Features

Use the optional conventions described in the following topics to describe additional DFT insertion features that can be used:

- [Partitions](#)
- [Scan I/Os](#)
- [Multiple Test Modes](#)
- [Additional Naming Convention Rules](#)

Partitions

DFT partitions are created with the `define_dft_partition` command. Use the following conventions to describe whether DFT partitions exist or are inserted in the flow.

Dashed suffixes indicate the scan characteristics applied during scan insertion. Add `-P` at the end of the flow name if partitions are used during insertion or integration.

[Figure 466](#) shows the TDI-C-P flow.

Figure 466 The TDI-C-P Flow

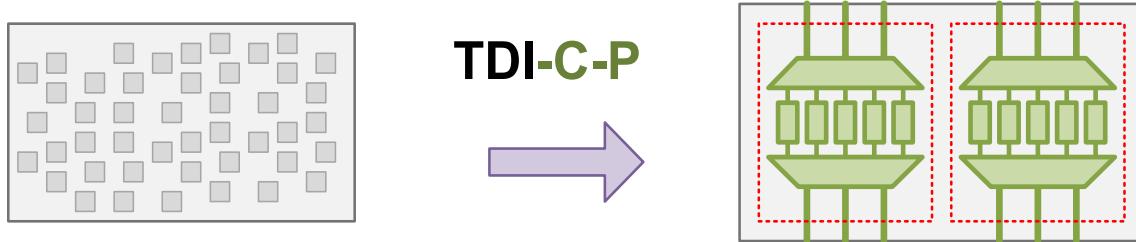
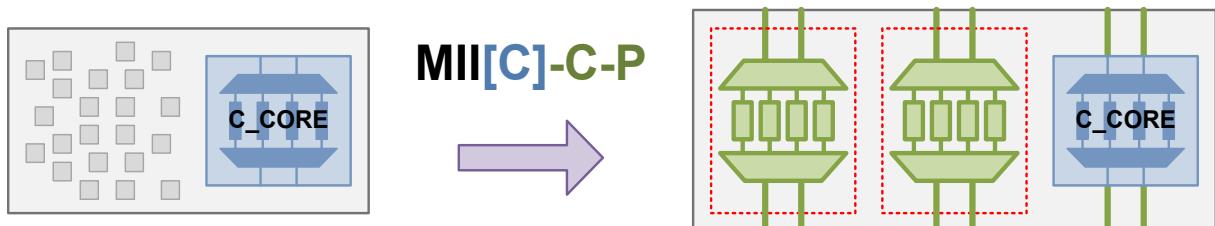


Figure 467 shows the MII[C]-C-P flow.

Figure 467 The MII[C]-C-P Flow

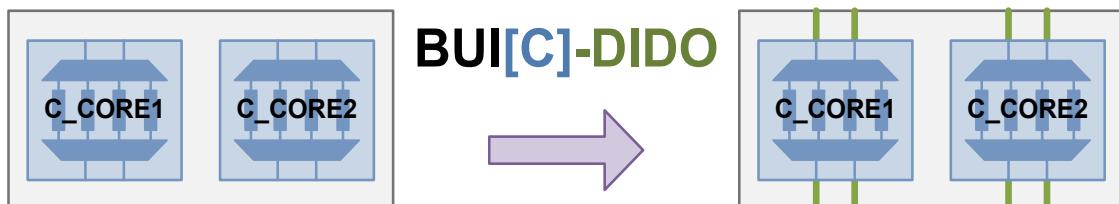


Scan I/Os

Use the following conventions to describe how the scan I/Os are connected to the scan structures in the flow:

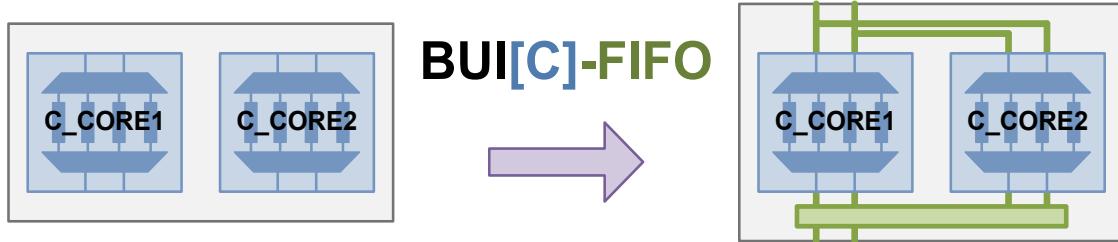
- Use DIDO (dedicated inputs, dedicated outputs) to describe scan I/O connections that are dedicated to each scan chain or core scan pin. [Figure 468](#) shows the BUI[C]-DIDO flow.

Figure 468 The BUI[C]-DIDO Flow



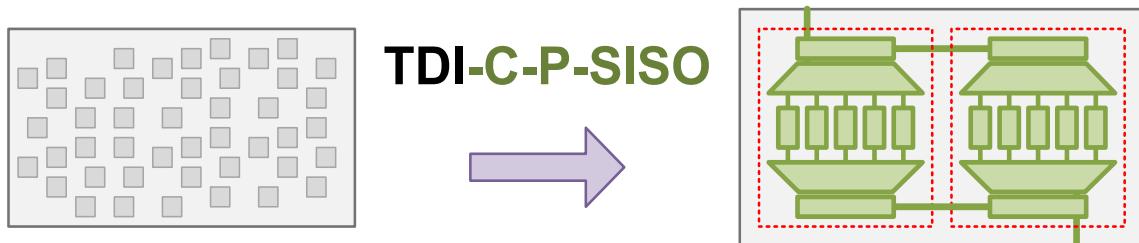
- Use FIFO (fanin, fanout) to describe scan I/O connections that have multiple connections in their fanin or fanout. [Figure 469](#) shows the BUI[C]-FIFO flow.

Figure 469 The BUI[C]-FIFO Flow



- Use SISO (serial input, serial output) to describe scan I/O connections that connect serially to multiple codecs. [Figure 470](#) shows the TDI-C-P-SISO flow.

Figure 470 The TDI-C-P-SISO Flow

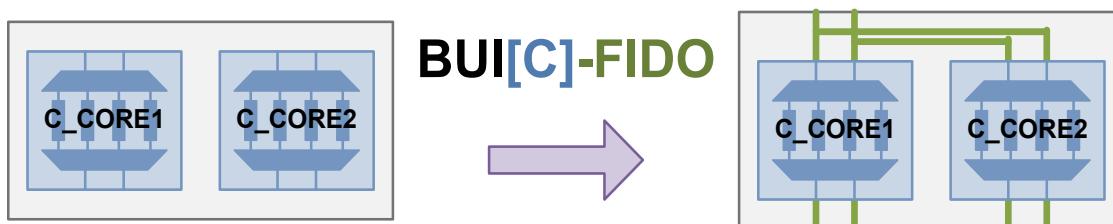


Note:

The SISO scan I/O connection type pertains only to sequential scan compression technologies, such as DFTMAX Ultra compression.

You can mix different scan connection types for scan inputs and scan outputs. [Figure 471](#) shows the BUI[C]-FIDO flow, which uses shared scan inputs and dedicated scan outputs.

Figure 471 The BUI[C]-FIDO Flow



Multiple Test Modes

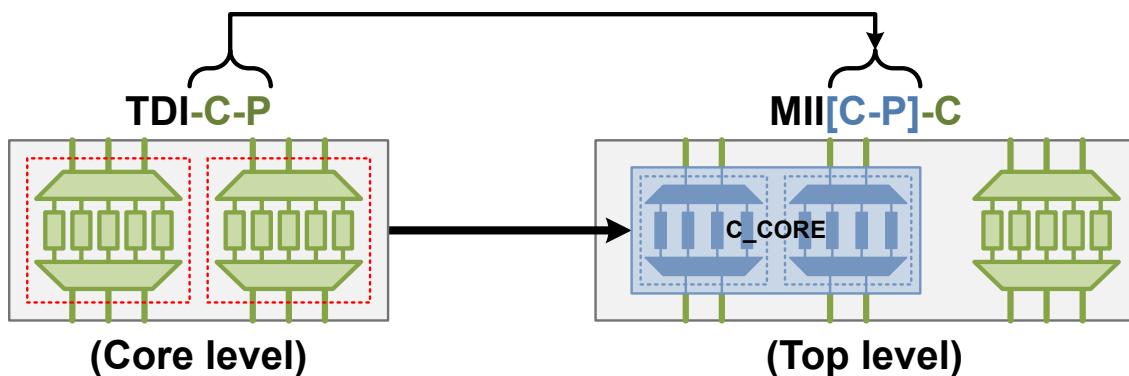
User-defined test modes are created with the `define_test_mode` command. Use the following conventions to describe how multiple user-defined test modes exist or are created in the flow:

- Add `-MM` at the end of the flow name if multiple user-defined test modes are defined in the current flow.
- Include `[-MM]` after the integration flow type if cores containing multiple user-defined test modes are integrated in the flow.

Additional Naming Convention Rules

Any dashed DFT feature suffixes at the core level can become square-bracketed feature descriptions at the integration level. In [Figure 472](#), compressed scan with partitions is inserted at the core level using a TDI-C-P flow. When this core is integrated, the core characteristics are captured parenthetically in the MII[C-P]-C flow name.

Figure 472 Integration of a [C-P] Scan-Inserted Core Created in a TDI-C-P Flow



When constructing a flow name, you only need to specify the flow and feature aspects needed for your reference. The flow name can be as specific or as general as you need it to be. If scan connections, partitions, or test modes are not relevant for a generalized flow you are describing, do not include them in the description.

Scan Flow Mapping

This topic contains several figures that show how the traditional DFT Compiler and DFTMAX flow names map to this flow name convention. The figure titles specify the traditional flow names and the figures specify the flow names using this naming convention.

Figure 473 The Top-Down Insertion Standard Scan Insertion Flow

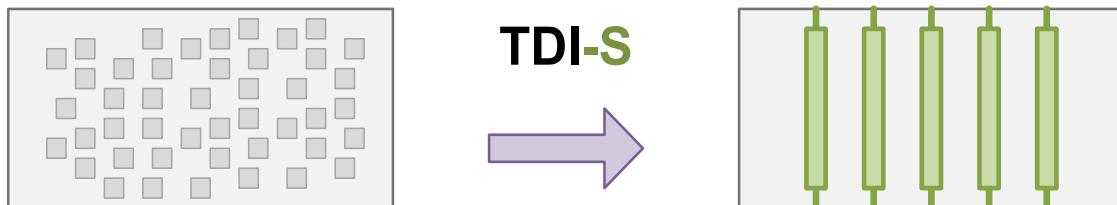


Figure 474 The Top-Down Insertion Compressed Scan Insertion Flow

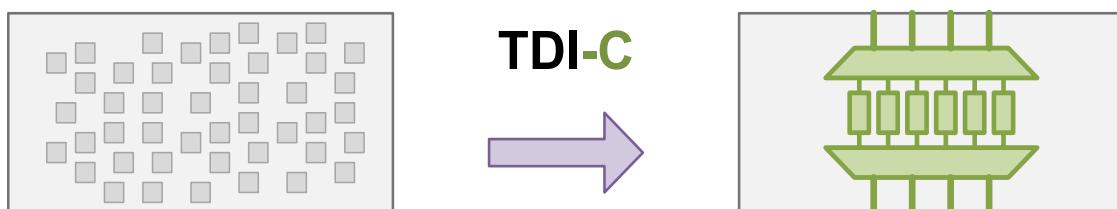


Figure 475 The Standard Scan HSS Integration Flow

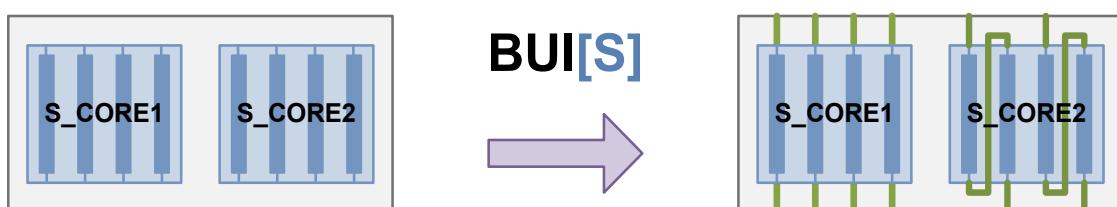


Figure 476 The Compressed Scan HSS Integration Flow

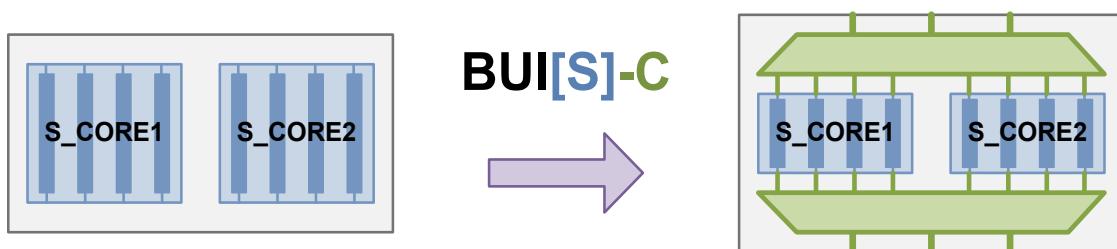


Figure 477 The HASS Integration Flow

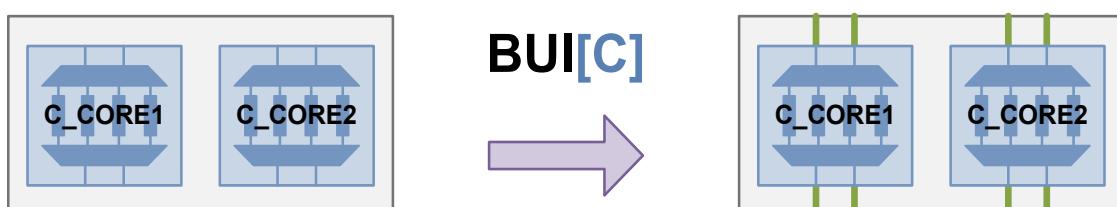
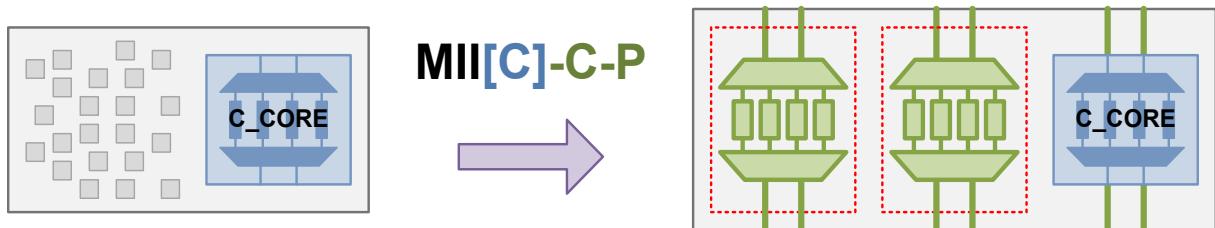


Figure 478 The Hybrid Integration Flow



Figure 479 The Hybrid Flow With Top-Level Partitions



Part 5: DFTMAX LogicBIST Self-Test

29

Introduction to LogicBIST

This chapter provides an introduction to the LogicBIST tool, which is a synthesis-based solution for in-system self-test of digital integrated circuits used in automotive, medical, and aerospace applications. LogicBIST addresses functional safety requirements set forth by standards such as ISO 26262 for the automotive semiconductor industry.

The following topics introduce LogicBIST self-test:

- [Introduction to LogicBIST](#)
 - [LogicBIST Requirements](#)
 - [The LogicBIST Flow](#)
-

Introduction to LogicBIST

Built-in self-test (BIST) capability enables a design to test itself autonomously without using external test data. The LogicBIST tool provides a low-overhead logic BIST (LBIST) solution for digital logic designs, such as automotive applications. The characteristics of this solution are:

- Low BIST controller area overhead
- Reuses the scan chain and test-mode control logic already implemented for manufacturing test
- Low self-test pin requirements
- Easy interface to functional logic
- Seed and expected signature values can be hardcoded or programmable
- Targets stuck-at and transition-delay faults
- Simple one-pass DFT insertion flow

LogicBIST Requirements

The LogicBIST flow requires the following:

- You must have the Design Compiler and TestMAX ATPG tools installed and licensed at your site.
- You must have the DFTMAX or TestMAX DFT tool licensed at your site.
- You must have an HDL Compiler license for compressed scan insertion.
- Blocks must be X-clean. See [Blocking Internal X Sources on page 1027](#).
- You must integrate the self-test logic into your design in one of the following ways:
 - Through signal connections to your functional logic
 - Through DFT-inserted IEEE 1500 logic

See Also

- [The LogicBIST Control and Data Signals on page 1012](#) for details on the control and status signals used
 - [Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces on page 377](#) for details on inserting IEEE 1500 logic
-

The LogicBIST Flow

From a high level, the standard LogicBIST flow can be summarized as follows:

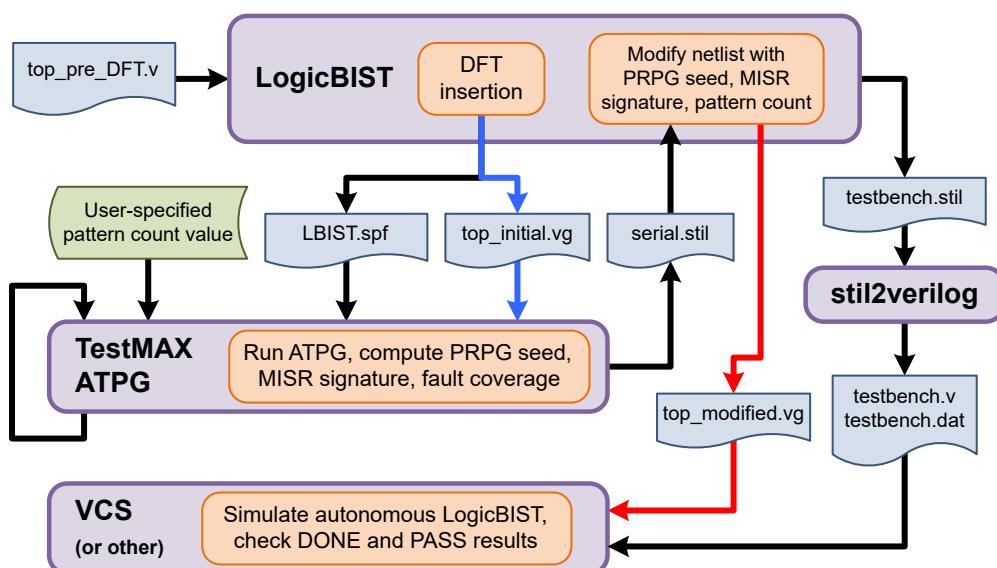
1. Insert the LogicBIST DFT logic in the design.
2. Use TestMAX ATPG to create self-test patterns for the design. TestMAX ATPG chooses a seed value for the design, then it computes the expected signature value for that seed value.
3. Write out an autonomous self-test testbench file, which simulates on-chip self-test.
4. Apply the bused seed, signature, and pattern count values computed by TestMAX ATPG to the design.
5. Simulate the resulting netlist and testbench in a Verilog simulator, such as VCS, to verify the correctness of autonomous BIST operation.

Constant-Driven Values

By default, the seed, signature, and pattern count values are driven by constants in the netlist. This results in the lowest area overhead, but it also requires that the netlist be modified to drive those values.

[Figure 480](#) shows the flow diagram for LogicBIST insertion, pattern generation, and verification. The original netlist (in blue) has the seed, signature, and pattern count values tied to all-zeroes, and the modified netlist (in red) contains the values computed by TestMAX ATPG.

Figure 480 The LogicBIST Flow With Constant-Driven Values



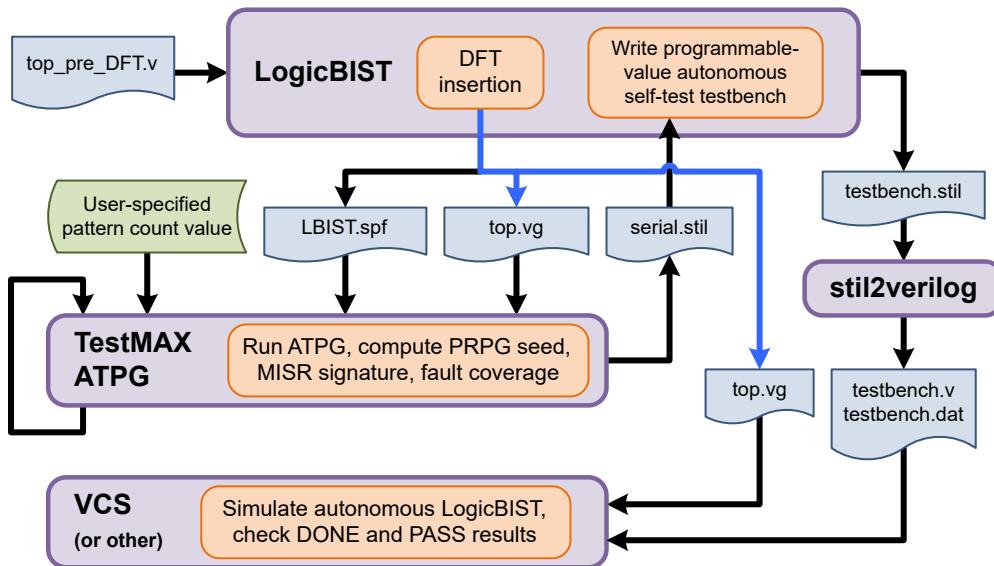
Programmable Values

You can also implement programmable seed, signature, and pattern count values. This can provide the following benefits:

- Test with multiple seed and signature pairs, for coverage-critical applications
- Divide self-test into many small segments over time, for time-critical applications
- No need for constant-value netlist modification for scan or functional logic changes

[Figure 481](#) shows this flow. In this case, no netlist modification step is needed, but you still generate the testbench files from the DFT environment.

Figure 481 The LogicBIST Flow With Programmable Values



Programmable values can be implemented in one of two ways:

- Using DFT-inserted IEEE 1500 logic
The tool creates the self-test protocol and testbench for you.
- Connecting self-test signals to internal functional registers using hookup pins
You must create your own self-test protocol and testbench for simulation.

See Also

- [Using Programmable LogicBIST Configuration Values on page 1062](#) for details on configuring programmable self-test logic
- [Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces on page 377](#) for details on inserting IEEE 1500 logic

30

The LogicBIST Architecture

LogicBIST self-test enables a design to test itself using the same scan chains already implemented for manufacturing test. It uses a pseudo-random pattern generator (PRPG) to create scan data, and a multiple-input signature register (MISR) to capture the design response. At the end of the test, if the actual signature matches the expected signature, the self-test asserts a PASS status.

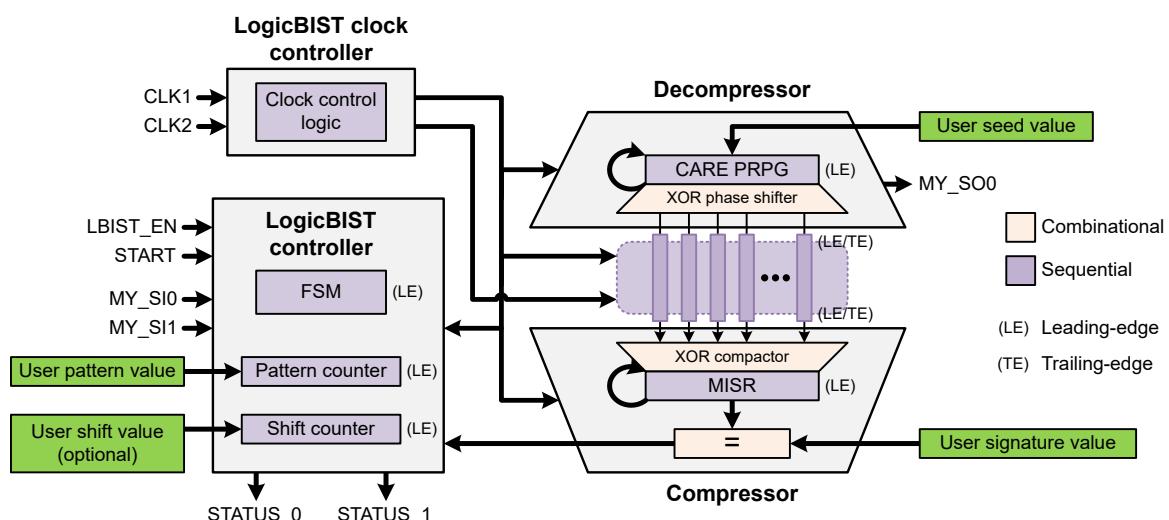
The following topics describe the LogicBIST architecture:

- [LogicBIST Architecture Overview](#)
- [LogicBIST Clock Control](#)
- [Isolating the Design During LogicBIST Self-Test](#)
- [Providing Testability for LogicBIST Self-Test](#)

LogicBIST Architecture Overview

The LogicBIST architecture consists of four components - LogicBIST controller, decompressor, compressor, and LogicBIST clock controller - as shown in [Figure 482](#).

Figure 482 The LogicBIST Architecture



The primary components are described in the following sections:

- [The LogicBIST Decompressor](#)
- [The LogicBIST Compressor](#)
- [The LogicBIST BIST Controller](#)
- [The LogicBIST Clock Controller](#)
- [The LogicBIST Control and Data Signals](#)

The LogicBIST Decompressor

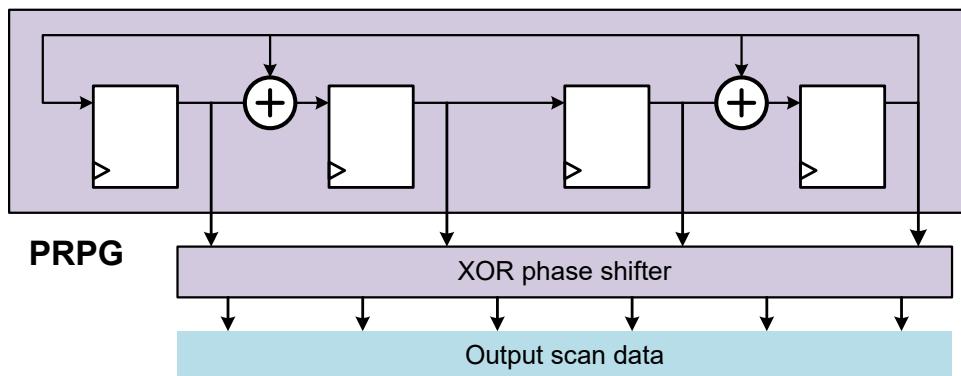
The LogicBIST decompressor feeds data into the compressed scan chains in the core logic. It is responsible for generating target fault care bits.

A PRPG, or pseudo-random pattern generator, is comprised of the following two components:

- A linear feedback shift register (LFSR) that generates the next data bit of the next data word as a linear XOR function of its current data word
- An XOR phase shifter that removes the correlations that result from the shift-register nature of the LFSR output taps

[Figure 483](#) shows a simple example PRPG register.

Figure 483 An Example PRPG Register



The PRPG operates as follows:

- When the design is operating in mission mode, the PRPG is idle and is not clocked.
- When the design is in a non-LogicBIST scan mode, the PRPG operates as scannable design logic so that the decompressor logic can be scan-tested.
- During LogicBIST operation,
 - At the beginning of the test program, the user-specified seed value is parallel-loaded into the PRPG in a single clock cycle.
 - During the test program, the PRPG generates a new pseudorandom data word in each clock cycle, which is used to generate scan data for the compressed scan chains.

Note:

For simplicity, the control logic that implements these modes of operation is not shown in [Figure 483](#).

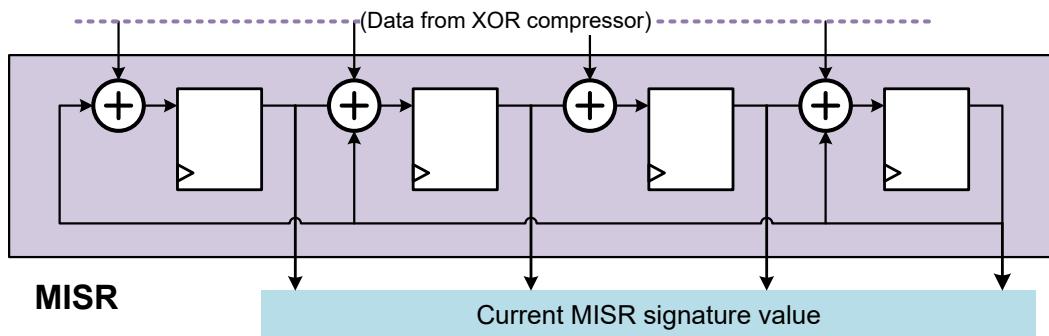
After the PRPG is loaded with a seed value and clocked, it generates a stream of data values that appear to be random values, but are actually a function of that seed value. Each seed value produces a stream of data values unique to that seed value.

The LogicBIST Compressor

The LogicBIST compressor receives and compresses data from the internal chains during the unload process. It consists of an XOR-tree compressor and a multiple-input signature register (MISR). The XOR compressor has no X-tolerance masking.

In the MISR, each register input captures an XOR of the previous register's input and a data input signal from the XOR compressor to the MISR. [Figure 484](#) shows a simple example MISR.

Figure 484 An Example MISR Register



The MISR operates as follows:

- When the design is operating in mission mode, the MISR is idle and is not clocked.
- When the design is in a non-LogicBIST scan mode, the MISR operates as scannable design logic so that the compressor logic can be scan-tested.
- During LogicBIST operation,
 - The MISR is reset when the design enters the LogicBIST operation mode. The MISR now has an initial known signature value (all zeros).
 - For the first pattern, the MISR remains unclocked because the unloaded scan data is unknown.
 - For the second and subsequent patterns, the MISR is clocked. The MISR captures values from the XOR compressor in each shift clock cycle and incorporates it into the current signature value of the MISR.

During the test program, the sequence of MISR values is dependent on the scan data that it captures. At the end of the test program, the signature value of the MISR is compared against the user-specified expected signature value, and the STATUS_* signals are set to indicate test completion and pass/fail status.

The LogicBIST BIST Controller

The LogicBIST BIST controller contains the following:

- A small finite state machine (FSM) that controls BIST operation.
- A pattern counter that applies the user-specified number of test patterns.
- A shift counter that applies the correct number of shift clock cycles for each test pattern. For each completed sequence of the shift counter, the pattern counter decrements by one.

The LogicBIST controller operates as follows:

- When the design is in mission mode and LogicBIST is disabled, the LogicBIST controller is idle and its clock is disabled. The FSM registers are held in reset (unless mission mode is overloaded onto a scan mode).
- When the design is in a non-LogicBIST scan mode, the pattern counter and shift counter operate as scannable design logic so that their logic can be scan-tested. The FSM flip-flops are excluded from scan testing so any OCC, ICG, reset, or scan-enable control logic does not interfere with scan testing.

- When the design is in mission mode and LogicBIST is enabled,
 - The LogicBIST controller controls the scan-enable and wrapper-shift signals in the design.
 - At the beginning of the test program, the FSM initializes the decompressor PRPG and compressor MISR to their initial states, and it loads the pattern and shift counters to their user-specified initial values.
 - During the test program, the LogicBIST controller runs the pattern and shift counters through their sequences. As the shift counter counts through its sequence, the scan chains perform load/unload using the PRPG/MISR, respectively. When the shift counter reaches zero, the FSM issues the capture cycle(s), decrements the pattern counter, and begins a new shift counter sequence.
 - When the pattern counter reaches zero, the current MISR signature value is compared with the user-specified expected signature value. If they match, the test passes; if not, the test fails.

See Also

- [Previewing and Inserting the LogicBIST Implementation on page 1043](#) for information on determining the mission-mode and self-test-mode encodings

The LogicBIST Clock Controller

The LogicBIST clock controller operates as follows:

- When the design is in mission mode, the clocks operate normally.
- When the design is in a non-LogicBIST scan mode, the clocks operate normally.
- When LogicBIST self-test is active,
 - The clock controller gates the clock signal to the functional design logic as directed by the LogicBIST controller.
 - A free-running *BIST clock* must be available, running at the desired scan frequency, for the duration of the LogicBIST test operation.

LogicBIST self-test supports multiple clock configurations, each of which uses its own clock controller logic structure. For more information, see [LogicBIST Clock Control on page 1015](#).

The LogicBIST Control and Data Signals

The LogicBIST-specific control and data signals in a LogicBIST implementation are described in the following topics:

- [The LogicBIST Operational Modes](#)
- [The LBIST_EN and START Signals](#)
- [The STATUS_0 and STATUS_1 Signals](#)
- [The Scan-In and Scan-Out Signals](#)

The LogicBIST Operational Modes

LogicBIST self-test can operate in the following modes:

- *ATPG mode*—This mode is used only for core-level seed and signature computation in TestMAX ATPG. TestMAX ATPG accesses state elements (via a scan chain) through core-level scan ports during this process; the LogicBIST FSM is unused.

This mode is activated when the LogicBIST test-mode encoding is applied. The LBIST_EN and START signals are not used.

- *Autonomous mode*—This mode can be used after the seed and signature values have been applied to the design. All BIST operations perform autonomously, as controlled by the LogicBIST FSM. This is the mode that is simulated and ultimately used in silicon operation.

This mode is activated when the LBIST_EN and START signals are asserted while the mission-mode test-mode encoding is applied.

The LBIST_EN and START Signals

The LBIST_EN and START signals work together as follows:

- The LBIST_EN signal is used only when the mission-mode test-mode encoding is applied. When this signal is asserted in mission mode, the design enters autonomous LogicBIST operation mode. Any DFT logic associated with the LogicBIST self-test mode (wrapper chains, test points, and so on) is enabled.
- When the START signal is de-asserted (regardless of the state of the LBIST_EN signal), the LogicBIST FSM is unclocked and asynchronously held in reset to the idle state.
- When the LBIST_EN signal is asserted while the START signal is de-asserted, the pattern counter and MISR are reset to all-zeros.
- LogicBIST self-test begins when the START signal is asserted while the LBIST_EN signal is already asserted. The test runs to completion as long as the START signal

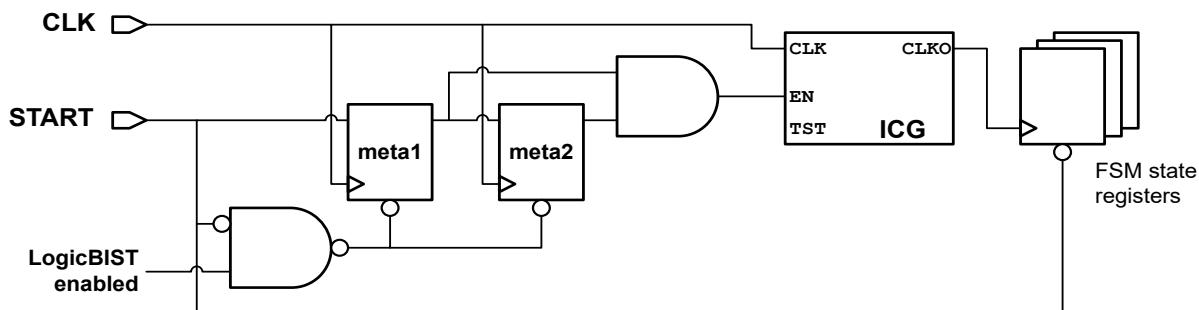
remains asserted. If the START signal is de-asserted during the test, the test halts and the LogicBIST logic returns to its idle state.

- When the LBIST_EN signal is de-asserted, any DFT logic associated with the LogicBIST self-test mode (wrapper chains, test points, and so on) is disabled.

Because all reset signals inside the LogicBIST IP are generated from the START signal, no connection to a functional reset signal is needed.

The START signal is synchronized to the BIST clock, as shown in [Figure 485](#), to avoid metastability issues. Due to the synchronizer delay, the pattern counter and MISR are reset even if the LBIST_EN and START signals are asserted at the same time. The metastability registers are included in scan testing.

Figure 485 Synchronization of the START Signal to the BIST Clock



These signals are used in autonomous mode.

See Also

- [Enabling DFT Logic During Autonomous Self-Test](#) on page 1024 for more information on how test-mode signals are used in a LogicBIST design

The STATUS_0 and STATUS_1 Signals

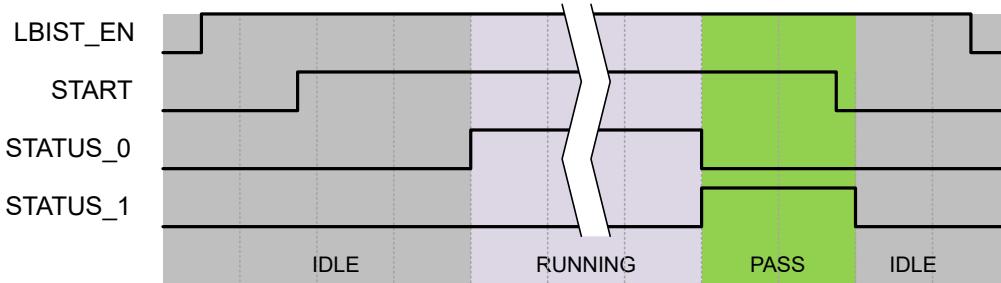
The STATUS_0 and STATUS_1 signals indicate the status of autonomous self-test. When the LBIST_EN signal is asserted, the two-bit bus {STATUS_1, STATUS_0} has the following possible values:

- 00: LogicBIST logic idle or inactive
- 01: LogicBIST test running
- 10: LogicBIST test complete and passed
- 11: LogicBIST test complete and failed

These signals are used in autonomous mode.

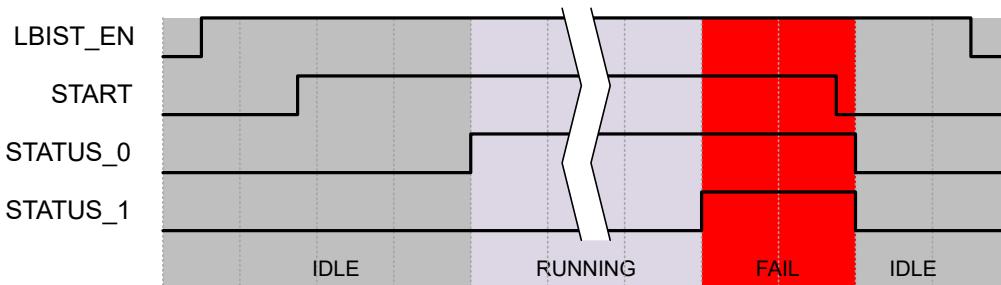
[Figure 486](#) shows the status signal behavior when self-test completes and passes. The passing status is held until START is de-asserted.

Figure 486 Status Signal Behavior When Self-Test Completes and Passes



[Figure 487](#) shows the status signal behavior when self-test completes and fails. The failing status is held until START is de-asserted.

Figure 487 Status Signal Behavior When Self-Test Completes and Fails



The status signals are组合ally derived from the self-test FSM state. For details on monitoring them from your functional logic, see [Monitoring the Self-Test Status Signals on page 1055](#).

The Scan-In and Scan-Out Signals

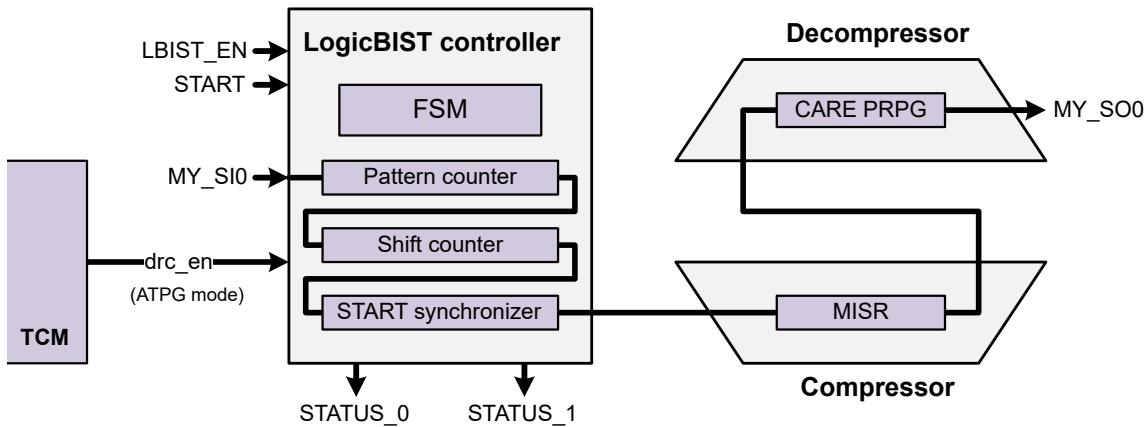
A LogicBIST implementation requires at least one user-defined scan-in signal. (For designs with IEEE 1500 logic, the WSI signal serves this purpose.)

When the design is in ATPG mode, key LogicBIST registers (pattern counter, shift counter, START synchronizer, PRPG, and MISR) are placed in a scan chain driven by the first user-defined scan-in signal. This allows TestMAX ATPG to access the registers during LogicBIST seed and signature computation.

For other test modes, this scan-in port is used as a regular scan-in port.

[Figure 488](#) shows how TestMAX ATPG accesses the scannable LogicBIST access chain in ATPG mode.

Figure 488 Scan Chain Access to the LogicBIST Logic



See Also

- [Enabling DFT Logic During Autonomous Self-Test](#) on page 1024 for details on how the drc_en (ATPG mode) signal is generated

LogicBIST Clock Control

LogicBIST clock control is described in the following topics:

- [Overview of Clock Configurations](#)
- [External Clocks](#)
- [OCC-Controlled Clocks With Default Capture Behavior](#)
- [OCC-Controlled Clocks With Weighted Clock Capture Groups](#)
- [External and Internal Clocks in the Same Design](#)

Overview of Clock Configurations

LogicBIST self-test supports the following three clock configurations:

- External (port-driven) clocks

Use this configuration when all clocks are driven by input ports and there are no on-chip clocking (OCC) sources, such as phase-locked loops (PLLs).
- OCC-controlled clocks with default capture behavior

Use this configuration when there is a single OCC-controlled clock or multiple OCC-controlled clocks that do not interact during capture.

- OCC-controlled clocks with weighted clock captures

Use this configuration when there are multiple OCC-controlled clocks that interact or if there is an asynchronous set or reset signal in your design.

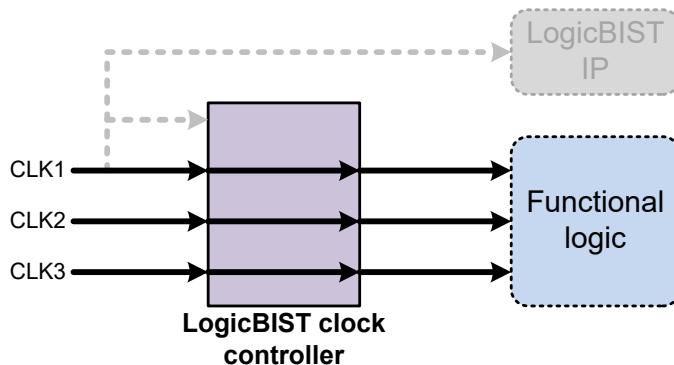
If there is minimal communication between asynchronous clock domains, you can use test points to block the capture paths between those domains. This can reduce or eliminate the need for clock weights.

All clocks in the design must use the same configuration. If you have a mix of external and OCC-controlled clocks, you must use an OCC controller for the external clocks. If you have an asynchronous set or reset in your design, you must disable it or use weighted clock captures—even for a single clock.

External Clocks

If the design has no on-chip clocking (OCC) sources, then all clocks are external (driven by input ports). [Figure 489](#) shows how the clock controller passes all clocks transparently when LogicBIST self-test is inactive.

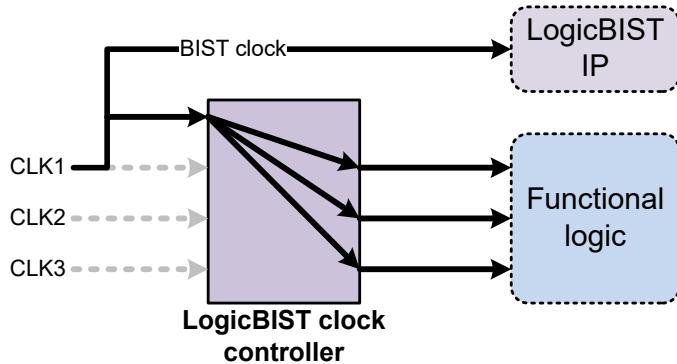
Figure 489 External Clocks When LogicBIST Self-Test Is Inactive



When LogicBIST self-test is active, if the design contains multiple scan clock domains, the clock controller drives all scan clock domains with a single BIST clock. The non-BIST clock input ports do not clock any scan chains.

[Figure 490](#) shows how the clock controller drives all scan clock domains with the LogicBIST clock when LogicBIST self-test is active (gated under the control of the LogicBIST controller). CLK1 is selected as the BIST clock.

Figure 490 External Clocks When LogicBIST Self-Test Is Active

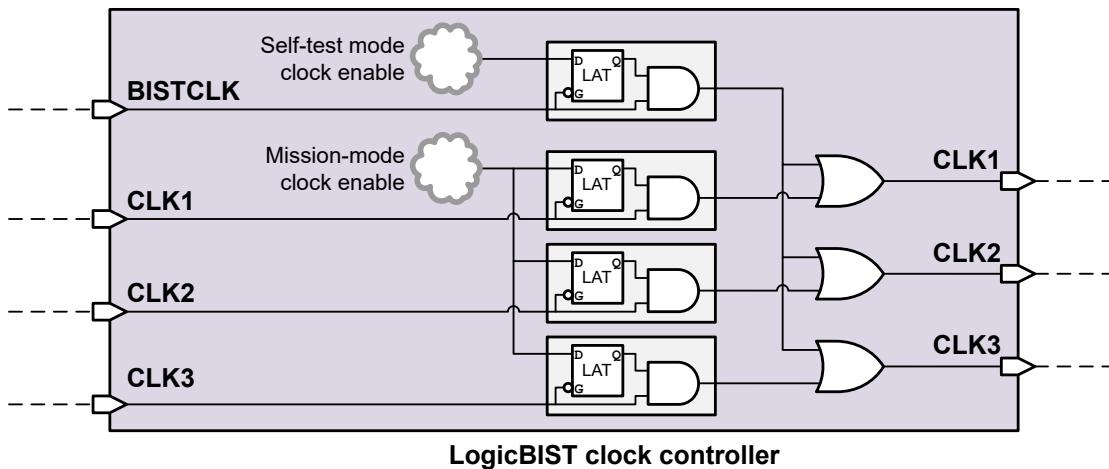


Caution:

When the design contains multiple external clocks, you must ensure that cross-domain paths meet timing in LogicBIST mode because the clock trees are driven by a single port but might have different latencies.

The LogicBIST clock controller logic structure is shown in Figure 491. (The figure is intended to show the logic function; actual implementation might vary.)

Figure 491 External Clock Controller Structure



OCC-Controlled Clocks With Default Capture Behavior

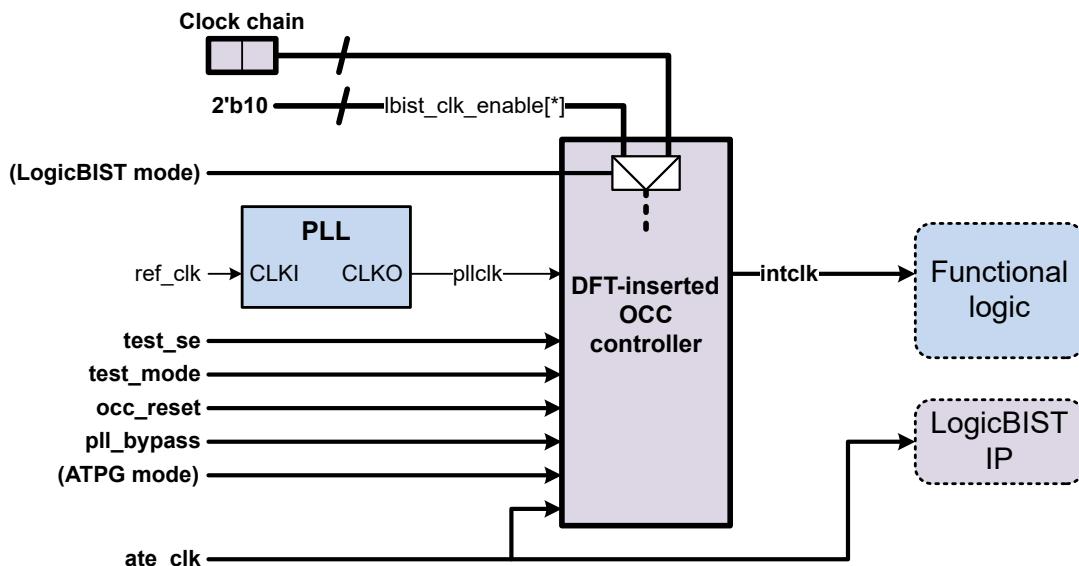
When you have a DFT-inserted OCC controller in your design, the tool uses an OCC controller design with additional LogicBIST clock control logic. The ATE clock is used as the BIST clock.

In autonomous mode, the OCC controller operates normally, except that the clock pulses are determined by a pulse pattern signal (`lbist_clk_enable[]`) instead of the clock chain. The width of this bus is the same as the clock chain length. By default, the clocks use a single-pulse pattern. You can also specify a programmable pattern, as described in [Using Programmable LogicBIST Configuration Values on page 1062](#).

In ATPG mode, the `pll_bypass` signal of the OCC controller is asserted to use the ATE clock for TestMAX ATPG.

The OCC controller logic is shown in [Figure 492](#).

Figure 492 LogicBIST OCC Controller



If you have multiple OCC-controlled clocks in your design, all clocks capture in each pattern. If capture paths exist between the clock domains, you must use weighted-clock captures as described in the next section, or you must block the capture paths using RTL logic or test points.

Some limitations apply to designs with OCC controllers. See [Chapter 33, LogicBIST Limitations and Known Issues](#).

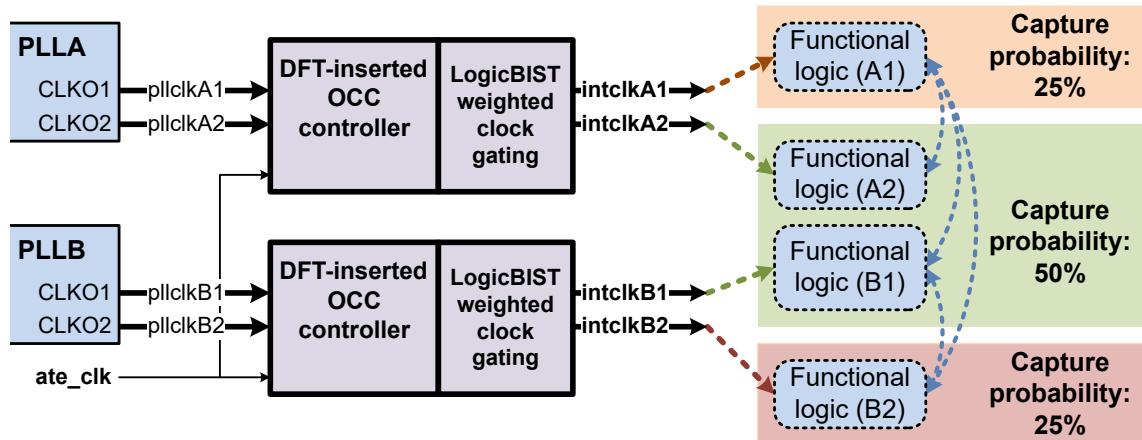
OCC-Controlled Clocks With Weighted Clock Capture Groups

By default, all OCC clocks capture in each LogicBIST pattern. If capture paths exist between clock domains, additional logic is required to selectively enable non-interacting capture clocks in each pattern. This avoids capturing an X value from an asynchronous clock domain that is also clocked in that pattern.

To implement this logic, you separate clocks into groups and assign a weight to each group. In each pattern, a single clock group is selected for capture, proportionally to the weight values.

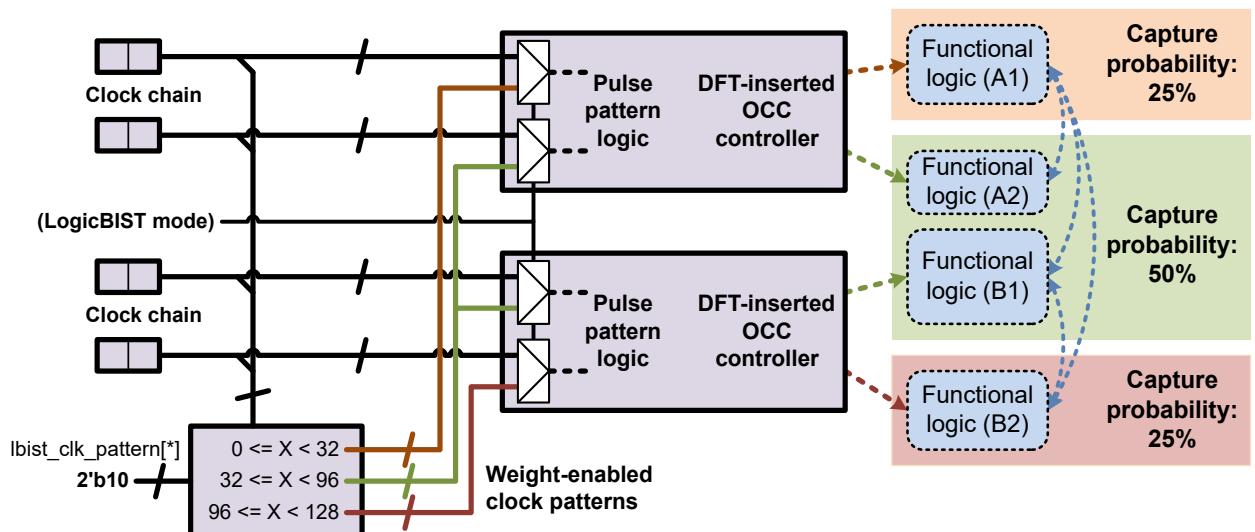
In the following example, clock domains A2 and B1 do not interact with each other and can be grouped together. Clock domains A1 and B2 have less logic than A2+B1 and can have a lower weight than A2+B1 to capture less often.

Figure 493 LogicBIST OCC Controller With Weighted Capture Groups



Because LogicBIST does not use the clock chain registers, they are repurposed for clock group selection during LogicBIST self-test. In each pattern, the clock chains load a pseudorandom value from the PRPG. This value feeds a weighted clock group selector that enables the pulse pattern for one of the capture groups, as shown in [Figure 494](#).

Figure 494 Weighted Capture Groups Logic Structure



The comparator value is seven bits. Thus, you must have at least seven clock chain bits in the design; additional bits are not used for clock selection. In addition, all OCC controller clocks must have the same clock chain length.

The clock pulses are determined by a pulse pattern signal instead of the clock chain. The width of this bus is the value specified with the `-cycles_per_clock` option of the `set_dft_clock_controller` command. By default, the clocks use a single-pulse pattern. You can also specify a programmable pattern, as described in [Using Programmable LogicBIST Configuration Values on page 1062](#).

[Table 57](#) shows the minimum clock chain length as a function of clock count.

Table 57 Minimum Total Clock Chain Length in a LogicBIST Design

Number of OCC clocks	Minimum clock chain length
2	4
3	3
4 to 6	2
7 or more	1

Additional limitations apply to designs with OCC controllers. See [Chapter 33, LogicBIST Limitations and Known Issues](#)."

See Also

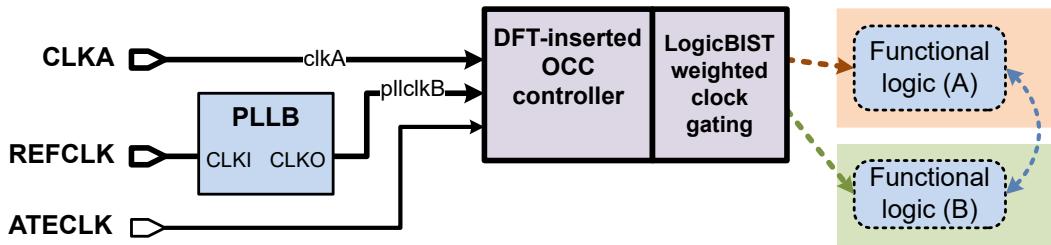
- [Configuring Clock and Reset Weights on page 1036](#) for details on configuring clock weights
- [Simplifying the Weighted Clock/Reset Logic on page 1065](#) for details on simplifying the comparator logic

External and Internal Clocks in the Same Design

If your design uses both external (port-driven) and internal (OCC-controlled) clocks, you must control the external clocks with a DFT-inserted OCC controller. If the clock domains have capture interactions, you must also use weighted clock capture groups.

[Figure 495](#) shows a design with one external clock and one internal clock using weighted clock capture groups.

Figure 495 Using OCC Control for External Clocks



For more information, see [Specifying OCC Controllers for External Clock Sources](#) on page 550.

Isolating the Design During LogicBIST Self-Test

When LogicBIST self-test is active, it generates and applies the test data autonomously (on-chip). ATE data is not available to control the design input ports, and the ATE does not observe the design output ports.

As a result, the design must be isolated on-chip during LogicBIST self-test. Its inputs must be controlled to avoid X capture; its outputs should be observed to ensure coverage.

The following topics describe two design isolation methods:

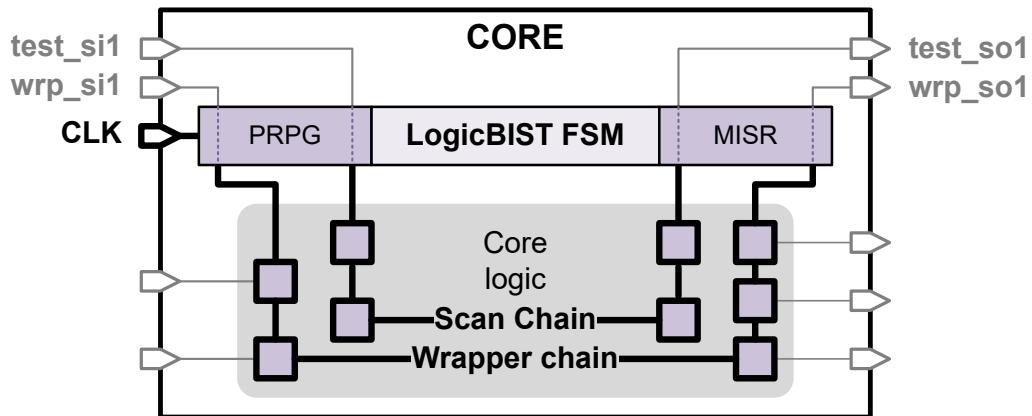
- [Isolating the Self-Test Design Using Core Wrapping](#)
- [Isolating the Self-Test Design Using Test Points](#)
- [Comparing the Two Isolation Approaches](#)

Isolating the Self-Test Design Using Core Wrapping

You can use the core wrapping feature to insert a wrapper chain that isolates the design during self-test. Wrapper chains inherently provide this needed isolation.

With this approach, the LogicBIST test mode becomes an inward-facing mode that drives LogicBIST-generated data into the input wrapper chain and incorporates the captured output wrapper chain data into the MISR, as shown in [Figure 496](#).

Figure 496 Isolating the Self-Test Design Using Core Wrapping



If your design already implements wrapper chains, you use this approach by default.

If most of the I/O ports in your design are registered, you can reuse the existing I/O registers to build the wrapper chain, which minimizes area. This is called the maximized-reuse flow.

If the self-test design drives top-level logic that cannot tolerate pseudorandom output data during self-test, you can specify safe values to be driven at the outputs during self-test.

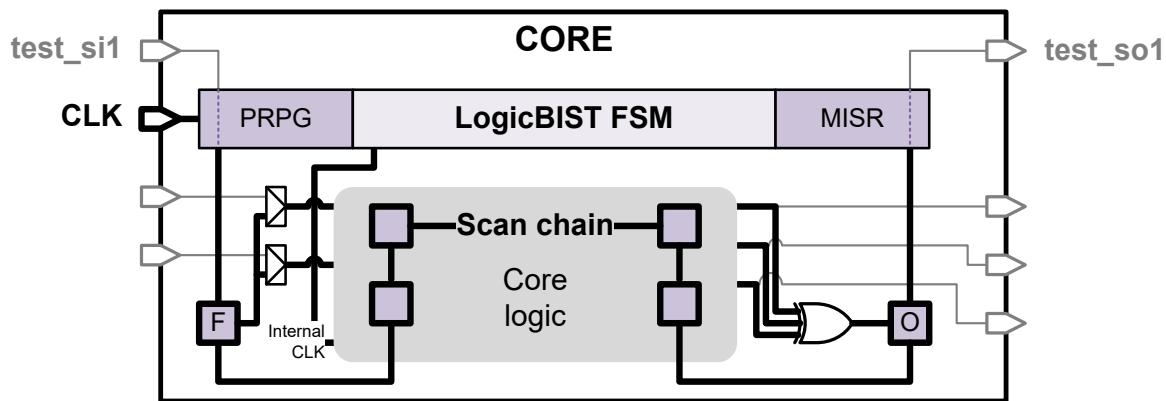
See Also

- [Configuring Wrapper Chain Isolation Logic on page 1037](#) for details on using wrapper chains for self-test isolation

Isolating the Self-Test Design Using Test Points

To reduce area, you can use test points instead of core wrapping to provide boundary testability. The `core_wrapper` target of automatic test point insertion inserts `force_01` test points at the inputs and `observe` test points at the outputs, as shown in [Figure 497](#).

Figure 497 Isolating the Self-Test Design Using Test Points



Multiple test points can share a single test point register, which reduces the area.

Use this isolation method only if your design is not already core-wrapped.

See Also

- [Configuring Test Point Isolation Logic on page 1039](#) for details on using test points for self-test isolation

Comparing the Two Isolation Approaches

[Table 58](#) compares the two isolation approaches.

Table 58 Comparison of the Two Self-Test Isolation Approaches

	Core wrapping	Test points
Primary benefit	“Free” for core-wrapped designs	Minimal area for unwrapped designs
Controlled by	Current test-mode encoding	Dedicated test-point control signal
Provides INTEST?	Yes	Yes
Provides EXTEST?	Yes	No
Provides transparent (unisolated) ATPG mode?	Optional (see Top-Down Flat Testing With Transparent Wrapped Cores on page 511)	Yes (deassert the isolation test-point control signal)

In general,

- Use core-wrapping isolation if your design already uses core wrapping for hierarchical test reasons.
- Use test-point isolation if your design does not require core-wrapping capabilities.

Providing Testability for LogicBIST Self-Test

During LogicBIST self-test, no X values can be captured during testing. If an X value reaches the MISR, it recirculates and spreads within the MISR, corrupting its value. The following topics describe ways to keep the design clean of X values:

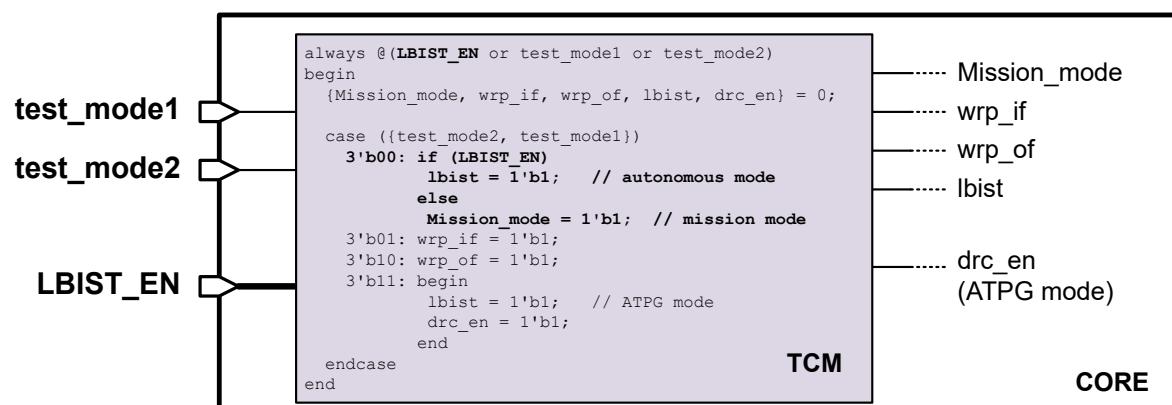
- [Enabling DFT Logic During Autonomous Self-Test](#)
- [Blocking Internal X Sources](#)
- [Ensuring Testability for Reset Signals](#)
- [Ensuring Testability for Integrated Clock-Gating Cells](#)

Enabling DFT Logic During Autonomous Self-Test

When a LogicBIST core performs autonomous self-test, its LBIST_EN signal is asserted while its test-mode signals are set to the mission-mode encoding. To ensure that testability logic inside the core is enabled during autonomous self-test, the LogicBIST test-mode output of the test control module (TCM) is asserted during self-test:

[Figure 498](#) shows the TCM logic for a core-wrapped design.

Figure 498 Test Control Module (TCM) in a Core-Wrapped LogicBIST Design



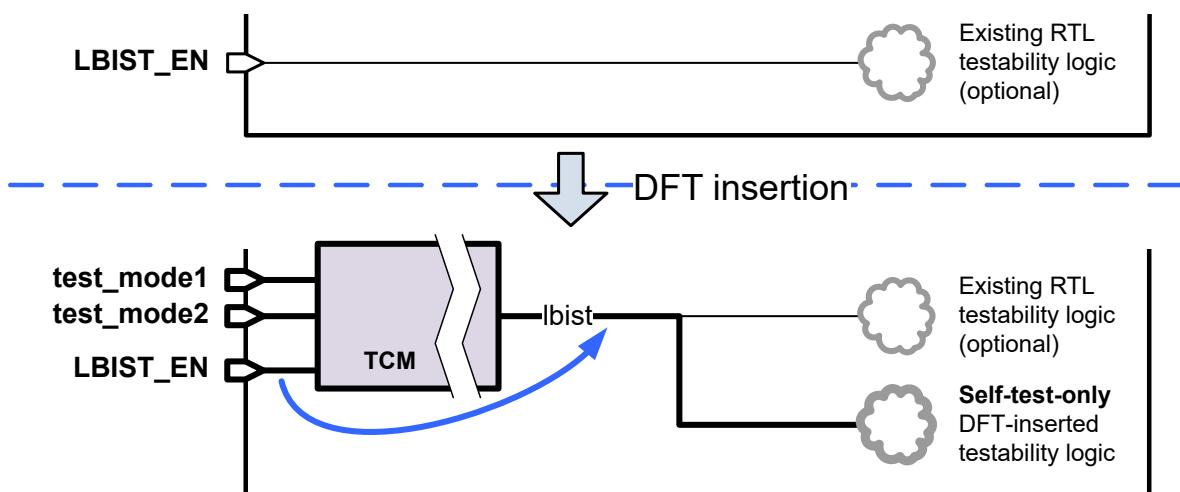
The TCM self-test output enables all testability logic controlled by the DFT-inserted TCM, such as reconfiguration MUXs, wrapper chains, and so on.

DFT Logic Intended for Self-Test Only

If you have testability logic in your design that should be enabled *only* during self-test operation (autonomous and ATPG modes), then enable it using the LBIST_EN signal instead of a test-mode signal.

During DFT insertion, any features that use LBIST_EN as the control signal are enabled from the self-test-mode output decoded by the TCM, as shown in [Figure 499](#). Any existing RTL logic connections to the LBIST_EN signal source are also remapped to this self-test-mode output.

Figure 499 Self-Test Assertion Logic for Self-Test-Only DFT Signals



Caution:

The self-test-mode output of the TCM is not synchronized, registered, or guaranteed glitch-free. Any logic connections made to it (including test points) must be handled accordingly.

DFT Logic Intended for Both Manufacturing Test and Self-Test

By default, global DFT logic enabled for manufacturing test modes is also enabled during self-test operation (autonomous and ATPG modes).

The tool automatically inserts self-test assertion logic for the following DFT signals:

- The self-test mode output of the test control module (TCM)
- Test-mode signals for OCC controllers

The tool **does not** insert self-test assertion logic for the following DFT signals:

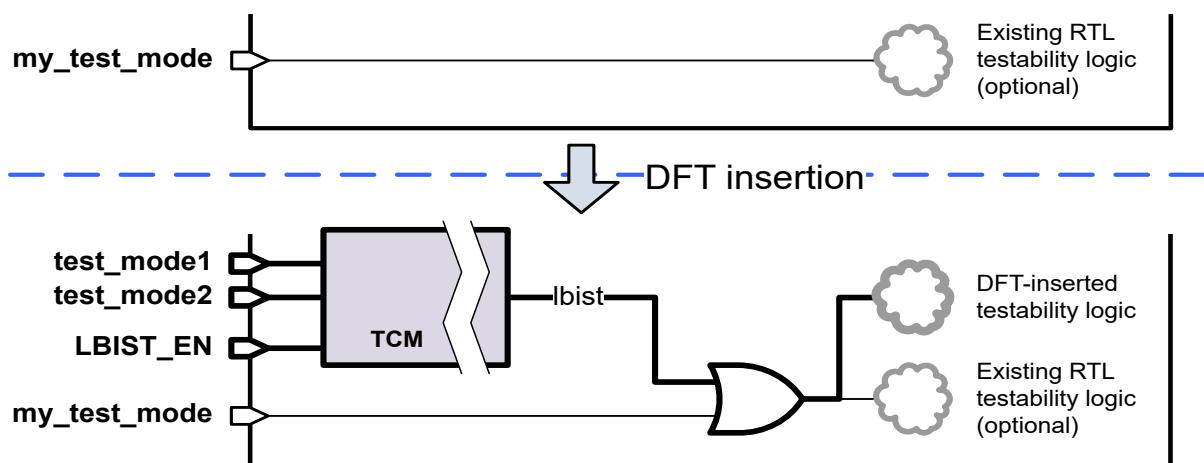
- Test-mode signals for test points
- Constant signals (`set_dft_signal -type Constant`)

To assert these signals during self-test, you must insert a `force_0` or `force_1` user-defined test point that uses `LBIST_EN` as the control signal:

```
dc_shell> set_test_point_element -control_signal LBIST_EN \
    -type force_1 my_test_mode
```

This results in the assertion logic shown in [Figure 500](#). The test point also asserts any existing RTL logic connections to the signal source.

Figure 500 Self-Test Assertion Logic for Unasserted DFT Signals



For ports that are not connected before executing `insert_dft`, when a `set_test_point_element` is placed, you get the following warning message:

Warning: Skipping test point at location tp_tm. Port is not connected.

As described in, [Using -type Constant versus Using -type TestMode on page 226](#), the Constant signal type is not guaranteed to describe the signal's expected (or required) state during mission mode. However, you can insert a test point to meet the design requirements, if needed.

Blocking Internal X Sources

LogicBIST designs must be X-clean. If a scan cell captures an X value, the signature value in the MISR is corrupted and the self-test results are useless (despite what coverage might be reported). If you have X sources inside your design, you must block them during self-test operation.

TestMAX ATPG provides features to identify X sources. For DRC, you can set the X2 violation severity to warning to identify scan cells that capture X values. During seed and signature computation, M740 violations report patterns in which the MISR captures X values.

Possible X sources include:

- Unwrapped or unisolated input ports

During self-test, input ports must block external X values from being captured. You can use core wrapping or isolation test points, as described in [Isolating the Design During LogicBIST Self-Test on page 1021](#).

Caution:

Avoid sharing functional input ports with DFT signals (such as scan-in or scan-enable signals) if possible. DFT signals are not wrapped during core wrapping, and inserting isolation test points that do not interfere with manufacturing test requires careful attention to detail.

- Nonscan cells

If you have scan cells that capture values from nonscan cells, you might be able to use the loadable nonscan cell feature in TestMAX ATPG to prevent the nonscan cells from driving X values into the scan cells. For more information, see “Using Loadable Nonscan Cells” in TestMAX ATPG and TestMAX Diagnosis Online Help.

- Black boxes

If you have unmodeled macro cells or memories without ATPG models, and any outputs of these blocks reach a scan cell, you must insert X-blocking logic at the component outputs by inserting test points or modifying the RTL.

- Timing exceptions

Scan cells that are endpoints of timing exceptions capture X values. You can use test points to prevent these X values from being captured.

Ensuring Testability for Reset Signals

Asynchronous reset (and set) signals are defined using the `set_dft_signal` command:

```
set_dft_signal -view existing_dft -type Reset \
    -port RSTN -active_state 0
```

These signals are similar to clocks in that they cause sequential cells to capture a value. As a result, you must ensure that there is no capture interaction between clock and reset signals during LogicBIST operation.

If you have OCC-controlled clocks, you *must* use weighted clock capture groups to allocate capture cycles between clocks and reset signals.

If all clocks are external (port-driven), you can choose either of the following:

- Insert OCC controllers for the external clocks and use weighted clock capture groups to allocate capture cycles between clocks and reset signals.

Because clocks and resets are separated into groups that avoid capture interactions, this method provides coverage of the reset network but might increase pattern count for a given coverage.

- Use the external clock controller and disable the reset signal during LogicBIST operation.

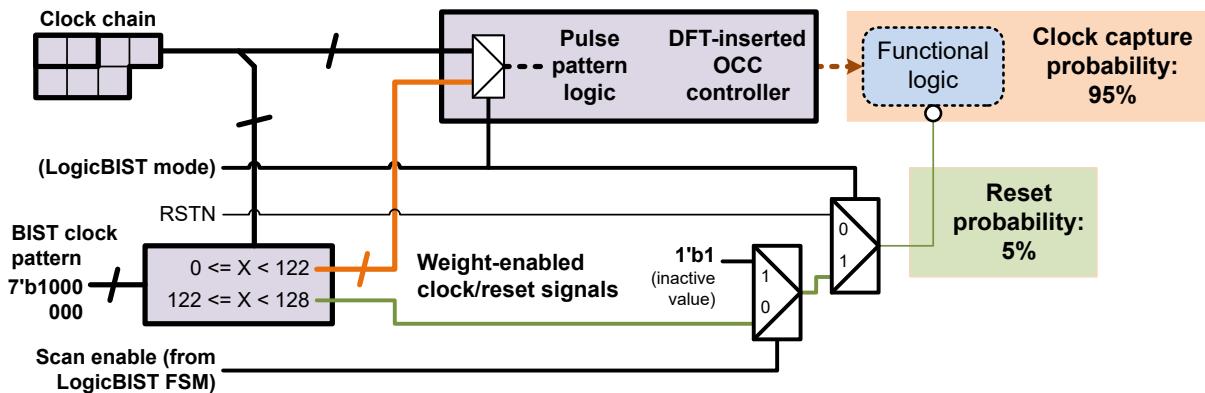
Because all clocks capture on every cycle, this method improves clock capture efficiency but does not provide coverage on the reset network.

Caution:

Do not define any asynchronous reset signal with this method. Instead, define the reset source as a constant signal. For TestMAX ATPG in manufacturing test modes, you must manually define the reset signal.

[Figure 501](#) shows the weighted clock/reset logic structure.

Figure 501 Using Weighted Capture Groups for Reset (or Set) Signals



When LogicBIST operation is active,

- The reset signal is asserted when LBIST_EN is asserted but START is de-asserted. (Because of the synchronizer delay on the START signal, this condition always occurs.)
- When START is asserted, the test begins. Within each self-test pattern,
 - The reset signal is de-asserted during scan shift.
 - The reset signal is controlled by the clock/reset weight decoder during scan capture.
- The reset signal is reasserted when the test completes.
- The reset signal is controlled by the functional logic when LBIST_EN is de-asserted.

For a design with multiple reset signals, you typically include all reset signals in a single reset group.

If your design uses external clocks, see [Specifying OCC Controllers for External Clock Sources on page 550](#).

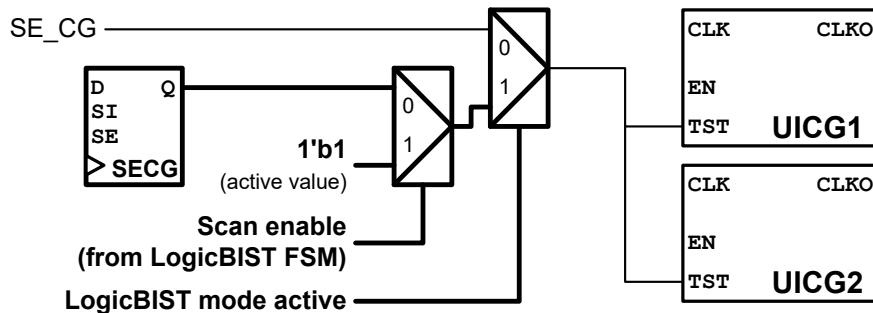
See Also

- [OCC-Controlled Clocks With Weighted Clock Capture Groups on page 1018](#) for details on how weighted capture groups work

Ensuring Testability for Integrated Clock-Gating Cells

If you have or are inserting integrated clock-gating (ICG) cells, LogicBIST requires a dedicated scan-enable signal for ICG test pins. The tool automatically inserts testability logic for ICG cells identified by DFT insertion. [Figure 502](#) shows the clock-gating cell testability logic structure.

Figure 502 LogicBIST Testability Logic for Clock-Gating Cells



When LogicBIST operation is active,

- The clock-gating scan-enable signal is asserted during scan shift.
- The clock-gating scan-enable signal is controlled by a dedicated DFT-inserted testability scan cell (SECG in Figure 502) during scan capture.

If you declare multiple clock-gating scan-enable signals, the tool inserts a separate control register for each signal declaration.

Caution:

Clock-gating cells require a dedicated scan-enable signal. For details, see [LogicBIST Limitations and Known Issues on page 1077](#).

If you have instantiated clock-gating cells in your RTL, they *must* be identified so they are controlled during self-test. See [Inserting LogicBIST in Designs With Clock-Gating Cells on page 1042](#) for details.

Discrete-logic clock-gating cells have no test pin, so no testability logic can be inserted for them.

31

Using LogicBIST Self-Test

To use LogicBIST self-test, you specify the desired number of compressed chains and scan patterns to run. The tool synthesizes the scan and BIST circuitry and writes the architectural information to the SPF file. The TestMAX ATPG tool then computes seed and signature values, which are used for both testbench simulation and autonomous device self-test operation.

This chapter includes the following topics:

- [Configuring LogicBIST Self-Test](#)
 - [Previewing and Inserting the LogicBIST Implementation](#)
 - [Computing the Seed and Signature Values in TestMAX ATPG](#)
 - [Setting the Seed and Signature Values in Synthesis](#)
 - [Simulating Autonomous BIST Operation](#)
 - [Integrating the Self-Test Logic into the Functional Design Logic](#)
 - [Example LogicBIST Scripts](#)
-

Configuring LogicBIST Self-Test

LogicBIST configuration is described in the following topics:

- [Defining the LogicBIST Control Signals](#)
- [Defining the LogicBIST Scan-In Signal](#)
- [Defining the LogicBIST Self-Test Mode](#)
- [Configuring the PRPG and MISR Lengths](#)
- [Configuring the Pattern Counter and Shift Counter Lengths](#)
- [Configuring the Self-Test Capture Clock Timing](#)
- [Configuring Clock and Reset Weights](#)
- [Controlling Self-Test Through IEEE 1500 Logic](#)

- [Configuring Self-Test Isolation Logic](#)
 - [Inserting LogicBIST in Designs With Trailing-Edge Flip-Flops](#)
 - [Inserting LogicBIST in Designs With External Chains](#)
 - [Inserting LogicBIST in Designs With Clock-Gating Cells](#)
-

Defining the LogicBIST Control Signals

You can define the LogicBIST-specific signals (LBIST_EN, START, STATUS_0, STATUS_1) on existing ports using the following signal types:

```
set_dft_signal -view spec -port my_enable -type lbistEnable
set_dft_signal -view spec -port my_start -type lbistStart
set_dft_signal -view spec -port my_status0 -type lbistStatus_0
set_dft_signal -view spec -port my_status1 -type lbistStatus_1
```

You can also define these signals on internal pins using the `-hookup_pin` option.

The LBIST_EN and START signals cannot be defined on the same source object within the design because ATPG mode requires that they be separately controllable at the design level. However, they can both be driven by the same signal external to the design.

If you do not define these signals, the tool automatically creates them using the following signal port names: LBIST_EN, START, STATUS_0, STATUS_1.

See Also

- [The LogicBIST Control and Data Signals on page 1012](#) for details on how the control and status signals work
-

Defining the LogicBIST Scan-In Signal

A LogicBIST implementation requires at least one user-defined scan-in signal. Define it on an existing input port:

```
set_dft_signal -view spec -port {SI1} -type ScanDataIn
```

This signal is used for ATPG mode, which is described in [The LogicBIST Operational Modes on page 1012](#). No user-defined scan-out signal is required.

Defining the LogicBIST Self-Test Mode

Synthesis commands and command options related to LogicBIST self-test contain the word “logicbist”. To enable LogicBIST self-test insertion, use the following command:

```
dc_shell> set_dft_configuration -logicbist enable
```

To insert LogicBIST self-test in your design, define a test mode with a usage of `logicbist`. Then, use the `set_logicbist_configuration` command to configure the self-test configuration parameters.

The following example script is for a design that uses core wrapping for boundary testability:

```
# define scan clocks
set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port CLK1

# enable self-test insertion
set_dft_configuration -logicbist enable

# define the uncompressed inward-facing mode and its
# corresponding inward-facing scan compression mode
define_test_mode WRP_IF -usage wrp_if
define_test_mode LBIST -usage logicbist

# configure uncompressed scan mode
set_scan_configuration -test_mode WRP_IF -chain_count 2

# configure LogicBIST self-test mode
set_logicbist_configuration \
    -base_mode WRP_IF -test_mode LBIST \
    -clock CLK1 \
    -chain_count 32
```

You must explicitly configure the LogicBIST codec using the `-chain_count` or `-max_length` option of the `set_logicbist_configuration` command; there is no default for these options.

If your design has multiple scan clocks, you can use the `-clock` option to specify which clock to use for LogicBIST operation. The specified clock must be previously defined as a scan clock using the `set_dft_signal` command. The default is the first-defined scan clock.

Note that the test-mode encoding of the self-test mode is not actually applied to the design during in-place self-test; it is applied only during seed and signature computation in the TestMAX ATPG tool. For details, see [The LogicBIST Operational Modes on page 1012](#).

Configuring the PRPG and MISR Lengths

By default, the tool chooses the PRPG and MISR register widths based on the scan architecture. To specify a particular width for the PRPG or MISR register, use the following options:

```
set_logicbist_configuration \
    -prpg_width width_value \
    -mistr_width width_value
```

The minimum width value is a function of the number of compressed chains in the self-test mode. The specified width must be large enough to satisfy the following requirement:

```
width_value*(width_value-1)/2 >= num_compressed_chains
```

Configuring the Pattern Counter and Shift Counter Lengths

By default, the tool creates a pattern counter register with a width of 8, which allows up to 256 patterns. To specify a particular width for the pattern counter register, use the following option:

```
set_logicbist_configuration \
    -pattern_counter_width width_value
```

By default, the tool sizes the shift counter register according to the longest shift chain in the design. In most cases, this is sufficient. To change the shift counter width, use the following option:

```
set_logicbist_configuration \
    -shift_counter_width width_value
```

Configuring the Self-Test Capture Clock Timing

During self-test, the scan-enable signal is generated by the self-test controller according to the timing programmed into its finite state machine (FSM) logic. To control the timing relationship between shift and capture operations, you must configure how this logic is implemented.

Capture Window Duration

To specify the duration of the capture window, set the following application variable:

```
set_app_var test_logicbist_capture_cycles num_cycles
```

where *num_cycles* is the duration in BIST clock cycles.

The default of this variable is 1, which is suitable for single-pulse designs without OCC controllers.

For designs with OCC controllers, you *must* set this variable to a sufficiently large value to allow all at-speed capture clock pulse sequences to be issued, then to allow all registers to settle and be ready for scan shifting.

Use the following formula to compute the recommended value for your design:

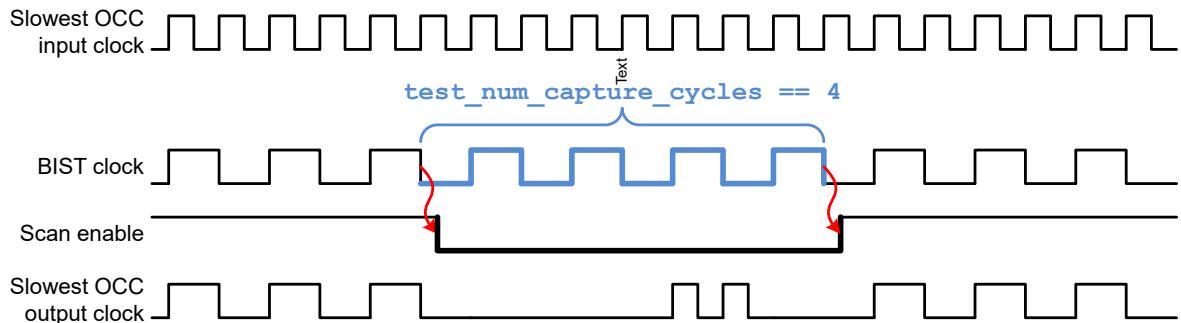
```
test_logicbist_capture_cycles =
    ( (P_ate_clk * 2 )
    + (P_fast_clk * ( 5 + num_capture_cycles ) ) ) / P_ate_clk
```

where

```
P_ate_clk = period of ATE clock
P_fast_clk = period of slowest OCC-controlled clock
num_capture_cycles = cycle count C specified by
    set_clock_controller_configuration -cycles C
```

[Figure 503](#) shows an example where `test_logicbist_capture_cycles` is set to 4.

Figure 503 Capture Cycle Timing Example



Adding Capture Window Margin for Scan Enable

If your scan-enable signal has a higher insertion delay than one or more of your scan clocks, then you might need to add margin to the deassertion of the scan-enable signal.

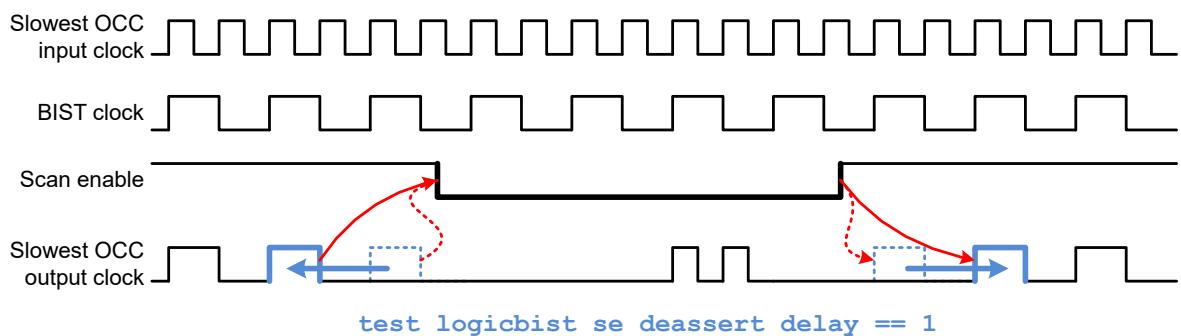
To do this, set the following application variable to the number of BIST clock cycles of margin to add:

```
set_app_var test_logicbist_se_deassert_delay num_cycles
```

The margin is added before scan-enable deassertion *and* after scan-enable assertion.

[Figure 504](#) shows an example where `test_logicbist_se_deassert_delay` is set to 1.

Figure 504 Capture Window Scan-Enable Signal Margin Example



Configuring Clock and Reset Weights

If your design requires weighted clock capture groups, use the `-occ_clock_weights` option to define them:

```
set_logicbist_configuration ... \
    -occ_clock_weights {{weight1 clock1 [clock2 ...]} \
        {weight2 clock3 [clock4 ...]} \
    ...}
```

The `-occ_clock_weights` option takes a list of group definitions, where each group is a list containing a weight value followed by the OCC-controlled clock signals in that group.

If you also have asynchronous reset or set signals, specify their weights with the `-reset_weights` option using the same group syntax. Typically there is a single reset group.

For example,

```
set_logicbist_configuration \
    -occ_clock_weights {{80 UPLL/CLKO CLK33} {44 CLK266}} \
    -reset_weights {{4 RSTN1 RSTN2}}
```

Caution:

The signals in a `-reset_weights` specification *must* be defined as resets using the `set_dft_signal -type Reset` command.

The weight values must be positive integers. There is no fixed scale for the weights; each group's capture probability is relative to the sum of all clock and reset group weights. However, the weight comparator logic uses a 7-bit PRPG word. Thus, the most accurate total weight values are powers of 2, from 2 to 128. For other total weight values, the weights are remapped to comparator values between 0 and 127, which might perturb the probabilities due to rounding errors.

You can also specify two special weight values, `always_on` and `always_off`, for clocks that should always pulse or never pulse, respectively. These values do not affect the sum of all numeric clock and reset group weights.

If you use reset weights, you must run the `update_fault -reset_au` command in the TestMAX ATPG tool before performing seed and signature analysis.

To determine how noninteracting clocks can be grouped, you can perform clock grouping analysis in the TestMAX ATPG tool. For details, see "Clock Grouping" in the TestMAX ATPG and TestMAX Diagnosis Online Help.

Configuring Self-Test Isolation Logic

The following topics describe how to configure logic that isolates your design from surrounding logic during self-test operation:

- [Configuring Wrapper Chain Isolation Logic](#)
- [Configuring Test Point Isolation Logic](#)

Configuring Wrapper Chain Isolation Logic

Wrapper chains inherently provide the isolation needed by self-test operation. If your design already implements wrapper chains, no further action is needed. Otherwise, you can enable and configure wrapper chains.

To use wrapper chains for self-test isolation, do the following:

1. Enable both the core wrapper and LogicBIST clients:

```
set_dft_configuration -wrapper enable -logicbist enable
```

2. Define global DFT signals, wrapper configuration settings, and LogicBIST configuration settings.

Any logic between the I/O ports and wrapper chain cannot be tested by LogicBIST self-test. To minimize such logic, either use the simple wrapper flow, or use the maximized-reuse flow and specify a low value for the `-depth_threshold` option of the `set_wrapper_configuration` command.

3. If the self-test design drives top-level logic that cannot tolerate pseudorandom output data during self-test, specify safe values for the output wrapper cells:

```
# global:
set_wrapper_configuration -class core_wrapper \
    -safe_state 0 ;# or 1

# per-port:
set_boundary_cell -class core_wrapper \
    -ports port_list -safe_state 0 ;# or 1
```

4. Define the uncompressed wrapper modes, scan compression modes, and LogicBIST self-test modes:

```
define_test_mode WRP_IF -usage wrp_if
define_test_mode WRP_OF -usage wrp_of      ;# if needed
define_test_mode DFTMAX -usage scan_compression
define_test_mode BIST -usage logicbist
```

5. When configuring the LogicBIST test mode, reference the inward-facing uncompressed test mode as its base mode:

```
set_logicbist_configuration -test_mode BIST -base_mode WRP_IF ...
```

The LogicBIST test mode becomes an inward-facing mode that drives LogicBIST-generated data into the input wrapper chain, and incorporates the captured output wrapper chain data into the MISR. As with other DFT signals, LogicBIST-specific signals (LBIST_EN, START, STATUS_0, and STATUS_1) are not wrapped.

When the core wrapper and LogicBIST clients are both enabled, the following core wrapping behaviors change:

- The tool no longer creates the outward-facing `wrp_of` mode by default; you must explicitly define it with the `define_test_mode -usage wrp_of` command. This allows core wrapping to be used only for self-test isolation and not full hierarchical test.
- In the maximized reuse core wrapping flow, dedicated wrapper cells are added for
 - Ports associated with feedthrough paths.
 - Ports that drive black-box cells that have no CTL model.

You can suppress wrapper cells on these ports by assigning them a wrapper cell type of `none`:

```
set_boundary_cell -class core_wrapper -type none -ports {port_list}
```

Note the following requirements and limitations when using core wrapping with LogicBIST self-test:

- If you have functional ports reused as scan ports, you must isolate them with user-defined test points. The tool does not wrap these ports.
 For simplicity, it is best to avoid this where possible.
- If your design has feedthrough paths, restrictions apply to the use of shared wrapper cells on them. For details, see [SolvNet article 2506549, “Feedthrough Path Caveats in Maximized-Reuse Wrapped LogicBIST Designs.”](#)
- The tool-created wrapper clock (`wrp_clk`) is not controlled by the BIST clock controller (OCC or non-OCC) and cannot be used. To avoid this, ensure that it is not present in the wrapper preview report.

See Also

- [Isolating the Self-Test Design Using Core Wrapping on page 1021](#) for details on how core wrapping provides design isolation
- [Example Core Insertion Script Using Core Wrapping on page 1057](#) for an example script

Configuring Test Point Isolation Logic

If your design is very area-sensitive and is not already core-wrapped, you can isolate the design using test points instead of a wrapper chain.

Note:

If the I/O ports in your design are mostly registered, a maximized-reuse wrapper chain might require less area than adding test points.

To use test points for isolation, enable and configure the `core_wrapper` target of automatic test point insertion. For example,

```
# enable automatic test point insertion
set_dft_configuration -testability enable

# enable and configure the core_wrapper target
set_testability_configuration \
    -target core_wrapper \
    -control_signal LBIST_EN
```

In this example, because the `core_wrapper` target uses `LBIST_EN` as the control signal, the test points isolate the logic during self-test but not manufacturing test.

The `core_wrapper` target provides additional configuration features and options. For details, see [Configuring the Core Wrapper Test Point Target on page 316](#).

See Also

- [Isolating the Self-Test Design Using Test Points on page 1022](#) for details on how test points provide lightweight design isolation
- [Connecting the Self-Test Signals to the Functional Design Logic on page 1051](#) for details on how the tool uses `LBIST_EN` as a control signal
- [Example Core Insertion Script Using Test-Point Isolation on page 1058](#) for an example script

Controlling Self-Test Through IEEE 1500 Logic

If you insert IEEE 1500 test-mode control and LogicBIST self-test together in the same design, the tool implements logic to control self-test entirely through the IEEE 1500 interface.

In this architecture,

- The pattern count, shift count, seed, and signature values are driven by the TMCDR.
 In designs with OCC controllers, the OCC capture mask is driven by the TMCDR.
- The control signals (START and LBIST_EN) are driven by the TMCDR.
- The status signals are captured for observation by the TMCDR.

Configuration

To properly allocate the TMCDR bits, all four self-test value widths must be explicitly specified. For example,

```
# Enable both IEEE 1500 and LogicBIST insertion
set_dft_configuration -ieee_1500 enable -logicbist enable

# Specify self-test value widths so that CDR bits are allocated
set_logicbist_configuration ... \
    -pattern_counter_width 12 \
    -shift_counter_width 6 \
    -prpg_width 18 \
    -misr_width 18
```

Any signals definitions of the following types are ignored:

```
lbistEnable
lbistStart
lbistStatus_0
lbistStatus_1
```

Testbench Generation

After DFT insertion, the `write_test` command writes a self-test testbench that runs self-test and checks the status entirely through the IEEE 1500 interface. To use it, you must specify the values and the TMCDR register name to use for the testbench as follows:

```
write_test -format stil -output bist_tb \
    -seed 110100100101010011 \
    -signature 011000011110100110 \
    -shift_counter 50 \
    -pattern_counter 250 \
    -capture_cycle {10} \
    -cdr_name CDR
```

The `-cdr_name` option specifies the name of the CDR register segment. By default, the tool names it "CDR". If you configure a TMCDR register with the `set_scan_path` command (see [Customizing the IEEE 1500 Architecture on page 384](#)), then specify that segment name instead.

If you have a serial STIL file containing the seed, signature, and pattern count values, you can use the `set_logicbist_constants` Tcl procedure (instead of the `write_test` command) to generate the testbench. For details, see [SolvNet article 2231010, "Setting the Seed and Signature Values in a LogicBIST Design."](#)

See Also

- [Test-Mode Control Using the IEEE 1500 and IEEE 1149.1 Interfaces on page 377](#) for details on implementing IEEE 1500 logic in your design

Inserting LogicBIST in Designs With Trailing-Edge Flip-Flops

The PRPG register in the LogicBIST decompressor is clocked on the leading edge. If your design has trailing-edge flip-flops, the tool inserts retiming registers as needed.

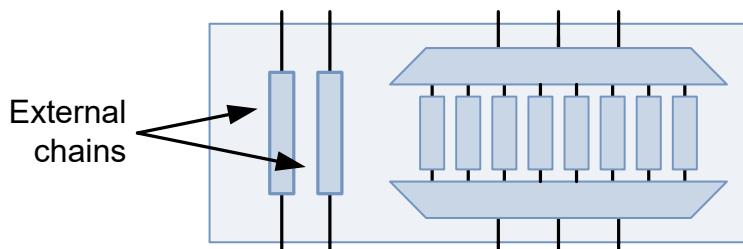
The beginning retiming registers inserted between the LogicBIST decompressor and trailing-edge head scan cells have a state-holding loop that is used during scan capture.

If you enable ending retiming flip-flops, the ending retiming registers inserted between trailing-edge tail scan cells and the LogicBIST compressor are regular non-state-holding retiming registers. They are inserted even though the path to the leading-edge MISR register would make timing without them, so a value of `begin_only` is preferred for minimal area.

Inserting LogicBIST in Designs With External Chains

External chains are scan chains that are excluded from scan compression. They are defined using the `-scan_data_in` and `-scan_data_out` options of the `set_scan_path` command to directly connect the scan chain to scan-in and scan-out ports, as shown in the following figure.

Figure 505 External Chains in a Compressed Scan Design



External chains are supported in DFTMAX and DFTMAX Ultra test modes. However, external chains are not supported in LogicBIST test modes because the inputs are unknown and the outputs are unobserved. This causes all scan elements in the external chain to become X sources.

To avoid this, use the `-test_mode` option of the `set_scan_path` command to define the external chain for the required test modes, but not the LogicBIST self-test mode.

For example, to define an external clock chain for all test modes except the self-test mode,

```
# define test modes
define_test_mode wrp_if -usage wrp_if
define_test_mode wrp_of -usage wrp_of
define_test_mode bist -usage logicbist
...
# must define external chain AFTER all test modes are defined,
# so that all_test_modes returns the set of defined modes
foreach tm [lminus [all_test_modes] bist] {
    set_scan_path EXT_OCC_${tm} -class occ \
        -scan_data_in OCC_SI \
        -scan_data_out OCC_SO \
        -test_mode $tm
}
```

Inserting LogicBIST in Designs With Clock-Gating Cells

If you have or are inserting integrated clock-gating (ICG) cells, LogicBIST requires a dedicated scan-enable signal for ICG test pins, as described in [Ensuring Testability for Integrated Clock-Gating Cells on page 1029](#).

To do this, define a separate signal with a usage of `clock_gating`:

```
set_dft_signal -view spec -type ScanEnable -port SE
set_dft_signal -view spec -type ScanEnable -port SE(CG) \
    -usage {clock_gating}
```

The dedicated signal allows the tool to use a special self-test control register to control ICG test pins during self-test. If you declare multiple clock-gating scan-enable signals, the tool inserts a separate control register for each signal declaration.

You cannot use a single scan-enable signal with combined usages of `{scan clock_gating}`, as then the signals are not independently controllable for each purpose. Test-mode clock-gating control signals are not supported.

Existing ICG Cells in the RTL

If you have instantiated ICG cells in your RTL, they *must* be identified so they are controlled during self-test. You can use any of the methods described in [Instantiating Clock-Gating Cells in the RTL on page 272](#) to identify the test pins.

If a test pin has an existing, non-constant connection in the RTL, then the tool-inserted logic controls it only during self-test; the existing connections remains logically in place in all other cases—including manufacturing test.

See Also

- [SolvNet article 3055862, “Defining Two LogicBIST Scan-Enable Signals at a Single Port”](#) for a workaround to the single-port limitation

Previewing and Inserting the LogicBIST Implementation

The following topics describe how to preview and insert LogicBIST self-test in your design, then write out the files needed for seed and signature generation:

- [Previewing the LogicBIST Implementation](#)
- [Inserting the LogicBIST Logic](#)
- [Writing Out the LogicBIST Design Files](#)

Previewing the LogicBIST Implementation

After you have configured your LogicBIST implementation, use the `preview_dft` command to preview the implementation. The preview report contains a LogicBIST section that describes the self-test values.

Constant-Driven Values

For a design with constant-driven values, the report shows the initial values:

```
*****
LogicBIST Compression information
*****  
  

PRPG size: 31
MISR size: 30
Shift counter size: 4
Pattern counter size: 12  
  

Shift counter data:
    top_U_LogicBISTController_bist/shift_count_data = 4'b1001  
  

Pattern counter data:
    top_U_LogicBISTController_bist/user_pattern_count_data =
        12'b000000000000
PRPG seed data:
    top_U_decompressor_bist/user_prpg_seed =
        31'b00000000000000000000000000000000
MISR signature data:
    top_U_compressor_bist/user_misr_signature =
        30'b00000000000000000000000000000000
```

The pattern counter, PRPG seed, and MISR signature values are set to all-zeros in the initial implementation; these values are determined later in TestMAX ATPG. The shift counter is automatically set according to the longest scan chain length in LogicBIST mode.

The test-mode section indicates the mission mode encoding that must be asserted for autonomous self-test. For example,

```
Test Mode Controller Index (MSB --> LSB)
-----
TM1, TM0

Control signal value - Test Mode
-----
00 Mission_mode - Normal

10 bist - InternalTest

01 wrp_if - InternalTest

11 dftmax - InternalTest
```

Information: For self-test, test mode 'Mission_mode' (opcode '00') is used for autonomous operation, while 'bist' (opcode '10') is used for TetraMAX ATPG DRC. (TEST-2096)

Designs With IEEE 1500

For a design with DFT-inserted IEEE 1500 logic, the report shows how the CDR bits are allocated to the self-test values and test-mode signals using a bit-by-bit format:

```
*****
LogicBIST Compression information
*****  
  

PRPG size: 18
MISR size: 18
Shift counter size: 6
Pattern counter size: 12  
  

SHIFT COUNTER CONNECTIONS:
*****
    top_Test_Controller_1500_inst/CDR[9] connected to
        top_U_LogicBISTController_bist/shift_count_data[0]
    top_Test_Controller_1500_inst/CDR[8] connected to
        top_U_LogicBISTController_bist/shift_count_data[1]
    ..
    top_Test_Controller_1500_inst/CDR[4] connected to
        top_U_LogicBISTController_bist/shift_count_data[5]  
  

PATTERN COUNTER CONNECTIONS:
*****
    top_Test_Controller_1500_inst/CDR[21] connected to
```

```

        top_U_LogicBISTController_bist/user_pattern_count_data[0]
...
        top_Test_Controller_1500_inst/CDR[10] connected to
        top_U_LogicBISTController_bist/user_pattern_count_data[11]

SEED CONNECTIONS:
*****
        top_Test_Controller_1500_inst/CDR[57] connected to
        top_U_decompressor_bist/user_prg_seed[0]
...
        top_Test_Controller_1500_inst/CDR[40] connected to
        top_U_decompressor_bist/user_prg_seed[17]

EXPECTED SIGNATURE CONNECTIONS:
*****
        top_Test_Controller_1500_inst/CDR[39] connected to
        top_U_compressor_bist/user_misr_signature[0]
...
        top_Test_Controller_1500_inst/CDR[22] connected to
        top_U_compressor_bist/user_misr_signature[17]

```

The test-mode section indicates the mission mode encoding that must be asserted by the TMCDR for autonomous self-test. For example,

```

=====
Test Mode Controller Information
=====

Test Mode Controller Ports
-----
test_mode: top_Test_Controller_1500_inst/CDR[0]
test_mode: top_Test_Controller_1500_inst/CDR[1]

Test Mode Controller Index (WSO --> WSI)
-----
top_Test_Controller_1500_inst/CDR[0],
top_Test_Controller_1500_inst/CDR[1]

Control signal value - Test Mode
-----
00 Mission_mode - Normal

01 bist - InternalTest

10 wrp_if - InternalTest

11 dftmax - InternalTest

Information: For self-test, test mode 'Mission_mode' (opcode '00') is
used for autonomous operation, while 'bist' (opcode '01') is used for
TetraMAX ATPG DRC. (TEST-2096)

```

The LBIST_EN and START signals are also driven by the TMCDR, but they are not shown in the report.

Clock Information

The self-test section of the preview report lists the number of scan cells clocked by each clock:

```
Clock to scan registers distribution
Clock source UPLL1/CLKO: drives 64 scan registers (33%)
Clock source UPLL2/CLKO: drives 128 scan registers (66%)
```

If you are using clock weights, the report also lists the implemented weight percentages:

```
OCC clock weights information
*****
Group 1: Normalized Weight = 37 % : UPLL1/CLKO
Group 2: Normalized Weight = 62 % : UPLL2/CLKO
*****
```

Inserting the LogicBIST Logic

When you are satisfied with your DFT configuration, run the `insert_dft` command. Do not run an explicit incremental compile yet.

Writing Out the LogicBIST Design Files

After DFT insertion, write out the design netlist, SPF, and testbench file using the following commands:

```
# write out design netlist
write -format verilog -output top_no_seed_signature.vg -hierarchy

# write protocol for TestMAX ATPG to calculate seed/signature
write_test_protocol -test_mode LBIST -output LBIST.spf

# write testbench for Verilog simulation to validate
# LogicBIST implementation
#
# (this command produces bist_tb.stil)
write_test -format stil -output bist_tb
```

The files created by these commands serve the following purposes:

- The `write_test_protocol` command creates an SPF used by TestMAX ATPG to compute the seed and signature values in ATPG mode. This SPF does not simulate the actual autonomous LogicBIST test process; instead, it enables ATPG mode so that TestMAX ATPG can access the LogicBIST configuration registers through scan to evaluate seed values. OCC controllers are bypassed in the SPF.

- The `write_test` command creates a STIL pattern file that can be used to simulate the LogicBIST logic in autonomous mode (with the test-mode signals set to mission mode encoding). OCC controllers are enabled in the testbench.

Autonomous BIST operation cannot be simulated until you set the seed, signature, and pattern count values, as described in [Setting the Seed and Signature Values in Synthesis on page 1049](#).

For designs with IEEE 1500, you can use the `set_logicbist_constants` Tcl procedure (instead of the `write_test` command) to write out the testbench.

The test protocol and STIL pattern files drive X values at the design inputs, which validates that self-test is unaffected by external values.

Post-DFT DRC (running the `dft_drc` command after DFT insertion) is supported for LogicBIST designs. DRC checking is also performed in the TestMAX ATPG tool during seed and signature calculation.

If desired, you can leave your synthesis session up while you generate seed and signature values in TestMAX ATPG. You can then return to the synthesis session to set the seed and signature values.

Computing the Seed and Signature Values in TestMAX ATPG

To calculate the seed and signature value in TestMAX ATPG, use the initial netlist and LogicBIST-mode SPF. For example,

```

read_netlist -library /project/libs/my_class.v
read_netlist top_no_seed_signature.vg
run_build top

# Enable LogicBIST DRC
set_drc -seq_comp_jtag_lbist_mode light_lbist
set_drc -allow_unstable_set_reset ;# only needed if reset signal exists
run_drc LBIST.spf

# Specify a particular seed value (optional)
#add_lbist_seeds 00000000000000000000000000000001

# Run LogicBIST ATPG for 133 patterns and 1 capture clock cycle
run_atpg -auto -jtag_lbist {1 133 1}
run_simulation
report_patterns -all

# write serial STIL file containing seed and signature values
write_patterns serial.stil -format stil -replace -unified -serial

```

If your design contains asynchronous set or reset signals, you must apply the `set_drc -allow_unstable_set_reset` setting to enable DRC to understand the reset-control logic.

In addition, if you use reset weights, you must run the `update_fault -reset_all` command in the TestMAX ATPG tool before performing seed and signature analysis.

The resulting STIL pattern file contains the seed and signature values generated by the `run_atpg` command.

Caution:

You cannot use this STIL pattern file to simulate autonomous self-test, as it contains only the computation of the signature value from the seed value. Instead, use the STIL pattern file written out by the DFTMAX `write_test` command. For details, see [Figure 480 on page 1005](#).

Choosing a Seed Value

By default, the `run_atpg` command uses seed values from a pseudorandom sequence of seed values. To evaluate a new seed value, run the `run_atpg` command again. Identical TestMAX ATPG sessions yield the same sequence of seed values.

You can also explicitly specify seed values using the `add_lbist_seeds` command. For details, see TestMAX ATPG and TestMAX Diagnosis Online Help.

Some seed values provide better coverage than others. To find an optimal seed value, use the `find_seed` Tcl procedure provided in [SolvNet article 2220819, “Finding Optimal Seed Values for the LogicBIST PRPG.”](#)

Computing the Signature Value

The values provided to the `-jtag_lbist` option are: number of seed values (always 1), pattern count, number of capture cycles (usually 1). The maximum pattern count value is $(2^{\text{pattern_count_width}})-2$, which allows for an additional load-only pattern at the beginning of self-test.

After ATPG completes, the serial STIL pattern file contains the seed, signature, pattern counter, and shift counter values for the test, provided in STIL annotation comments. The `report_patterns` command also reports this information.

If the `run_atpg` command issues M740 (MISR X capture) violations, the resulting signature value is invalid. For details on resolving these violations, see [SolvNet article 2460245, “How Do I Debug M740 Violations \(MISR X Capture\) in TetraMAX?”](#)

Note that the `write_testbench` command in the TestMAX ATPG tool writes a testbench that uses ATPG mode to externally access seed and signature values. It does not write a testbench that tests autonomous LogicBIST operation, as the `write_test` command does in the DFT environment.

Setting the Seed and Signature Values in Synthesis

After you write the serial STIL pattern file from TestMAX ATPG, you can use the file to set the seed, signature, and pattern counter values in your design. You can do this in the same DFT session where you wrote out the files for TestMAX ATPG. If that session is no longer running, reload the design from a .ddc file.

To set the values in the design to the values contained in the serial STIL file, use the `set_logicbist_constants` Tcl procedure from [SolvNet article 2231010, “Setting the Seed and Signature Values in a LogicBIST Design”](#). Its behavior depends on the type of seed, signature, and pattern count values used in your design:

- For constant-driven values, it modifies the netlist in memory to the desired values.
- For port-driven values, it creates a VCS command value to force the desired values at the block ports.
- For IEEE 1500 designs, it writes a testbench that performs self-test through the IEEE 1500 interface using the desired values.

Note:

There is no automation for programmable seed, signature, or pattern count values driven by functional logic. In this case, you must create your own initialization protocol and testbench for simulation.

The following example shows constant-driven values being set in a netlist:

```
dc_shell> source set_logicbist_constants.tcl
1
dc_shell> set_logicbist_constants -file_name serial.stil
Found LogicBIST controller for 'LBIST' test mode.
Obtained data from 'serial.stil' file.

Verifying shift counter value is set to '1011'...
  Verified top_U_LogicBISTController_LBIST/shift_count_data[3] is set to logic
  1
  Verified top_U_LogicBISTController_LBIST/shift_count_data[2] is set to logic
  0
  Verified top_U_LogicBISTController_LBIST/shift_count_data[1] is set to logic
  1
  Verified top_U_LogicBISTController_LBIST/shift_count_data[0] is set to logic
  1

Setting pattern counter value to '000001100101'...
Setting PRPG seed value to '1101011000000100111100100000110'...
Setting MISR signature value to '100011110010000110000111111011'...
```

Once you have set the desired values, the serial STIL pattern file is no longer needed. The LogicBIST simulation is performed using only the testbench created by the `write_test` command in the tool.

For constant-driven values, you should take care when optimizing the DFT logic early in the flow. For details, see [Post-DFT Design Optimization on page 1071](#).

Simulating Autonomous BIST Operation

To simulate autonomous LogicBIST operation in VCS, create a testbench from the STIL file generated by the `write_test` command in synthesis. To do this, use the `stil2verilog` command:

```
% stil2verilog bist_tb.stil bist_tb
```

This generates a Verilog testbench file (`bist_tb.v`) and associated data file (`bist_tb.dat`). You can then simulate autonomous operation using this testbench along with the final netlist.

An example VCS command line is as follows:

```
vcs \
    -notice -Mupdate -timescale=1ns/10ps \
    +nospecify +notimingcheck +tetramax +delay_mode_zero \
    -l vcs_lbist.log \
    -v libs/class.v \
    bist_tb.v \
    top.vg

./simv -l vcs_sim_lbist.log
```

For other simulators, see their documentation.

The resulting simulation should complete with no errors:

```
% ./simv -l vcs_sim_lbist.log
Notice: timing checks disabled with +notimingcheck at compile-time
Chronologic VCS simulator copyright 1991-2014
Contains Synopsys proprietary information.
Compiler version J-2014.12-SP2; Runtime version J-2014.12-SP2; Jun 23 15:34 2
015
#####
MAX TB Version K-2015.06
Test Protocol File generated from original file "bist_tb.stil"
STIL file version: 1.0
#####

XTB: Starting serial simulation of 0 pattern
XTB: Simulation of 0 patterns completed with 0 mismatches (time: 6144800.00 ns
' cycles: 61448)

      V C S      S i m u l a t i o n      R e p o r t

Time: 6144800 ns
```

If the design logic is modified, the simulation should complete with unexpected values on the STATUS_0 output:

```
% ./simv -l vcs_sim_lbist.log
Notice: timing checks disabled with +notimingcheck at compile-time
Chronologic VCS simulator copyright 1991-2014
Contains Synopsys proprietary information.
Compiler version J-2014.12-SP2; Runtime version J-2014.12-SP2; Jun 23 15:34 2
015
#####
MAX TB Version K-2015.06
Test Protocol File generated from original file "bist_tb.stil"
STIL file version: 1.0
#####
XTB: Starting serial simulation of 0 pattern
>>> Error during VectorStmt pattern 0
>>>     At T=6144540.00 ns, V=61446, exp=0, got=1, signal STATUS_0
>>> Error during VectorStmt pattern 0
>>>     At T=6144640.00 ns, V=61447, exp=0, got=1, signal STATUS_0
XTB: Simulation of 0 patterns completed with 2 mismatches (time: 6144800.00 ns
' cycles: 61448)

V C S   S i m u l a t i o n   R e p o r t

Time: 6144800 ns
```

For more information on the `stil2verilog` command, see “Using MAX Testbench” in TestMAX ATPG and TestMAX Diagnosis Online Help.

Integrating the Self-Test Logic into the Functional Design Logic

The following topics describe how to integrate the LogicBIST self-test logic into your functional design:

- [Connecting the Self-Test Signals to the Functional Design Logic](#)
- [Ensuring the Required Test Mode for Autonomous Self-Test](#)
- [Monitoring the Self-Test Status Signals](#)

Connecting the Self-Test Signals to the Functional Design Logic

To connect LogicBIST self-test logic to your functional design logic, you must connect the signal pins shown in [Table 59](#) to your preexisting design logic.

Table 59 Required LogicBIST Self-Test Signals

Signal type	Direction	Description
lbistEnable	Input	Enables autonomous self-test during mission mode
lbistStart	Input	Begins autonomous self-test
lbistStatus_0	Output	Reports current self-test status (idle, running, pass, fail)
lbistStatus_1		

Caution:

The `lbistEnable` signal must be de-asserted during manufacturing test as well as mission mode.

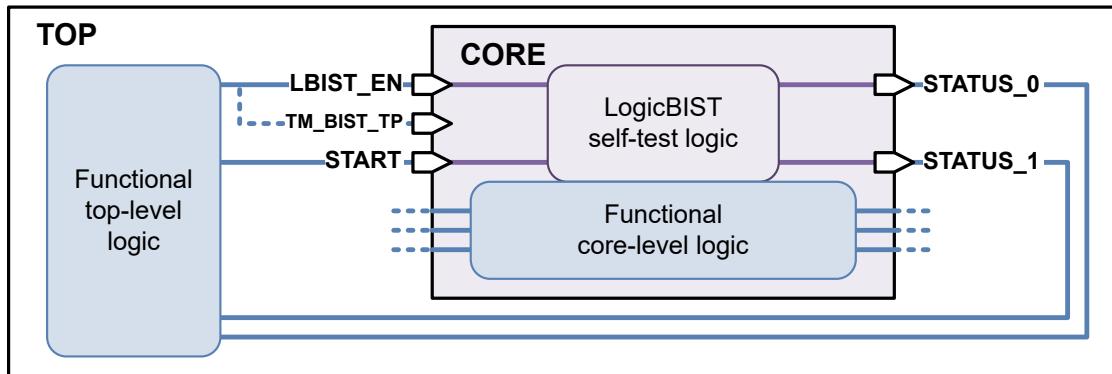
In addition, you can connect the optional signal pins shown in [Table 60](#) to your preexisting design logic.

Table 60 Optional LogicBIST Self-Test Signals

Signal type	Direction	Description
lbistPatternCount	Input	Specifies the number of self-test patterns to run
lbistShiftLength	Input	Specifies the number of shift cycles in each pattern
lbistSeedValue	Input	Specifies the initial seed value loaded into the PRPG
lbistSignatureValue	Input	Specifies the expected final signature value of the MISR
lbistBurnInEnable	Input	Enables burn-in mode using self-test logic
lbistBurnInStopOnFail	Input	Specifies whether burn-in mode should stop or continue upon failure

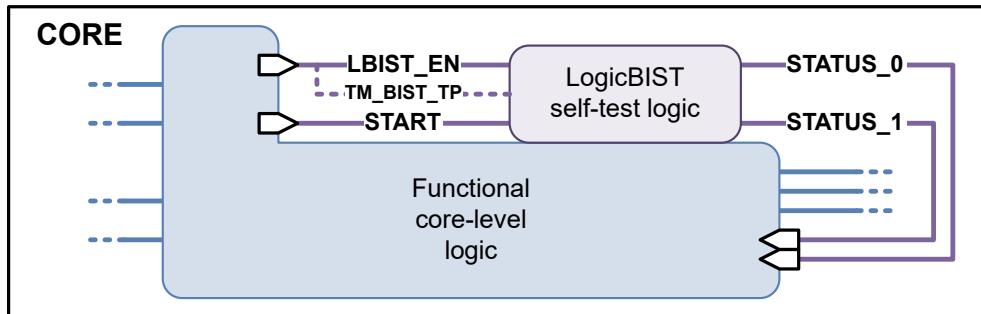
For LogicBIST signals connected by DFT insertion to input and output ports of the core module, the connections should preexist (or be made manually) at the next hierarchical level where the core is integrated, as shown in [Figure 506](#).

Figure 506 Connecting LogicBIST Self-Test Port Signals at the Top Level



For LogicBIST signals that connect to hookup pins inside the core module, the connections are made inside the core during DFT insertion, as shown in [Figure 507](#). The hookup pins can be pins of leaf cells or hierarchical cells.

Figure 507 Connecting LogicBIST Self-Test Hookup Pin Signals Inside the Core



You can use a mix of port-connected and hookup-pin-connected signals as needed.

If you have test points used only in self-test mode (and not in manufacturing test modes), connect their test-mode control signal to the `IbistEnable` signal. This connection should preexist (or be manually made) prior to DFT insertion, as DFT insertion does not modify existing test-mode signal connections.

You do not need to connect any scan-in, scan-out, or test-mode signals to your functional design logic. LogicBIST self-test automatically enables any required DFT logic during autonomous mode, as described in [Enabling DFT Logic During Autonomous Self-Test on page 1024](#).

See Also

- [Isolating the Self-Test Design Using Test Points on page 1022](#) for details on how test points provide design isolation

Ensuring the Required Test Mode for Autonomous Self-Test

Autonomous self-test is intended for operation in the device's final functional environment—the die has been packaged, passed manufacturing test, is installed on a circuit board, and is powered up in its operating environment and conditions.

The self-test logic is designed to operate in this functional environment. Correspondingly, the LBIST_EN signal enables autonomous self-test only when the test-mode signals of the device are asserted to their mission-mode encoding.

The `preview_dft` and `insert_dft` commands report this required encoding as follows:

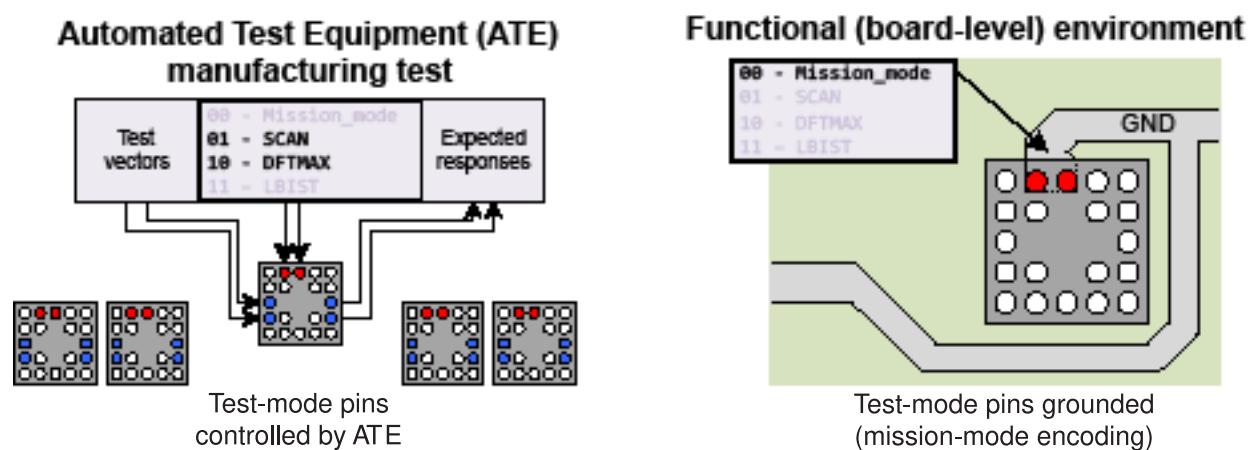
Information: For self-test, `test mode 'Mission_mode' (opcode '00') is used for autonomous operation`, while '`bist`' (opcode '`10`') is used for TetraMAX ATPG DRC. (TEST-2096)

By default, the tool selects this required test-mode encoding as follows:

- In core-wrapped designs, the tool-created Mission_mode mode
- In non-core-wrapped designs, in order of highest precedence first:
 - Unused all-zeroes encoding (created as Mission_mode)
 - A random unused nonzero encoding (created as Mission_mode)
 - A random standard scan test mode

Figure 508 shows a simple DFT design in its manufacturing test and functional operating environments. In the functional (board-level) environment, the test-mode package pins are tied to the ground plane. Thus, LBIST_EN enables autonomous self-test when asserted.

Figure 508 Test-Mode Pin Connections During Manufacturing Test and Functional Operation



Caution:

Be sure that the manufactured device sets the test-mode signals to the value required by the self-test logic. The autonomous self-test testbench written by the TestMAX DFT tool uses the required values, assuming they will be driven accordingly in the functional operating environment.

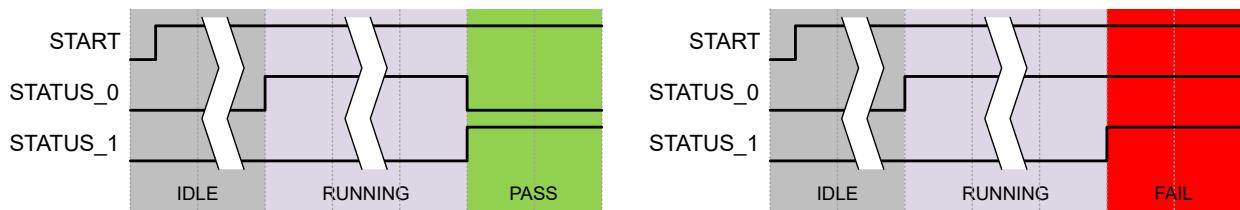
See Also

- [Changing the Test Mode Used for Autonomous Self-Test on page 1071](#) for details on specifying a nondefault test mode for autonomous self-test

Monitoring the Self-Test Status Signals

The BIST status signals, STATUS_0 and STATUS_1, are combinationally derived from the self-test FSM state. The status encodings, shown in [Figure 509](#), are designed so that STATUS_1 indicates self-test completion, at which time STATUS_0 indicates the pass/fail condition.

Figure 509 Status Signal Values for Passing and Failing Self-Test

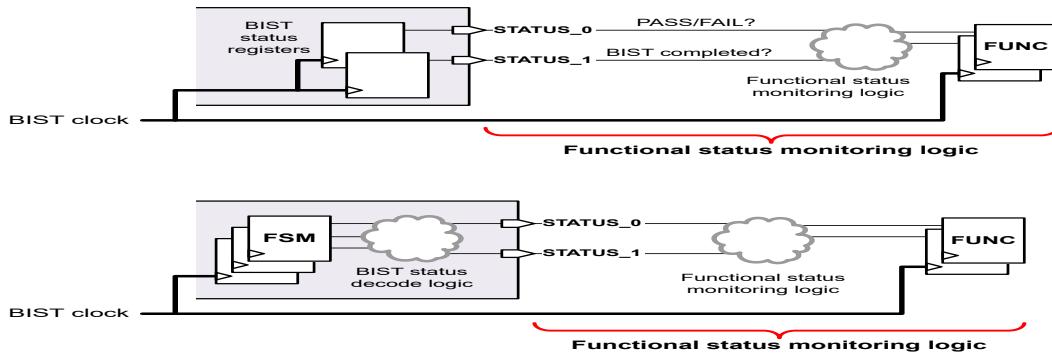


If you monitor the status signals using the BIST clock (or a clock synchronous to it), then

- No synchronization logic is needed.
- The status signal paths are synchronous single-cycle register-to-register paths.
- You can further process the status signals using combinational logic, as it becomes part of the register-to-register paths.

[Figure 510](#) shows an example of synchronous status monitoring logic.

Figure 510 Synchronous-Clock BIST Status Monitoring Logic

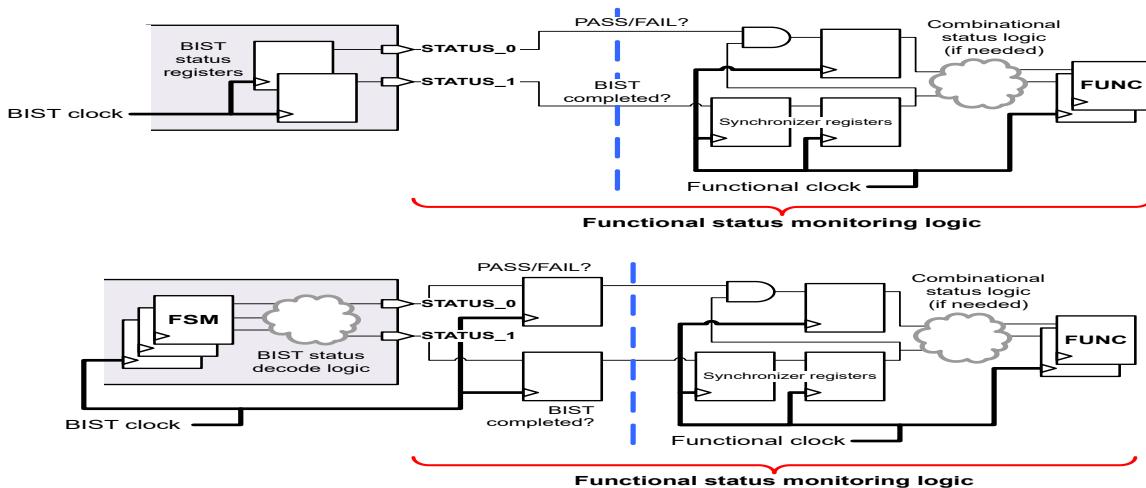


If you monitor the status signals using a clock asynchronous to the BIST clock, then

- You must implement synchronization logic to monitor the status signals.
- The status signals are combinational and must be re-registered on the BIST clock.
- The transition from RUNNING to PASS toggles both status signals. Take care to avoid race conditions between the completion (STATUS_1) and pass/fail (STATUS_0) signals when resynchronizing the status signals.

Figure 511 shows an example of asynchronous status monitoring logic. When the synchronized completion signal is asserted in the functional clock domain, the pass/fail signal is registered and stable (while START remains asserted).

Figure 511 Asynchronous-Clock BIST Status Monitoring Logic



When designing your interface logic, take the following into consideration: relative clock frequencies, clock-tree skew within each domain, and relative delays in the cross-domain status paths.

Caution:

The schematics in this section are conceptual examples only. You should implement status monitoring logic that meets your specific design requirements.

See Also

- [The STATUS_0 and STATUS_1 Signals on page 1013](#) for details on status signal behavior and values

Example LogicBIST Scripts

This topic provides the following example scripts:

- [Example Core Insertion Script Using Core Wrapping](#)
- [Example Core Insertion Script Using Test-Point Isolation](#)
- [Example Script to Automatically Set Seed and Signature Values](#)

Example Core Insertion Script Using Core Wrapping

This example script uses core wrapping to isolate the core during self-test. It also uses the random_resistant testability target (see [Automatically Inserted Test Points on page 305](#)) to improve coverage.

```
read_verilog ./top.v
current_design top
link
compile -scan

# define DFT signals
set_dft_signal -view spec -type ScanDataIn -port {SI1}
set_dft_signal -view spec -type TestMode -port {TM}
set_dft_signal -view spec -type lbistEnable -port {LBIST_EN}
set_dft_signal -view existing_dft -type MasterClock -port CLK1 \
    -timing {45 55}
set_dft_signal -view existing_dft -type MasterClock -port CLK2 \
    -timing {45 55}
set_dft_signal -view existing_dft -type Reset -port RSTN -active 0

# enable clients
set_dft_configuration -logicbist enable -wrapper enable

set_dft_configuration -testability enable
set_testability_configuration \
    -target random_resistant \
    -control_signal TM ;# enable during manufacturing test AND self-test
```

```

# ensure that TM is asserted during self-test too
set_test_point_element -control_signal LBIST_EN -type force_1 TM

# define and configure test modes
define_test_mode SCAN -usage wrp_if
define_test_mode LBIST -usage logicbist
set_scan_configuration -test_mode SCAN -chain_count 2 \
    -clock_mixing mix_clocks
set_logicbist_configuration -test_mode LBIST -base_mode SCAN \
    -chain_count 32 \
    -clock CLK1 \
    -pattern_counter_width 16 ;# maximum of 2^16 patterns

# preview and insert DFT
create_test_protocol
dft_drc
run_test_point_analysis ;# for random_resistant target
preview_dft -show all
insert_dft

# write out design netlist
write -format verilog -output top_no_seed_signature.vg -hierarchy

# write test protocol for TestMAX ATPG to calculate seed/signature
write_test_protocol -test_mode LBIST -output LBIST.spf

# write testbench for Verilog simulation to validate
# LogicBIST implementation
#
# (this command produces bist_tb.stil)
write_test -format stil -output bist_tb

```

Example Core Insertion Script Using Test-Point Isolation

This example uses the `core_wrapper` and `random_resistant` testability targets (see [Automatically Inserted Test Points on page 305](#)) to isolate the core during self-test and improve coverage, respectively.

```

read_verilog ./top.v
current_design top
link
compile -scan

# define DFT signals
set_dft_signal -view spec -type ScanDataIn -port {SI1}
set_dft_signal -view spec -type TestMode -port {TM}
set_dft_signal -view spec -type lbistEnable -port {LBIST_EN}
set_dft_signal -view existing_dft -type MasterClock -port CLK1 \
    -timing {45 55}
set_dft_signal -view existing_dft -type MasterClock -port CLK2 \
    -timing {45 55}

```

```

set_dft_signal -view existing_dft -type Reset -port RSTN -active 0

# enable clients
set_dft_configuration -logicbist enable

set_dft_configuration -testability enable
set_testability_configuration \
    -target core_wrapper \
    -control_signal LBIST_EN ;# enable ONLY during self-test
set_testability_configuration \
    -target random_resistant \
    -control_signal TM ;# enable during manufacturing test AND self-test

# ensure that TM is asserted during self-test too
set_test_point_element -control_signal LBIST_EN -type force_1 TM

# define and configure test modes
define_test_mode SCAN -usage scan
define_test_mode LBIST -usage logicbist
set_scan_configuration -test_mode SCAN -chain_count 2 \
    -clock_mixing mix_clocks
set_logicbist_configuration -test_mode LBIST -base_mode SCAN \
    -chain_count 32 \
    -clock CLK1 \
    -pattern_counter_width 16 ;# maximum of 2^16 patterns

# preview and insert DFT
create_test_protocol
dft_drc
run_test_point_analysis ;# for core_wrapper and random_resistant targets
preview_dft -show all -test_points all
insert_dft

# write out design netlist
write -format verilog -output top_no_seed_signature.vg -hierarchy

# write test protocol for TestMAX ATPG to calculate seed/signature
write_test_protocol -test_mode LBIST -output LBIST.spf

# write testbench for Verilog simulation to validate
# LogicBIST implementation
#
# (this command produces bist_tb.stil)
write_test -format stil -output bist_tb

```

Example Script to Automatically Set Seed and Signature Values

The following script excerpt shows how to automate TestMAX ATPG seed and signature computation, netlist modification, and testbench creation. The commands prior to the `insert_dft` command are omitted.

Note:

Using the `Tcl exec` command requires the current process to be forked, which can require a lot of memory when large designs are loaded.

```
# ...previous commands omitted...
insert_dft

# write out design netlist
write -format verilog -output top_no_seed_signature.vg -hierarchy

# write test protocol for TestMAX ATPG to calculate seed/signature
write_test_protocol -test_mode LBIST -output LBIST.spf

# write testbench for Verilog simulation to validate
# LogicBIST implementation
#
# (this command produces bist_tb.stil)
write_test -format stil -output bist_tb

# create Verilog testbench file
#
# (this command produces bist_tb.v and bist_tb.dat)
exec stil2verilog -replace bist_tb.stil bist_tb

# generate seed and signature value in TestMAX ATPG
#
# (this command produces serial.stil)
exec tmax -shell ./tmax_bist.tcl

# set seed and signature values in design and write out final netlist
set_logicbist_constants -file_name serial.stil
write -f verilog -h -o top.vg

quit
```

The following script is the `tmax_bist.tcl` script referenced by the preceding synthesis script.

```
read_netlist -library /project/libs/my_class.v
read_netlist top_no_seed_signature.vg
run_build top

# Enable LogicBIST DRC
set_drc -seq_comp_jtag_lbist_mode light_lbist
run_drc LBIST.spf

# Run LogicBIST ATPG for 133 patterns and 1 capture clock cycle
run_atpg -auto -jtag_lbist {1 133 1}
run_simulation
report_patterns -all

# write serial STIL file containing seed and signature values
write_patterns serial.stil -format stil -replace -unified -serial
```

```
quit ;# required to resume execution in dc_shell
```

32

Advanced LogicBIST Configuration

This chapter describes advanced features that can be used while inserting LogicBIST self-test circuitry into your design. These features can be used to improve self-test flexibility and reduce the implementation area.

This chapter includes the following topics:

- [Using Programmable LogicBIST Configuration Values](#)
- [Simplifying the MISR XOR Compressor](#)
- [Simplifying the Weighted Clock/Reset Logic](#)
- [Minimizing Reconfiguration MUXs Across Test Modes](#)
- [Choosing a Particular Integrated Clock-Gating Cell](#)
- [Implementing Burn-In Mode](#)
- [Implementing Power Ramp-Up and Ramp-Down Logic](#)
- [Implementing MISR Monitoring Logic](#)
- [Changing the Test Mode Used for Autonomous Self-Test](#)
- [Post-DFT Design Optimization](#)

Using Programmable LogicBIST Configuration Values

By default, the tool implements the LogicBIST logic with placeholder buses in the netlist for the following values:

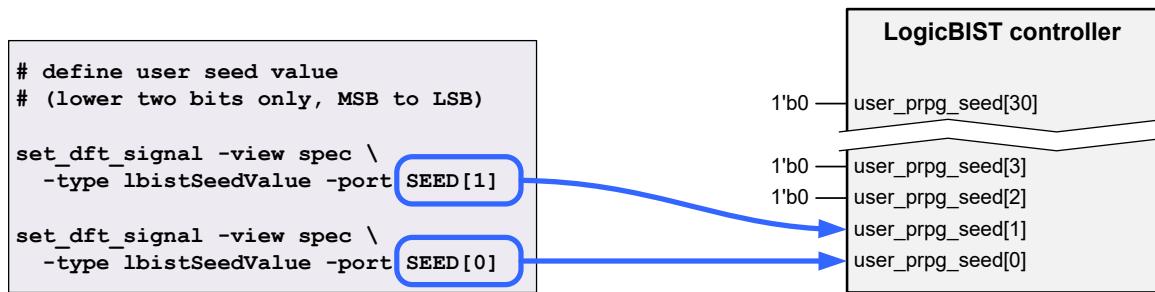
- User seed value - tied to logic 0 (eventually computed by TestMAX ATPG)
- User signature value - tied to logic 0 (eventually computed by TestMAX ATPG)
- User pattern value - tied to logic 0 (eventually computed by TestMAX ATPG)
- User shift value - automatically set to the longest shift chain length by DFT insertion (but can be overridden if needed)
- User OCC clock pulse pattern - tied to a single-pulse constant mask (2'b10)

Instead of setting these to hardcoded constant values in the netlist, you can make them programmable by driving them from ports or internal hookup pins. To do this, define DFT signals using the following signal types:

```
set_dft_signal -view spec -type lbistSeedValue ...
set_dft_signal -view spec -type lbistSignatureValue ...
set_dft_signal -view spec -type lbistPatternCount ...
set_dft_signal -view spec -type lbistShiftLength ...
set_dft_signal -view spec -type lbistCaptureCycleEnable ...
```

For each signal type, define the signal bits in order of most-significant bit (MSB) to least-significant bit (LSB). If fewer signals are defined than the bus width, the signals are justified against the LSB as shown in [Figure 512](#).

Figure 512 Assigning User-Defined Signals to Bus Bits



You can use the `-port` or `-hookup_pin` option when defining these signals. For bused ports, you can sort in dictionary order and define them all in a single command:

```
set_dft_signal -view spec -type lbistSeedValue \
    -port [sort_collection -dictionary -descending [get_ports {SEED[*]}]]
```

For hookup pins, define them one at a time in the required order:

```
# define a 31-bit initial PRPG seed register
for {set i 30} {$i >= 0} {incr i -1} {
    set_dft_signal -view spec -type lbistSeedValue \
        -hookup_pin CONFIG/SEED_reg[$i]
}

# define a 32-bit MISR expected signature register
for {set i 31} {$i >= 0} {incr i -1} {
    set_dft_signal -view spec -type lbistSignatureValue \
        -hookup_pin CONFIG/SIGNATURE_reg[$i]
}
```

When you define internally driven LogicBIST configuration signals, the `preview_dft` command reports the connections on a bitwise basis so you can confirm their correctness. For example,

```
*****
LogicBIST Compression
User signals information
*****
Shift counter data: top_U_LogicBISTController_bist/shift_count_data = 4'b1000
Pattern counter data: top_U_LogicBISTController_bist/user_pattern_count_data =
4'b0000
SEED CONNECTIONS:
*****
CONFIG/SEED_reg[0] connected to top_U_decompressor_bist/user_prpg_seed[0]
CONFIG/SEED_reg[1] connected to top_U_decompressor_bist/user_prpg_seed[1]
...
CONFIG/SEED_reg[29] connected to top_U_decompressor_bist/user_prpg_seed[29]
CONFIG/SEED_reg[30] connected to top_U_decompressor_bist/user_prpg_seed[30]

EXPECTED SIGNATURE CONNECTIONS:
*****
CONFIG/SIGNATURE_reg[0] connected to top_U_compressor_bist/user_misr_signature[0]
CONFIG/SIGNATURE_reg[1] connected to top_U_compressor_bist/user_misr_signature[1]
...
CONFIG/SIGNATURE_reg[28] connected to top_U_compressor_bist/user_misr_signature[28]
CONFIG/SIGNATURE_reg[29] connected to top_U_compressor_bist/user_misr_signature[29]
```

You can use the `find_seed` Tcl procedure from [SolvNet article 2220819, “Finding Optimal Seed Values for the LogicBIST PRPG”](#) to find an optimal sequence of seed values.

For port-driven signals, you can use the `set_logicbist_constants` Tcl procedure from [SolvNet article 2231010, “Setting the Seed and Signature Values in a LogicBIST Design”](#) to write out a VCS command file that forces the desired values for simulation.

Simplifying the MISR XOR Compressor

When the number of internal chains is equal to the MISR size, the XOR compressor inside the LogicBIST compressor is automatically removed and a direct connection is performed between the compressed scan chains and the MISR.

Set the following options of the `set_logicbist_configuration` command to the same value to implement this simplification:

```
set_logicbist_configuration ... \
    -chain_count chain_count_value \
    -misr_width chain_count_value
```

There is no equivalent removal of the XOR phase shifter in the LogicBIST decompressor because it is needed to remove correlations from the LFSR values.

Simplifying the Weighted Clock/Reset Logic

Weighted clock/reset capture groups use a seven-bit comparator/decoder driven by the first seven bits of the clock chain (`clk_chain_val[6:0]`). Any arbitrary positive group weight values can be specified, but the values are scaled (if needed) to a total weight of 128 for implementation.

Thus, certain weight conventions simplify the comparator logic. For example,

- Two groups, each with a weight of 50%, use comparator ranges that are 64 values wide, which requires only `clk_chain_val[6]`.
- Groups using weights that are a multiple of 25% use comparator ranges that are 32 values wide, which requires only `clk_chain_val [6:5]`.

Weight values that result in boundary (comparator) values with a longer sequence of least-significant zeros, like 32 (7'b01**00000**) or 96 (7'b11**00000**), require less comparator logic than boundary values with a shorter sequence of least-significant zeros, like 34 (7'b010001**0**) or 98 (7b'110001**0**).

If your design is area sensitive, you can consider this effect while determining weight values that meet your coverage requirements.

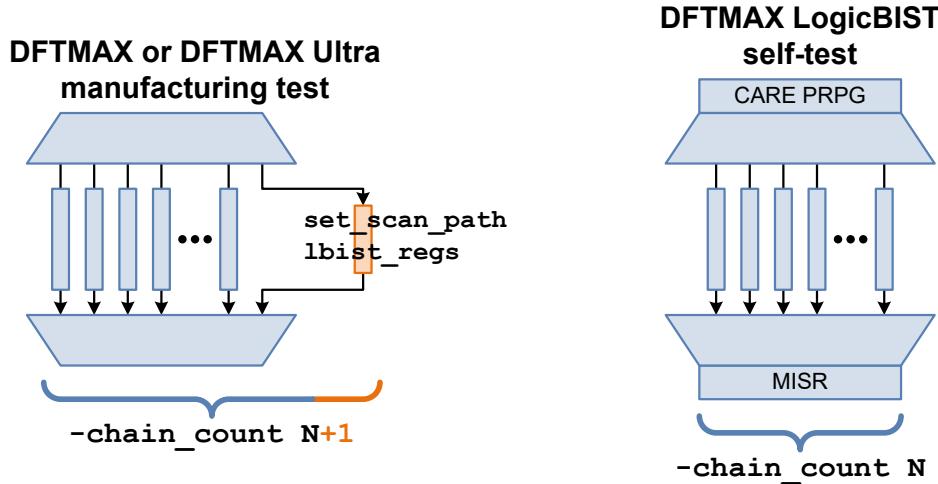
Minimizing Reconfiguration MUXs Across Test Modes

When you implement DFTMAX or DFTMAX Ultra scan compression alongside LogicBIST self-test, you can align their compressed chain structures to minimize reconfiguration MUXs.

However, the self-test registers are scannable in manufacturing test modes (to ensure that the self-test logic itself is tested) but not in self-test mode. As a result, simply specifying the same compressed chain count for both modes does not guarantee alignment.

To resolve this, you can define additional compressed scan chains that contain only the scannable self-test registers, as shown in [Figure 513](#).

Figure 513 Aligning the Compressed Chain Structures



For details, see

- SolvNet article 2656631, "Minimizing Reconfiguration MUXs in LogicBIST Designs."
- SolvNet article 2220819, "Finding Optimal Seed Values for the LogicBIST PRPG."

Choosing a Particular Integrated Clock-Gating Cell

The LogicBIST architecture uses clock gating for the following constructs to reduce area and power consumption:

- MISR
- PRPG
- Pattern counter
- Shift counter
- LogicBIST clock controller (external or OCC)

By default, the tool builds discrete clock-gating logic using separate latch and combinational gate cells. To use integrated clock-gating cells (ICGs) instead, enable ICG insertion and specify the desired ICG library cell using the following variables:

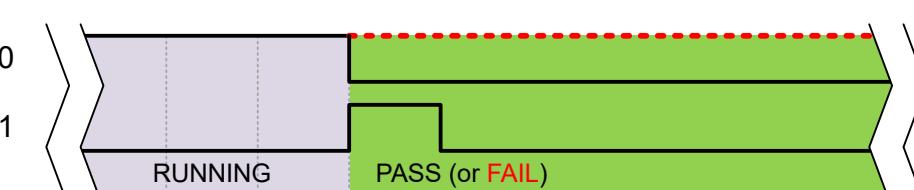
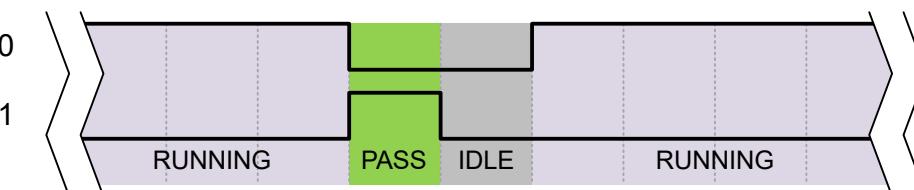
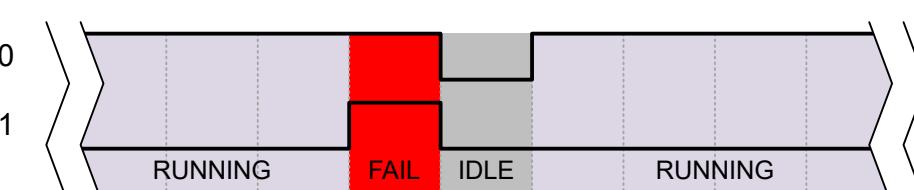
```
set_app_var test_occ_insert_clock_gating_cells true
set_app_var test_icg_p_ref_for_dft ICG_library_cell
```

Implementing Burn-In Mode

LogicBIST provides a *burn-in mode* feature that runs autonomous self-test continuously as long as the START signal is asserted. The scan and capture activity stresses the tested logic and causes continuous power draw during self-test. Power consumption can be controlled by adjusting the clock frequency.

Burn-in operation is configured by two DFT signals. [Table 61](#) shows how they affect self-test while the START signal is held asserted.

Table 61 Burn-In Configuration Signals and Behaviors

Self-test success?	IbistBurnInEnable signal value	IbistBurnInStopOnFail signal value
Pass or fail	0 (burn-in disabled)	X
		
Pass	1	X (don't-care when passing)
		
Fail	1	0 (continue on fail)
		
Fail	1	1 (stop on fail)

Self-test success?	IbistBurnInEnable signal value	IbistBurnInStopOnFail signal value
STATUS_0		
STATUS_1	RUNNING	FAIL

When the IbistBurnInEnable signal is de-asserted, the LogicBIST engine runs in its normal mode of operation to completion, generating the pass or fail indication as described in [The STATUS_0 and STATUS_1 Signals on page 1013](#).

The burn-in capability is not implemented by default. To implement it, specify the following option:

```
dc_shell> set_logicbist_configuration -burn_in enable
```

Define the burn-in configuration signals on existing ports using the following signal types:

```
dc_shell> set_dft_signal -view spec \
           -type lbistBurnInEnable -port my_burnin_enable
dc_shell> set_dft_signal -view spec \
           -type lbistBurnInStopOnFail -port my_burnin_SOF
```

You can also define these signals on internal pins using the `-hookup_pin` option.

If you do not define these signals, the tool automatically creates them using the following signal port names: `burnin_mode`, `fail_mode`.

The burn-in capability is implemented by modifying existing states in the BIST state machine rather than adding states; the area overhead is negligible.

Implementing Power Ramp-Up and Ramp-Down Logic

By default, when self-test starts, it immediately operates at the provided clock frequencies. When self-test completes, it stops the clock to the functional logic. These abrupt changes in functional logic activity could cause transient spikes in the power rail voltage.

However, you can use the power ramp-up and ramp-down features to smooth these transitions in power consumption. These features are disabled by default.

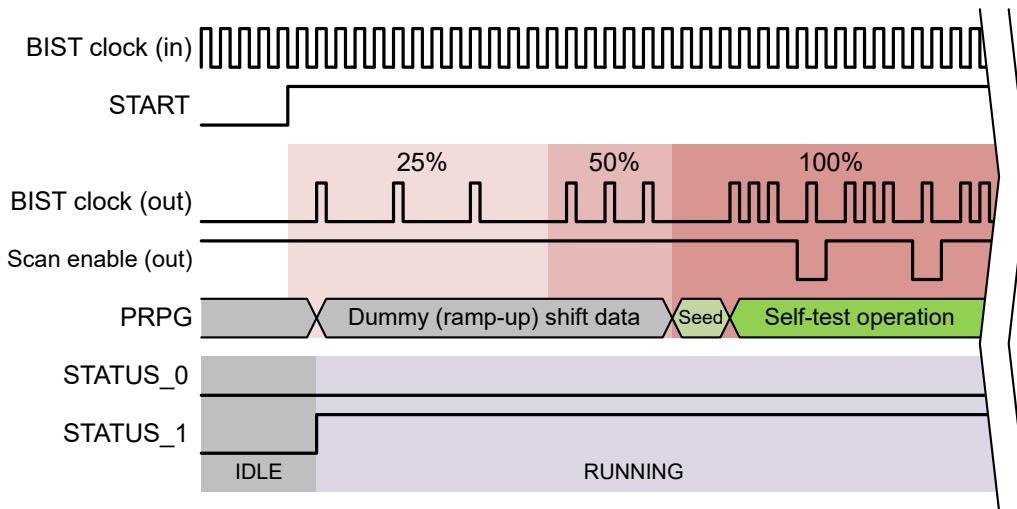
Power Ramp-Up

To implement power ramp-up, specify the following option:

```
dc_shell> set_logicbist_configuration \
           -power_ramp_up enable
```

The self-test logic is augmented to shift through two “ramp-up” dummy patterns—one at 25% frequency, then one at 50% frequency—before beginning full-frequency self-test, as shown in [Figure 514](#).

Figure 514 Power Ramp-Up Clocking



The BIST controller creates the dummy ramp-up patterns by modulating the clock-enable signal within the LogicBIST clock controller. The dummy patterns are a preamble to the actual self-test operation; they perform no testing and are not included in the pattern count specification. The seed value is loaded in the PRPG after ramp-up completes.

Enabling ramp-up patterns increases self-test duration by the equivalent of six full-frequency self-test patterns. The testbench written out by the `write_test` command includes this additional duration.

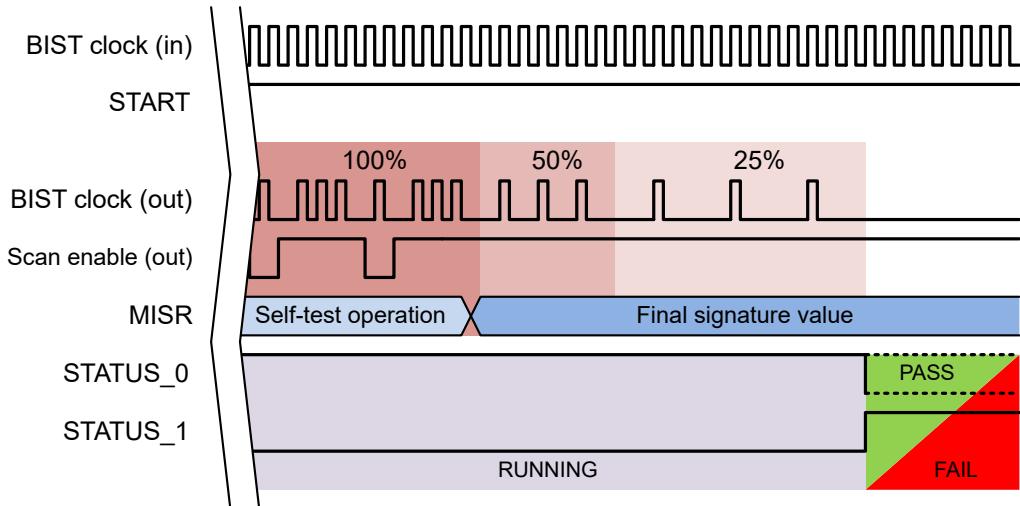
Power Ramp-Down

Power ramp-down requires that power ramp-up also be enabled. To implement them, specify the following options:

```
dc_shell> set_logicbist_configuration \
           -power_ramp_up enable \
           -power_ramp_down enable
```

The self-test logic is augmented to shift through two “ramp-down” dummy patterns—one at 50% frequency, then one at 25% frequency—after full-frequency self-test completes, as shown in [Figure 514](#).

Figure 515 Power Ramp-Down Clocking



The dummy patterns are a postamble to the actual self-test operation; they perform no testing and are not included in the pattern count specification. The status signals are not asserted until ramp-down completes.

Enabling ramp-up and ramp-down patterns together increases self-test duration by the equivalent of twelve full-frequency self-test patterns.

Implementing MISR Monitoring Logic

In designs with LogicBIST self-test, it might be useful to monitor the MISR during self-test. For example, an on-chip microcontroller could initiate self-test, then verify that the MISR is toggling as expected.

You can use the `lbistMISROutput` signal type to monitor one or more MISR bits. Usage is as follows:

- Define the signals in downward order (the last signal defined is driven by bit zero of the MISR).
- You can monitor a single bit (bit zero) or the entire MISR.
- To monitor the MISR using output ports, use the `-port` option:

```
set_dft_signal -view spec -type lbistMISROutput \
    -port {MISR[2] MISR[1] MISR[0]}
```

- To monitor the MISR using hookup pins, use the `-hookup_pin` option:

```
set_dft_signal -view spec -type lbistMISROutput \
    -hookup_pin my_MISR_check_reg[2]/D
set_dft_signal -view spec -type lbistMISROutput \
    -hookup_pin my_MISR_check_reg[1]/D
set_dft_signal -view spec -type lbistMISROutput \
    -hookup_pin my_MISR_check_reg[0]/D
```

You do not need to enable the internal pins flow, as the signal connection does not affect DRC.

Changing the Test Mode Used for Autonomous Self-Test

By default, autonomous self-test operation requires that the design be placed in mission mode when `LBIST_EN` is asserted. In other words, the test-mode signals must be asserted to their mission-mode encoding. The tool chooses this encoding as described in [Ensuring the Required Test Mode for Autonomous Self-Test on page 1054](#).

However, if needed, you can specify that another test mode be used for autonomous self-test operation. To do this, specify the desired test mode with the following option:

```
set_logicbist_configuration -self_test_mode my_selftest_mode
```

The specified mode must be previously defined with the `define_test_mode` command. It cannot be a scan compression mode of any type (LogicBIST, DFTMAX, or DFTMAX Ultra).

When you use this feature, references to mission mode in this documentation apply to the specified mode instead.

Post-DFT Design Optimization

LogicBIST self-test provides some unique considerations in the synthesis flow. BIST is often used in area-sensitive designs. Propagating the seed and signature values as hardcoded constants into the design logic yields an area savings, but at the cost of no longer being able to change their values (such as for a functional ECO).

These aspects of post-DFT optimization are described in more detail in the following topics:

- [Post-DFT Optimization and BIST Constants](#)
- [Preserving the BIST Constants in a compile Flow](#)
- [Preserving the BIST Constants in a compile_ultra Flow](#)

- Regenerating Seed and Signature Values after Design Changes
- Ungrouping LogicBIST Blocks for Additional Area Reduction

Post-DFT Optimization and BIST Constants

LogicBIST self-test requires that the values in [Table 62](#) be specified in the design logic.

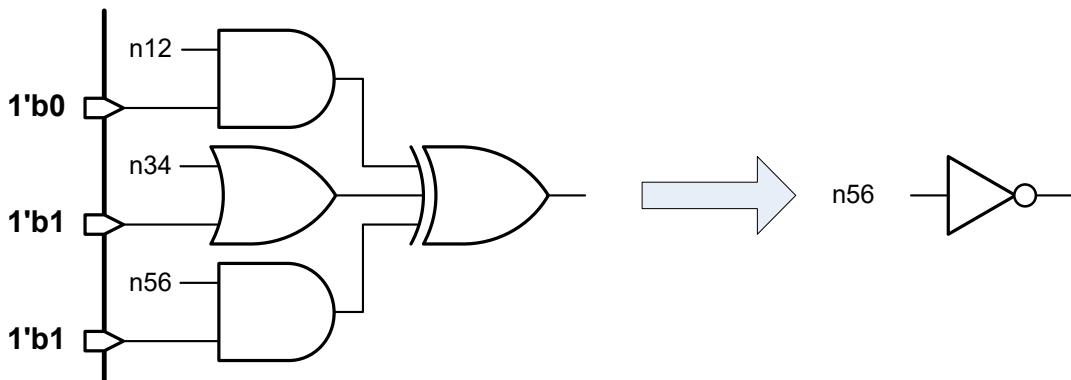
Table 62 User-Specified LogicBIST Constant Values

BIST value	Hierarchical bused pins
PRPG seed	<i>design_U_decompressor_mode/user_prpg_seed[*]</i>
MISR signature	<i>design_U_compressor_mode/user_misr_signature[*]</i>
Pattern count	<i>design_U_LogicBISTController_mode/user_pattern_count_data[*]</i>
Shift count ¹²	<i>design_U_LogicBISTController_mode/shift_count_data[*]</i>

These values are set in the design after determining the values in TestMAX ATPG, as described in [Setting the Seed and Signature Values in Synthesis on page 1049](#).

By default, these BIST values are driven by constants. If post-DFT optimization propagates these BIST constants into the design, they become permanently integrated into the logic and their values cannot be changed, as shown in [Figure 516](#).

Figure 516 BIST Constant Propagation Into Block Logic



This prevents you from performing design ECOs later in the flow, which would require new seed and signature values to be set in the design.

12. This value is set by DFT insertion and does not need to be user-modified (unless the BIST mode scan length is manually changed later in the flow). Accordingly, this bus name has no “user_” prefix.

If you use programmable (register-driven) values as described in [Using Programmable LogicBIST Configuration Values on page 1062](#), you do not need to worry about constant propagation for those values.

Preserving the BIST Constants in a compile Flow

The `compile` command does not propagate constants into DFT-created blocks (which have the `is_test_circuitry` attribute set to `true`) when boundary optimization is enabled. The only way to propagate such constants is to ungroup the DFT blocks, via the `ungroup`, `set_ungroup`, or `compile -auto_ungroup` command.

To preserve the BIST constants, do not ungroup the LogicBIST controller, decompressor, or compressor blocks.

See Also

- [Ungrouping LogicBIST Blocks for Additional Area Reduction on page 1075](#) for information on ungrouping the LogicBIST blocks

Preserving the BIST Constants in a compile_ultra Flow

By default, the `compile_ultra` command propagates constants into blocks, even DFT-created blocks (which have the `is_test_circuitry` attribute set to `true`). To preserve the BIST constants while allowing full optimization for the rest of the logic, disable constant propagation specifically on these pins, with a "user_" name prefix:

```
# disable constant propagation for BIST constants
# (these pins have a "user_" prefix)
set_compile_directives -constant_propagation false \
[get_pins -of [get_cells -hier * -filter {is_hierarchical == true && \
is_test_circuitry == true}] -filter {name =~ user_*}]

# perform post-DFT optimization
compile_ultra -scan -incremental
```

These commands do not prevent constant propagation on the `shift_count_data[*]` bus because the shift length set by the `insert_dft` command does not typically need to be changed and can be optimized into the logic. To allow for functional ECOs that affect the shift length, disable constant propagation on this bus too.

To propagate the constants later in the flow, ungroup the LogicBIST blocks or reenable constant propagation for the pins, then perform an incremental compile.

See Also

- [Ungrouping LogicBIST Blocks for Additional Area Reduction on page 1075](#) for information on ungrouping the LogicBIST blocks

Regenerating Seed and Signature Values after Design Changes

You must regenerate the LogicBIST signature value whenever the functional or DFT logic changes. This includes design logic ECOs and scan chain reordering or repartitioning in layout. In addition, to improve coverage, the seed value can also be changed.

To regenerate the seed and signature values, do the following:

1. Write out the modified Verilog design netlist. You can do this from any source, such as the Design Compiler tool, the layout tool, or a text editor.
2. Modify the TestMAX ATPG seed and signature generation script to use the modified design netlist, then run it to write out a STIL pattern file with new seed and signature values. See [Computing the Seed and Signature Values in TestMAX ATPG on page 1047](#).

When generating new seed and signature values for a design, existing seed and signature values in the design are not considered. Accordingly,

- If the logic changes are small (such as a small design logic ECO) and you have an optimal seed value you want to keep, specify it with the `add_lbist_seeds` command.
 - If the logic changes are significant (such as scan reordering), you will likely obtain better results by finding a new seed value.
3. Read the modified Verilog netlist from step 1 into the Design Compiler tool. No timing or DFT constraints are needed.
 4. Use the `set_logicbist_constants` command to apply the new seed and signature values from the STIL patterns generated in step 2. See [Setting the Seed and Signature Values in Synthesis on page 1049](#).
 5. Write out the updated netlist, which contains new seed and signature values.
 6. Reuse the original testbench to simulate the updated netlist from step 5. See [Simulating Autonomous BIST Operation on page 1050](#).

The following TestMAX ATPG script generates new seed and signature values for an ECO-modified netlist:

```
read_netlist -library /project/libs/my_class.v
read_netlist top_eco_oldseedsig.vg
run_build top

# Enable LogicBIST DRC
set_drc -seq_comp_jtag_lbist_mode light_lbist
run_drc LBIST.spf

# Run LogicBIST ATPG for 133 patterns and 1 capture clock cycle
```

```

run_atpg -auto -jtag_lbist {1 133 1}
run_simulation
report_patterns -all

# write serial STIL file containing seed and signature values
write_patterns serial_eco.stil -format stil -replace -unified -serial

quit ;# required to resume execution in dc_shell

```

The following Design Compiler script updates the design netlist with the new seed and signature values:

```

# (library setup not shown)

# read netlist containing ECO change and pre-ECO seed/signature values
read_verilog top_eco_oldseedsig.vg
current_design top
link

# set seed and signature values in design and write out final netlist
set_logicbist_constants -file_name serial_eco.stil
write -format verilog -hierarchy -output top_eco_newseedsig.vg

```

See Also

- [SolvNet article 2231010, “Setting the Seed and Signature Constant Values in a LogicBIST Design”](#) to obtain the `set_logicbist_constants` Tcl procedure
- [The LogicBIST Operational Modes on page 1012](#) for more information on ATPG mode versus autonomous mode

Ungrouping LogicBIST Blocks for Additional Area Reduction

After DFT insertion and seed and signature setting, you can achieve significant area reduction by grouping the LogicBIST controller, clock or OCC controller(s), and codec together, then ungrouping the hierarchy inside that block. This allows the control logic to be highly optimized between those blocks while still containing the logic within a block. You can use this technique with the `compile` or `compile_ultra` command.

The following commands perform this ungrouping. Hierarchical clock-gating cells are left grouped. This example assumes that all cells to be grouped exist at the top level. Change the "bist" suffix in the filter expression to match your LogicBIST test mode name.

Caution:

Ungrouping propagates the BIST constants, which prevents future modifications to their values. Use this only when the design is finalized (including scan reordering) or when you can rerun synthesis and DFT insertion if the design logic changes.

```
# group LogicBIST controller and OCC blocks together into a new
# block named "LogicBIST"
group -cell_name LogicBIST -design_name LogicBIST \
[get_cells * -filter {ref_name == LOGICBIST_CONTROLLER || \
ref_name =~ top_DFT_clk_mux_* || name =~ *compressor_bist}]

# flatten everything in the new LogicBIST block except hierarchical
# clock-gating cells
set_ungroup [get_cells -hierarchical -filter {is_hierarchical == true \
&& full_name =~ LogicBIST/* && full_name !~ *clkgt*}]

# incrementally compile
compile_ultra -scan -incremental
```

33

LogicBIST Limitations and Known Issues

This chapter contains the limitations and known issues that apply to LogicBIST self-test.

This chapter contains the following topic:

- [LogicBIST Limitations and Known Issues](#)
-

LogicBIST Limitations and Known Issues

The following requirements, limitations, and known issues apply to LogicBIST self-test:

- LogicBIST settings are not stored in .ddc files; you must apply any `set_logicbist_configuration` settings if you read a pre-DFT .ddc file back in.
- Designs that capture X values are not supported.
- Clock-gating cells require a dedicated ScanEnable signal defined with the `-usage {clock_gating}` option of the `set_dft_signal` command (although [SolvNet article 3055862](#) provides a workaround for this limitation).

If there is no such signal, no clock-gating testability logic is created. If the signal is defined with `-usage {scan_clock_gating}`, scan cells are hooked up to the clock-gating testability scan-enable signal, which is incorrect.

TestMode signals cannot be used as clock-gating control signals.

- External (uncompressed) chains cannot be used in LogicBIST test modes. See [Inserting LogicBIST in Designs With External Chains on page 1041](#).
- Clock-gating cells with pre-existing test-mode pin connections must be identified prior to DFT insertion. For details, see the TEST-130 man page.
- The LogicBIST test mode requires at least one user-defined scan-in signal; it is not automatically created.
- Sharing scan-in ports with functional ports is not supported.
- You can integrate a core that contains a LogicBIST test mode, but there is no automation provided to access the LogicBIST functionality at the integration level.

- For designs with OCC controllers,
 - A mix of non-OCC-controlled and internal OCC-controlled scan clock domains is not supported. In this case, you must also control the port-driven clocks with an OCC controller.
 - All OCC controllers must have the same clock chain length.
 - If you are using weighted clock capture groups, there must be at least seven clock chain bits across all clock chains in the design.
 - Synchronous OCC controllers are not supported.
 - User-defined OCC controllers are not supported.
 - Existing OCC controllers inside DFT-inserted cores are not supported.
 - External clock chains cannot be used in LogicBIST test modes. See [Inserting LogicBIST in Designs With External Chains on page 1041](#).
 - You cannot use ANDOR22 library cells for the clock ORing logic. See [Specifying Library Cells for the Clock-ORing Logic on page 539](#).
- If any DFT signals use bused ports, the bus indexes must be ordered highest-to-lowest.
- If you are using test points, you must specify an existing clock as the test point clock. You cannot use the DFT-created default test point clock.
- The following DFT features are not supported:
 - Pipelined scan enable
 - DFT partitions
 - Multiple LogicBIST test modes
 - Domain-based scan enable
 - Terminal lock-up latches
 - Post-DFT DRC
 - Hybrid flow
- In TestMAX ATPG,
 - The `-observe_file` option of the `run_atpg` command is not supported.
- In TestMAX Diagnosis,
 - Diagnostics is not supported.

Appendices

A

DFT Attributes

This appendix describes DFT-related attributes available in DFT Compiler and TestMAX DFT tools.

The DFT attributes are listed in the following tables:

- [Cell Attributes](#)
- [Design Attributes](#)
- [Pin Attributes](#)
- [Port Attributes](#)

Note:

All attributes are undefined if the described conditions are not met.

Cell Attributes

[Table 63](#) shows the DFT attributes for cell objects.

Table 63 Attributes of the cell Object Class

Attribute name	Type	Description
cell_is_test_only	Boolean	This attribute is true for lock-up latches, retiming flip-flops, and pipeline registers.
dft_dont_connect_clock_gate_of_register	Boolean	This attribute is true for registers specified with the -dont_connect_cgs_of option of the set_dft_clock_gating_configuration command
is_test_circuitry	Boolean	This attribute is true for any cells inserted by the insert_dft command.

Attribute name	Type	Description
lockup_cell_to_segment_attr	String	This attribute is set on lock-up latches. It takes the form <i>cell_name;before after</i> which describes which scan cell it is associated with and whether that scan cell is before or after the lock-up latch.
retiming_flop	Boolean	This attribute is <code>true</code> for retiming flip-flops inserted by the <code>insert_dft</code> command.
scan_element	Boolean	This attribute is set by the <code>set_scan_element</code> command. It is undefined if the <code>set_scan_element</code> command is not applied, even if the scan cell is scanned by default or unscanned due to a DRC violation.
scan_lockup	Boolean	This attribute is <code>true</code> for lock-up latches inserted by the <code>insert_dft</code> command.
scanned_by_test_compiler	Boolean	This attribute is <code>true</code> for cells that are scan-replaced by the <code>compile -scan</code> , <code>compile_ultra -scan</code> , or <code>insert_dft</code> commands. Note that this attribute is set to <code>true</code> for DRC-violating cells that remain scan-replaced but are excluded from scan chains, and it is set to <code>false</code> for the unscanned registers in a shift register; thus, it is not a reliable indicator that the cell is in a scan chain.
shift_register_flop	Boolean	This attribute is <code>true</code> for all unscanned shift-register cells after the scanned first (head) element.
shift_register_head	Boolean	This attribute is <code>true</code> for the scanned first (head) element of an identified shift-register segment.
test_dft_cell_is_skew_group	Boolean	This attribute is <code>true</code> for cells that are part of a scan skew group, defined with the <code>set_scan_skew_group</code> command. This attribute is set by the <code>preview_dft</code> and <code>insert_dft</code> commands.
test_dft_xcellViolation	Boolean	This attribute is <code>true</code> for static-X cells identified by the <code>dft_drc</code> command. Static-X analysis is enabled by the <code>-static_x_analysis</code> option of the <code>set_dft_drc_configuration</code> command.
test_scan_suppress_toggling	Boolean	This attribute is <code>true</code> for gating cells inserted by functional output gating, which is configured by the <code>set_scan_suppress_toggling</code> command.

Attribute name	Type	Description
testdb_test_cell_violated	Boolean	This attribute is set to <code>true</code> for cells that have DRC violations. Pre-DFT DRC and post-DFT DRC both update this attribute.

Design Attributes

[Table 64](#) shows the DFT attributes for design objects.

Table 64 Attributes of the design Object Class

Attribute name	Type	Description
current_dft_partition	String	This attribute indicates the current DFT partition set by the <code>current_dft_partition</code> command.
shift_registers_extracted	Boolean	This attribute is <code>true</code> on a design compiled with the <code>compile_ultra</code> command if at least one shift register was identified within the design hierarchy. This attribute is not set on subdesigns within the hierarchy that contain identified shift registers.

Pin Attributes

[Table 65](#) shows the DFT attributes for pin objects.

Table 65 Attributes of the pin Object Class

Attribute name	Type	Description
created_by_test_compiler	Boolean	This attribute is <code>true</code> for pins created on hierarchical cells by the <code>insert_dft</code> command to route new signals.
created_during_dft		

Attribute name	Type	Description
is_clock_gate_test_pin	Boolean	This attribute is <code>true</code> for test pins of clock-gating cells automatically or manually identified for DFT Compiler. It is also <code>true</code> for test pins of integrated clock-gating cells instantiated in the design that have not been manually identified. It is <code>false</code> for all other pins.
signal_type	String	<p>This attribute is set on leaf cell and hierarchical cell pins that have a DFT signal type.</p> <p>Possible values include:</p> <ul style="list-style-type: none"> <code>test_scan_enable</code> <code>test_scan_in</code> <code>test_scan_out</code> <p>An <code>_inverted</code> suffix on the value indicates an inverted signal value.</p>
testdb_autofix_d1testdb_autofix_d2t estdb_autofix_d3testdb_autofix_d4te stdb_autofix_d5testdb_autofix_d6tes tdb_autofix_d9testdb_autofix_d12tes tdb_autofix_d17	Boolean	These attributes are <code>true</code> for pins with the corresponding DRC violation that can be fixed by AutoFix. These attributes are set by pre-DFT DRC even if AutoFix is not enabled.
testdb_clock_name_and_edge_for_pin	String	<p>After DFT insertion, this attribute is defined on scan cell clock pins to indicate the test clock name and triggering edge-timing value as follows:</p> <p><code>clock_source;trigger_edge</code></p> <p>Clocks with internal pins are represented as:</p> <p><code>clock_source:int_pin;trigger_edge</code></p>

Port Attributes

Table 66 shows the DFT attributes for port objects.

Table 66 *Attributes of the port Object Class*

Attribute name	Type	Description
created_by_test_compiler	Boolean	This attribute is <code>true</code> for ports created for the current design by the <code>insert_dft</code> command to route new signals.
created_during_dft		

B

Legacy Test Point Insertion

This appendix documents the legacy test point functionality that results when you *do not* enable the testability client. For best results, use the newer functionality described in [Inserting Test Points on page 293](#).

The legacy test point capabilities are described in the following topics:

- [Introduction](#)
 - [Differences Between Newer and Legacy Test Point Features](#)
 - [Test Point Types](#)
 - [Force Test Points](#)
 - [Sharing Test Point Scan Cells](#)
 - [Automatically Inserted Test Points \(Legacy\)](#)
 - [User-Defined Test Points Example](#)
 - [Previewing the Test Point Logic](#)
 - [Inserting the Test Point Logic](#)
-

Introduction

Test points are points in the design where DFT Compiler inserts logic to improve the testability of the design. The tool can automatically determine where to insert test points to improve test coverage and reduce pattern count. You can also manually define where test points are to be inserted.

Caution:

This section documents the legacy test point functionality that results when you *do not* enable the testability client by using the following command:

```
dc_shell> set_dft_configuration -testability enable
```

It is recommended that you use the improved functionality provided by the testability DFT client, as described in [Inserting Test Points on page 293](#).

Differences Between Newer and Legacy Test Point Features

When the legacy test point functionality is used, the test point functionality differs from the newer testability test point functionality as follows:

- During capture, force and control registers no longer hold state.
- Observe registers might be reused as control registers (because control registers no longer need to hold state).
- The following three-state test point types are supported:
 - `control_z0`, `control_z1`, and `control_z01`
 - `force_z0`, `force_z1`, and `force_z01`
- The following options of the `set_test_point_element` command are supported:
 - `-power_saving disable | enable`
 - `-test_points_per_test_point_enable tp_count`
 - `-scan_source_or_sink enable | disable`
 - `-source_or_sink port_pin_name`
 - `-scan_test_point_enable enable | disable`
 - `-test_point_enable port_pin_name`

Test Point Types

The *force* and *control* test point types allow signals within logic cones to be actively controlled during test mode to improve the controllability of the logic. These test point types require the insertion of a multiplexer to conditionally override the original signal value, resulting in a slight delay and area penalty.

The *observe* test point type passively captures the value of selected hard-to-observe signals to improve the observability of the logic. No additional levels of logic are inserted along the path of the observed signal, but the extra observation logic does slightly increase the capacitive loading of the observed signal.

Test points are described in the following topics:

- [Force Test Points](#)
- [Control Test Points](#)
- [Observe Test Points](#)

Note:

The test point schematics in these topics show the functional operation of the test points. During synthesis, constant logic is simplified, and the test point logic is optimized into the surrounding logic.

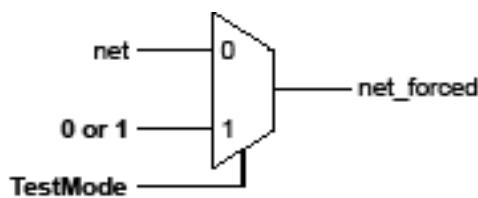
Force Test Points

Force test points are used when a value must be forced throughout the entire test session. The following force test point types are available:

- force_0
- force_1
- force_01
- force_z0
- force_z1
- force_z01

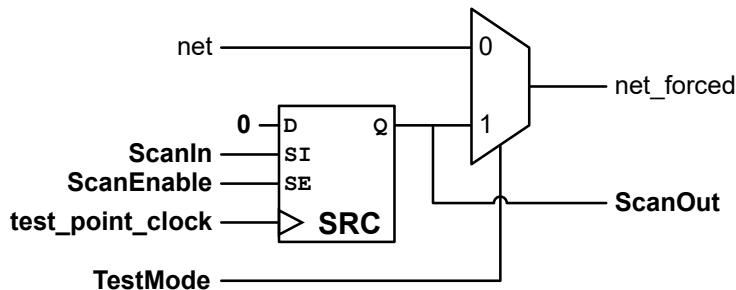
The `force_0` and `force_1` test point types allow a signal to be replaced with a constant 0 or constant 1 value throughout the entire test session. These test point types are useful when a particular signal must be forced to a known value for testability purposes. A multiplexer is used to replace the original signal with a fixed constant 0 or 1 value when the `TestMode` signal is asserted. See [Figure 517](#).

Figure 517 Example of a force_0 or force_1 Test Point



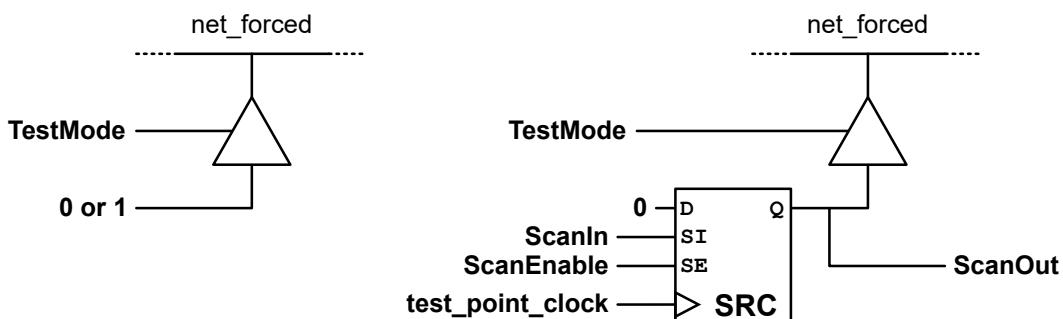
The `force_01` test point type allows a signal to be replaced with a scan-selected value throughout the entire test session. The scan-selected value comes from a source signal scan register, allowing the forced value to change for each test vector. A multiplexer is used to replace the original signal with the output of this scan register when the `TestMode` signal is asserted. See [Figure 518](#).

Figure 518 Example of a force_01 Test Point



The `force_z0`, `force_z1`, and `force_z01` test point types allow either a constant value or a scan-selected source signal value to be driven onto a tristate bus that is guaranteed to have no other active drivers during test mode. See [Figure 519](#).

Figure 519 Examples of force_z0, force_z1, and force_z01 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

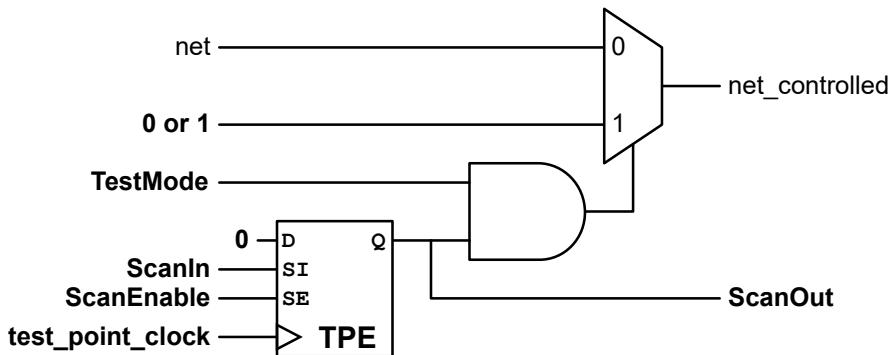
Control test points are used when a hard-to-control signal should be controllable (selectively forced) for some test vectors but left unaltered for others. Control test points are typically inserted to increase the fault coverage of the design. The following control test point types are available:

- `control_0`
- `control_1`
- `control_01`
- `control_z0`

- control_z1
- control_z01

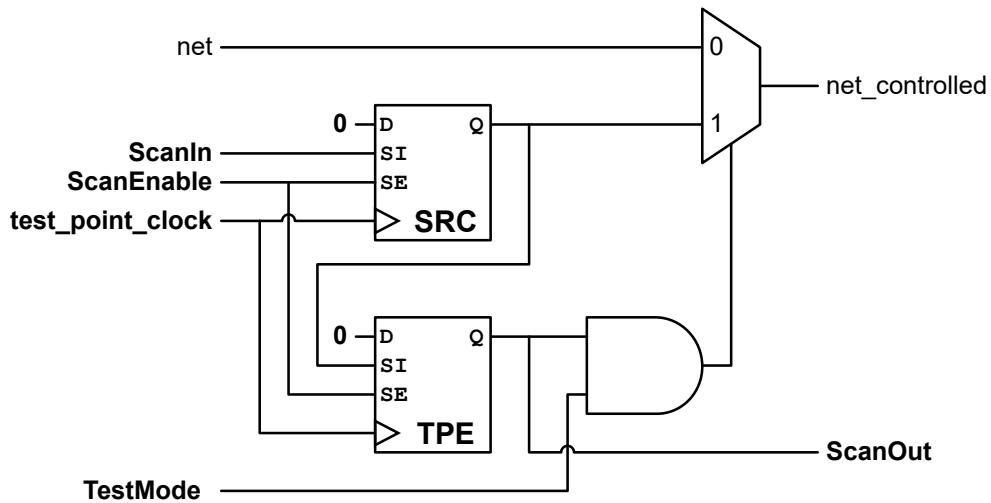
A `control_0` or `control_1` test point is built with a multiplexer, an AND gate, and a test point enable scan register. When `TestMode` is not asserted, the signal always retains its original value. When `TestMode` is asserted, the signal is forced with a fixed constant 0 or 1 value only when the output of the test point enable scan register selects the constant value. This allows the test program to select either the original signal behavior, or the constant-forced behavior on a vector-by-vector basis. This has the advantage of being able to control the signal for some test vectors without losing the observability of the upstream logic for the remaining vectors. See [Figure 520](#).

Figure 520 Example of a control_0 or control_1 Test Point



A `control_01` test point is similar to the `control_0` and `control_1` test point types, except that a scan-selected source signal value from a scan register is selectively driven onto the net on a vector-by-vector basis. As a result, the `control_01` test point requires two scan cells per control point, one for the source signal value and one for the enable register that specifies that the source signal should be driven. See [Figure 521](#).

Figure 521 Example of a control_01 Test Point



The `control_z0`, `control_z1`, and `control_z01` test point types allow either a constant value or a scan-selected source signal value to be selectively driven onto a bus that might be in a high-impedance state for some vectors but not for others. See [Figure 522](#) and [Figure 523](#).

Figure 522 Example of a control_z0 or control_z1 Test Point

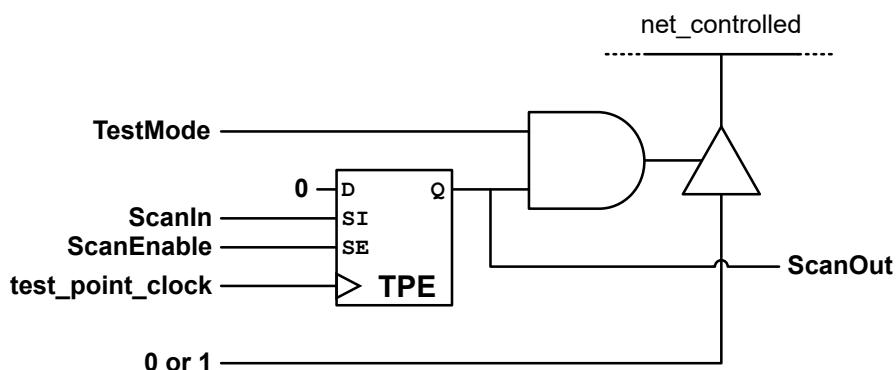
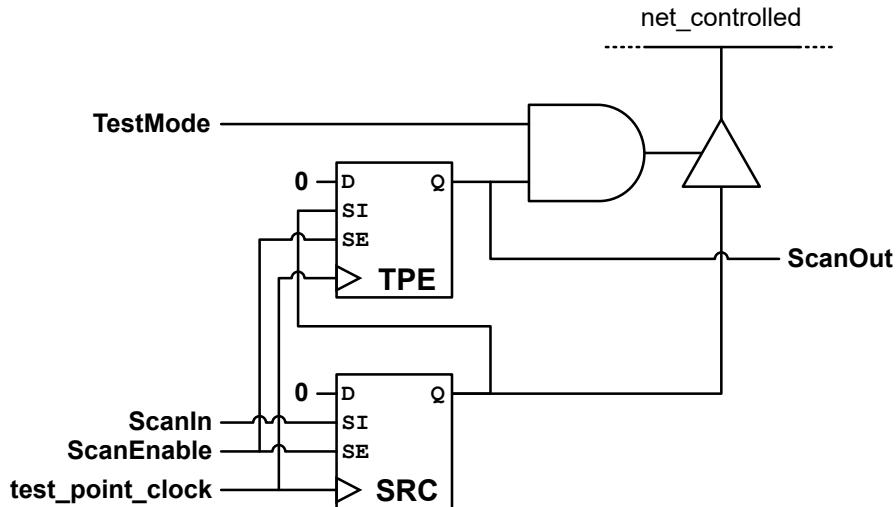


Figure 523 Example of a control_z01 Test Point

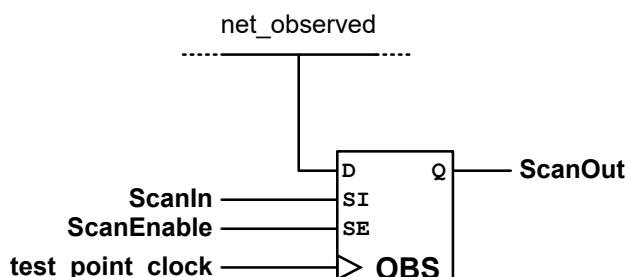


Observe Test Points

The `observe` test point type is typically inserted at hard-to-observe signals 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 sink signal to be observed. See Figure 524.

Figure 524 Example of an observe Test Point



Test Point Signals

Test points use source, sink, test point enable, and control signals as shown in Table 67.

Table 67 Test Point Signal Types

Test point type	Source signal	Enable signal	Sink signal	Control signal
force_0, force_1				X
force_01	X			X
force_z0, force_z1				X
force_z01	X			X
control_0, control_1		X		X
control_01	X	X		X
control_z0, control_z1		X		X
control_z01	X	X		X
observe			X	X ¹³

The control signal is the TestMode signal that activates the test point logic.

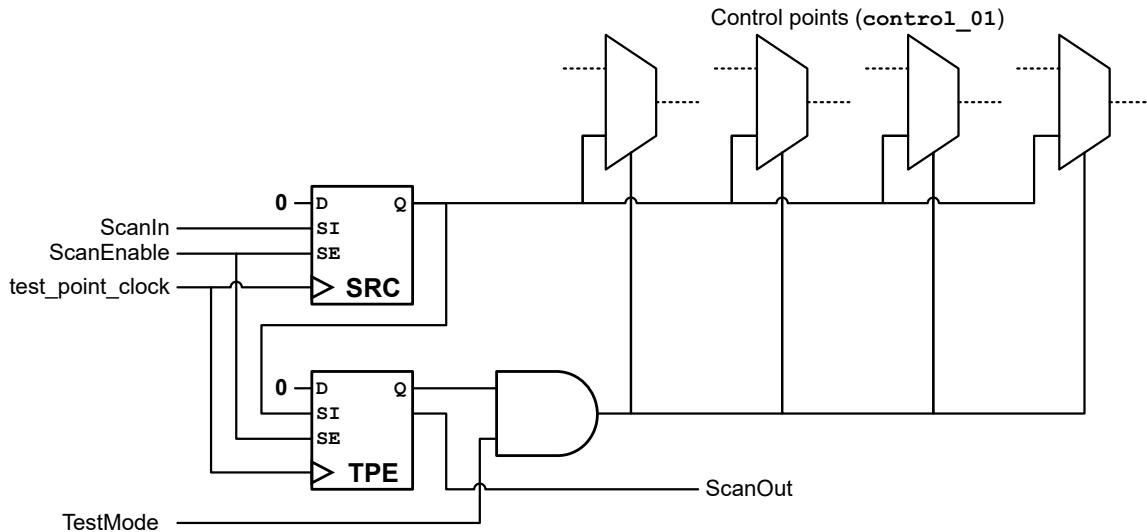
Sharing Test Point Scan Cells

To reduce the area requirements of test point logic, DFT Compiler allows you to share the test point enable, source signal, and sink signal scan registers with multiple test points.

You can share the same test point enable or source signal register with multiple control or force test points. No additional logic gates are required; the scan register outputs are tied to multiple test point logic gates. [Figure 525](#) shows the logic for multiple `control_01` test points that share the same scan registers.

13. Test-mode signal used only if low-power XOR observability tree is enabled

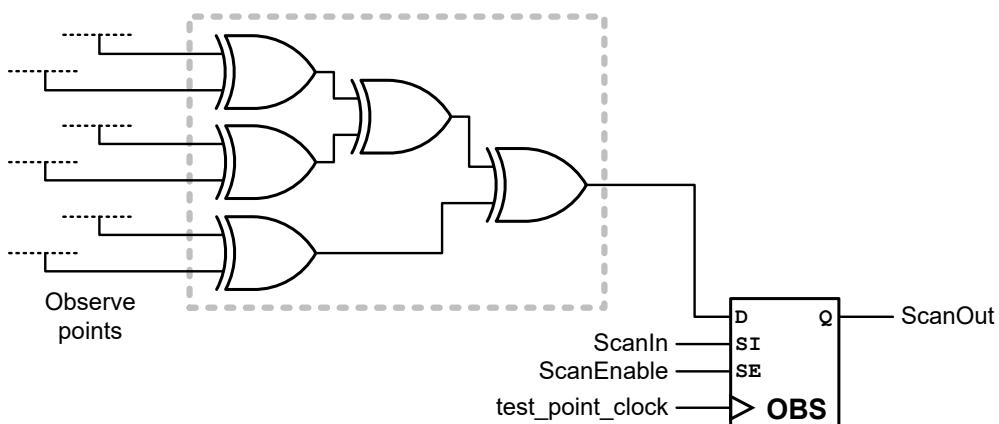
Figure 525 Shared Scan Registers for Multiple control_01 Test Points



Test point enable and source signal scan registers are not shared between different test point types.

You can share the same observe sink signal scan register with more than one `observe` test point. DFT Compiler builds an observability XOR tree which collapses multiple observed signals down to a single sink signal connected to the data input of the shared sink signal scan register. See [Figure 526](#).

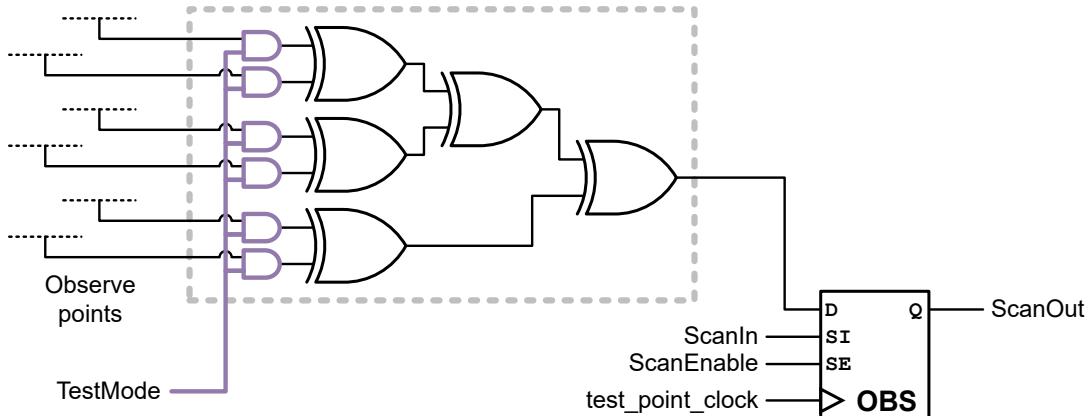
Figure 526 XOR Observability Tree For Multiple observe Test Points



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 fanout path through the XOR observability tree toggles. This toggling results in unnecessary power losses.

To avoid such losses in power, you can create a low-power observability tree for a shared observe scan register. The observe point signals are gated with a 2-input AND gate, with a test-mode signal used as the gating signal. See [Figure 527](#).

Figure 527 Low-Power XOR Observability Tree For Multiple observe Test Points



If a test-mode signal is defined, DFT Compiler uses it for the low-power gating signal. Otherwise, DFT Compiler creates a new test-mode signal that is used only for low-power observability gating.

Automatically Inserted Test Points (Legacy)

The tool can automatically insert test points to improve the testability of the design. You can optionally specify requirements for test point insertion, such as the maximum number of test points or the maximum additional area overhead. During DFT insertion, the tool inserts the optimal set of test points that meets the requirements.

The following topics describe how to configure automatic test point insertion:

- [Enabling Automatic Test Point Insertion](#)
- [Configuring Pattern Reduction and Testability Test Point Insertion](#)
- [Script Example](#)

Enabling Automatic Test Point Insertion

To enable automatic test point insertion, you must first issue the following command before pre-DFT DRC:

```
dc_shell> set_dft_configuration -test_points enable
```

Note:

A DFTMAX or TestMAX DFT license is required to use the automatic test point insertion feature.

After you have enabled automatic test point insertion, you can enable and configure one or more automatic test point targets with the `set_test_point_configuration` command, as described in the following topics. To enable multiple test point targets, issue a separate configuration command for each target.

Configuring Pattern Reduction and Testability Test Point Insertion

You can use the pattern reduction and testability automatic test point insertion targets to improve the testability of hard-to-test logic in your design. They work as follows:

- `pattern_reduction` – Enables only observe points
 This mode reduces the pattern count needed to achieve a given amount of test coverage. Observe points increase the loading along the observed path, but do not directly increase the logic depth. This mode has less impact on timing.
- `testability` – Enables both control and observe points
 This mode improves the testability of the design by increasing the controllability of hard-to-test logic. Keep in mind that control points insert logic along the path being controlled. Although gate-level optimization can combine the control points with the surrounding logic, there might be some impact on timing.

During pre-DFT DRC, performed by the `dft_drc` command, the tool analyzes the design to determine the optimal set of test points. During DFT insertion, performed by the `insert_dft` command, the tool inserts the test points into the design. Any needed dedicated clock or test mode signals are created.

To enable and configure the testability or pattern reduction targets of automatic test point insertion, use the `set_test_point_configuration` command as follows:

```
set_test_point_configuration
    -target pattern_reduction | testability
    [-control_signal control_name]
    [-clock_signal clock_name]
    [-clock_type dominant | dedicated]
    [-max_control_points n]
    [-max_observe_points n]
    [-test_points_per_scan_cell n]
    [-power_saving enable | disable]
    [-max_additional_logic_area n]
```

The `-target` option specifies which automatic test point insertion target to enable and is a required option. The `pattern_reduction` and `testability` targets are mutually exclusive.

You can use the following additional options, along with the `-target` option, to configure pattern reduction or testability test point insertion:

- By default, control points use any available TestMode port previously defined with the `set_dft_signal` command. To specify the TestMode signal that should activate the test points, use the following option:

```
dc_shell> set_test_point_configuration ... -control_signal pin_port
```

The specified control signal must be defined as a `TestMode` signal type with the `set_dft_signal` command.

- By default, the `insert_dft` command uses the dominant clock, which is the clock that clocks the most sequential elements in the design, to clock the inserted test point scan registers. In an on-chip clocking (OCC) flow, it chooses an OCC clock instead.

To specify that a dedicated test point clock signal should be used, use the following option:

```
dc_shell> set_test_point_configuration ... -clock_type dedicated
```

This option causes a new dedicated test point clock signal, `tpclk`, to be created.

To specify the test clock signal that should clock the test point scan registers, use the `-clock_signal` option:

```
dc_shell> set_test_point_configuration ... \
           -clock_type dedicated \
           -clock_signal clock_name
```

DFT Compiler supports the following clock name specifications:

- You can specify the name of a scan clock signal, defined as a `ScanClock` signal type with the `set_dft_signal` command.
- In a DFT-inserted OCC controller flow, you can specify the name of a PLL output pin. In this case, DFT Compiler maps the test point clock to the output pin of the corresponding OCC controller during DFT insertion.
- In a user-defined OCC controller flow, you can directly specify the name of an output pin of an existing OCC controller.

For more information about OCC controller flows, see [Chapter 12, On-Chip Clocking Support](#).

- By default, automatic test point insertion is limited to a maximum of 1000 control points. To specify a different limit, use the following option:

```
dc_shell> set_test_point_configuration ... -max_control_points n
```

This option is only valid in testability mode; it is ignored in pattern reduction mode.

- By default, automatic test point insertion is limited to a maximum of 1000 observe points. To specify a different limit, use the following option:

```
dc_shell> set_test_point_configuration ... -max_observe_points n
```

- By default, each source, sink, or enable scan register can be shared by up to eight test points. To specify the maximum number of test points that can share a single source, sink, or enable scan register, use the following option:

```
dc_shell> set_test_point_configuration ... \
           -test_points_per_scan_cell n
```

For more information about sharing scan cells, see [Sharing Test Point Scan Cells on page 1091](#).

- To insert power-saving AND gates at the top of the XOR observability trees to avoid excess switching power consumption during scan shift, use the following option:

```
dc_shell> set_test_point_configuration ... -power_saving enable
```

For more information about power-saving logic, see [Sharing Test Point Scan Cells on page 1091](#).

- To apply an area limit to the inserted test point logic, use the following option:

```
dc_shell> set_test_point_configuration ... \
           -max_additional_logic_area P
```

The value P is the percentage of the total design area that can be consumed by the test point logic and must be a value between 1 and 50. Low-power observability logic is included in the test point area value. If specified, the area limit applies in addition to the test point limit.

Script Example

The following script inserts testability test points, using a test-mode control signal named TM_TESTPOINTS.

```
# define DFT signals
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing [list 45 55]
set_dft_signal -view spec -type TestMode -port TM_TESTPOINTS

# enable automatic test point insertion
set_dft_configuration -test_points enable
```

```
# enable and configure testability test points
set_test_point_configuration \
    -target testability \
    -control_signal TM_TESTPOINTS

# preview test points
preview_dft -test_points all

# insert DFT logic
insert_dft
```

User-Defined Test Points (Legacy)

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 topics describe how to implement user-defined test points:

- [Configuring User-Defined Test Points](#)
- [User-Defined Test Points Example](#)

Configuring User-Defined Test Points

You can use the `set_test_point_element` command to specify the location and type of user-defined test points to insert in the design during DFT insertion, as well as other aspects of test point construction. User-defined test points can be defined at leaf pins, hierarchy pins, and ports. These test points are then inserted during the `insert_dft` command.

To define a user-defined test point, specify the test point type and list of signal pins or ports to be forced, controlled, or observed:

```
dc_shell> set_test_point_element -type test_point_type signal_list
```

For a list of test point types and their descriptions, see [Test Point Types on page 1085](#).

By default, any needed source, sink, and enable signals are supplied by scan registers inserted by the `insert_dft` command. Each scan register is shared by up to eight source, sink, or enable test point signals. For shared source and enable signal registers, the same scan-selected signal value is used by all shared test points. For shared sink signal registers, an XOR observability tree is used to combine the observed signals for capture by the sink register.

You can use the following options to control user-defined test point insertion:

- By default, force and control points use any available TestMode port previously defined with the `set_dft_signal` command. To specify the TestMode or ScanEnable signal that should activate the test points, use the following option:

```
dc_shell> set_test_point_element -control_signal pin_port ...
```

The specified control signal must be defined as a `TestMode` or `ScanEnable` signal type with the `set_dft_signal` command.

- By default, the `insert_dft` command creates a new clock signal, `tpclk`, to clock any inserted test point scan registers, even when test clocks have been defined. To specify the test clock signal that should clock the scan registers, use the `-clock_signal` option:

```
dc_shell> set_test_point_element -clock_signal clock_name ...
```

DFT Compiler supports the following clock name specifications:

- You can specify the name of a scan clock signal, defined as a `ScanClock` signal type with the `set_dft_signal` command.
- In a DFT-inserted OCC controller flow, you can specify the name of a PLL output pin. In this case, DFT Compiler maps the test point clock to the output pin of the corresponding OCC controller during DFT insertion.
- In a user-defined OCC controller flow, you can directly specify the name of an output pin of an existing OCC controller.

For more information about OCC controller flows, see [Chapter 12, On-Chip Clocking Support](#).

- To specify the maximum number of test points that can share a single source, sink, or enable register, use the following options:

```
dc_shell> set_test_point_element \
           -test_points_per_source_or_sink n ...
```

```
dc_shell> set_test_point_element \
           -test_points_per_test_point_enable n ...
```

Source, sink, or enable registers are created as needed, according to the specified sharing limit and the number of test point signal pins provided.

- To specify that the source, sink, or enable signals should come from primary input and output ports instead of scan registers, use the following options:

```
dc_shell> set_test_point_element -scan_source_or_sink false ...
```

```
dc_shell> set_test_point_element -scan_test_point_enable false ...
```

The same source, sink, and enable signal sharing is performed, except that primary input and output ports are created instead of scan registers.

- To specify that a specific user-supplied source, sink, or enable signal be used for a given test point definition, use the following options:

```
dc_shell> set_test_point_element \
           -source_or_sink source_or_sink_name ...
```

```
dc_shell> set_test_point_element \
           -test_point_enable test_point_enable_name ...
```

When a user-supplied source, sink, or enable signal is specified, it is used for all test points in that `set_test_point_element` command, and the previously described sharing limit and scan register options do not apply.

- To insert power-saving AND gates at the top of an XOR observability tree to avoid excess switching power consumption during scan shift, use the following option:

```
dc_shell> set_test_point_element -power_saving enable ...
```

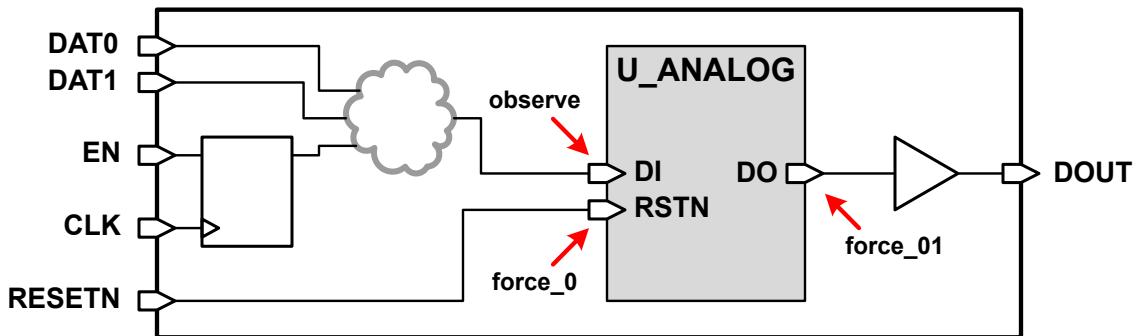
Each `set_test_point_element` command describes a unique test point element definition. Sharing is not performed between test point element definitions. If test points within a limited geographic region should share the same source, sink, or enable signals, they should all be provided in a single `set_test_point_element` command. If test points across a wide geographic region should not share signals to avoid routing congestion, they should be broken up into localized groups and specified with separate `set_test_point_element` commands.

After specifying test point definitions with the `set_test_point_element` command, you can report them with the `report_test_point_element` command, or remove them before DFT insertion with the `remove_test_point_element` command. For more information about these commands, see the man pages.

User-Defined Test Points Example

Consider the simple design shown in [Figure 528](#) and the corresponding example using the user-defined test points flow shown in [Example 167 on page 1101](#).

Figure 528 Design Example for User-Defined Test Points



In this design example, some control signals are combined using a cloud of combinational logic, then fed to the DI input of an analog block. The DO output of the analog passes through an output drive buffer so that it can drive a possible long route outside the block. Because this analog block is untestable, the logic fanin to the DI input cannot be observed, and the logic fanout from the DO output cannot be controlled.

To improve the testability of the logic around this analog block, the following user-defined test points can be specified:

- Insert an `observe` test point at the DI input of the analog block to provide observability of the data logic cone:

```
set_test_point_element -type observe U_ANALOG/DI
```

- Insert a `force_0` test point at the RSTN pin of the analog block to hold the block in a quiet, low-power reset state during the test program:

```
set_test_point_element -type force_0 U_ANALOG/RSTN
```

- Insert a `force_01` test point at the DO output of the analog block to provide controllability of the downstream logic:

```
set_test_point_element -type force_01 U_ANALOG/DO
```

A force test point is used at the DO output pin instead of a control test point. The output of the analog block is always unknown, and there is no reason to selectively allow this unknown value to propagate downstream. This force test point is placed at the analog block DO output instead of the DOUT output port so that faults at the drive buffer can be detected. If the test point was placed at the DOUT output port instead, faults between the analog block and the output port could not be detected.

The existing clock CLK is used to clock the test point scan registers. A new TESTMODE port is created to enable the test points.

Example 167 Example of a User-Defined Test Point Flow

```

# Read in the design and synthesize it
read_file -format verilog ./rtl/design.v
current_design TEST
link
read_sdc TEST.sdc
compile -scan

# Define the clock, reset, test-mode ports
set_dft_signal -view existing_dft -type ScanClock \
    -port CLK -timing {45 55}
set_dft_signal -view existing_dft -type Reset \
    -port RST -active_state 0
set_dft_signal -view spec -type TestMode \
    -port TESTMODE -active_state 1
set_scan_configuration -chain_count 10

# Provide the UDTP specifications
set_test_point_element -type observe U_ANALOG/DI \
    -clock_signal CLK -control_signal TESTMODE

set_test_point_element -type force_0 U_ANALOG/RSTN \
    -clock_signal CLK -control_signal TESTMODE

set_test_point_element -type force_01 U_ANALOG/DO \
    -clock_signal CLK -control_signal TESTMODE

# Run pre-DFT DRC
create_test_protocol
dft_drc -verbose

# Preview and insert DFT
preview_dft -show all -test_points all
insert_dft

# Run post-DFT DRC
dft_drc -verbose
report_scan_path

# Write out the netlist
write -hierarchy -format ddc -output TEST_udtp_scan.ddc
change_names -rules verilog
write -hierarchy -format verilog -output TEST_udtp_scan.v

```

Previewing the Test Point Logic

To preview the test point logic that the tool will implement according to your specifications, use the following command:

```
dc_shell> preview_dft -test_points all
```

This command reports the following information:

- Test point locations
- Instance names of inserted test point flip-flops
- Clocks used by inserted test point flip-flops
- Low-power observability tree status

You can use other options of the `preview_dft` command with the `-test_points` option.

Inserting the Test Point Logic

After you define the test point insertion configuration, the `insert_dft` command inserts the test point logic. Test point scan registers are placed in the lowest level of hierarchy shared by all test points for that register.

The following additional signals are created, depending on the test configuration:

- A new test point clock signal, if a test point clock is not defined
- A new test-mode signal for force, control, and observe test points, if a test-mode signal is not defined

C

Legacy RTL Design Rule Checking

This appendix documents the legacy RTL design rule checking (DRC) functionality provided in the synthesis tool. For best results, use the newer functionality provided by the TestMAX Advisor or SpyGlass DFT ADV tools.

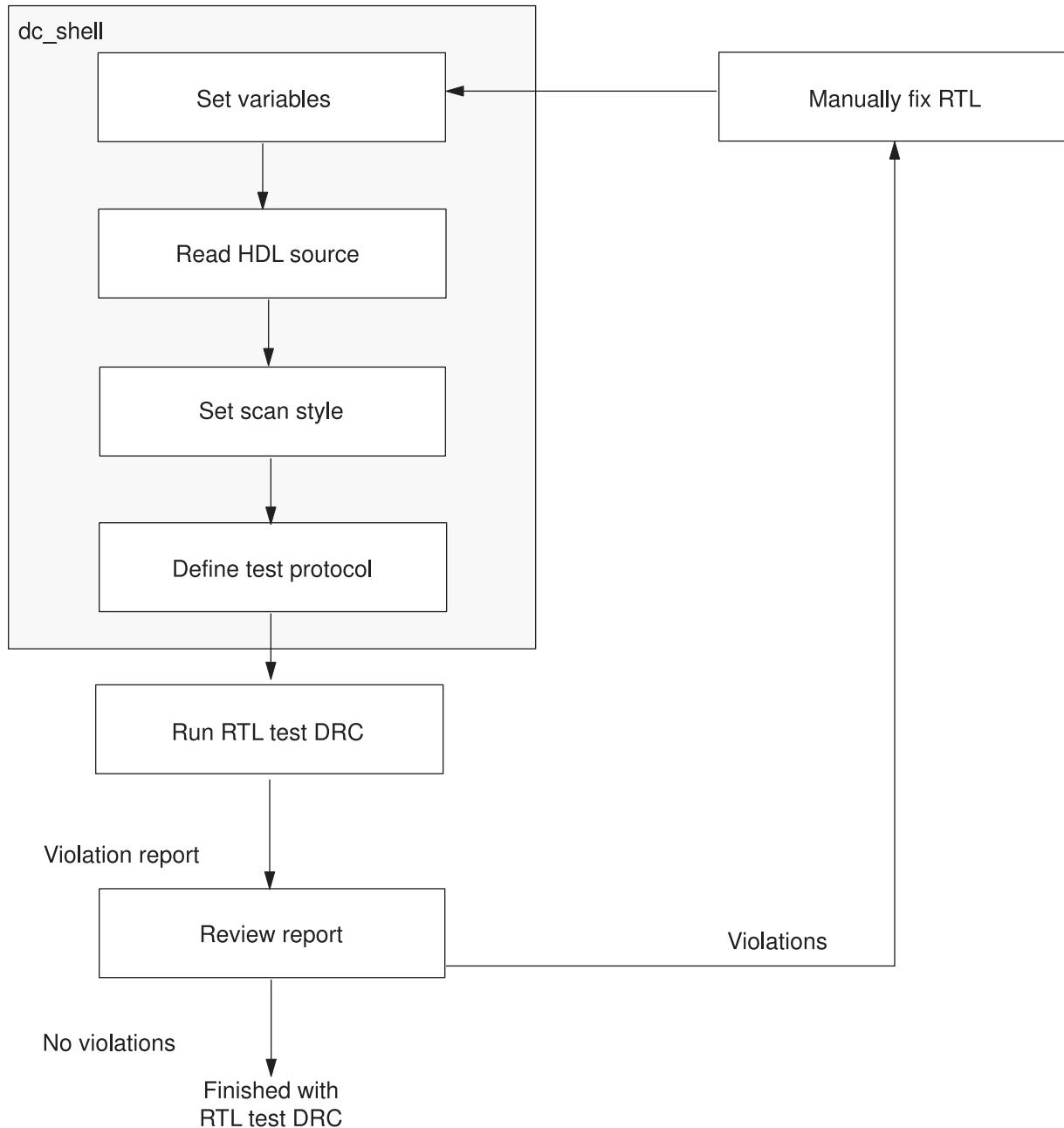
This appendix includes the following topics:

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

Understanding the Flow

[Figure 529](#) shows a typical RTL test DRC flow.

Figure 529 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 hlin_enable_rtldrc_info true
```

2. Make sure you define the list of searched logic 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 topic covers the following topics 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 should 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 a STIL initialization protocol:

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

[Example 168](#) 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 168 Complete Protocol File (init.spf)

```
STIL 1.0 {
    Design P2000.9;
}
Header {
```

Appendix C: Legacy RTL Design Rule Checking

Generating a Test Protocol

```

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
}

```

Appendix C: Legacy RTL Design Rule Checking

Generating a Test Protocol

```

        "_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 the `scan_style` value:

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

- scan_enabled_lssd
- combinational

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

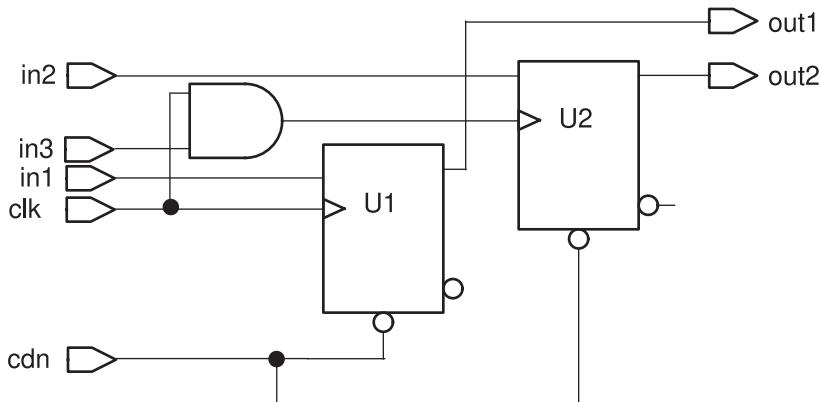
Design Examples

This topic 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 530](#) shows a schematic and the Verilog code for a simple RTL design that needs a test protocol.

Figure 530 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 <= 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 design example:

```

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

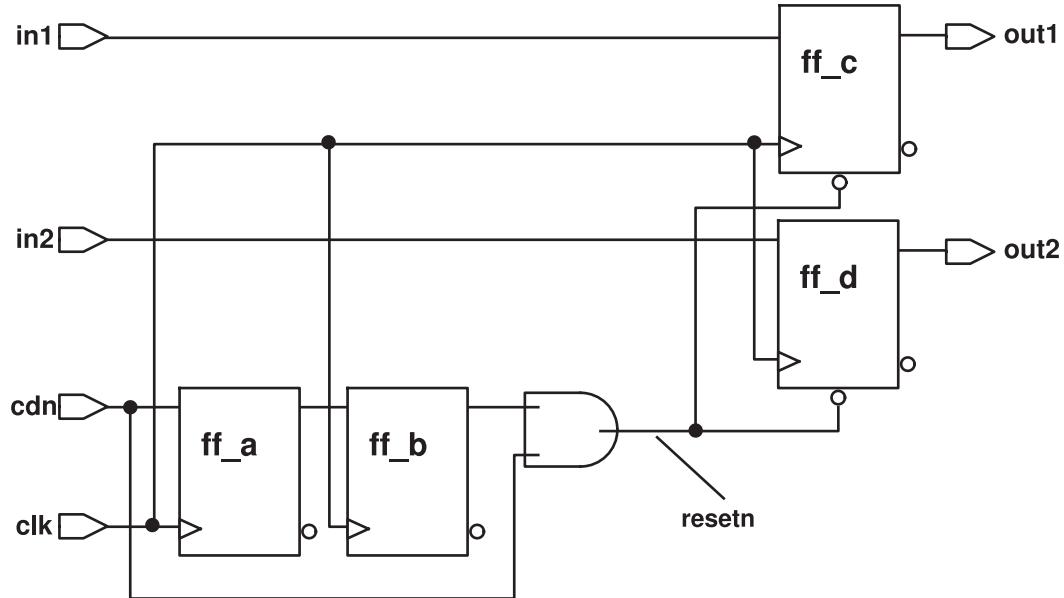
dc_shell> write_test_protocol -output design.spf

```

Test Protocol Example 2

[Figure 531](#) shows a schematic and the corresponding Verilog code for an RTL design that requires initialization.

Figure 531 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 169](#).

Example 169 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 { "cdn"=1; "clk"=0; }
    }
}
```

```
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 at the shell prompt, 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 topic 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.

This topic covers the following:

- [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 the TestMAX ATPG tool.
- 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 1105](#).

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 warning messages:

Warning: Clock input *I* of DFF *S* was not controlled. (D1-N)

Warning: Clock input *I* of DLAT *S* was not controlled. (D4-N)

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 warning messages:

Warning: Set input *I* of DFF *S* was not controlled. (D2-N)

Warning: Reset input *I* of DFF *S* was not controlled. (D3-N)

Warning: Set input *I* of DLAT *S* was not controlled. (D5-N)

Warning: Reset input *I* of DLAT *S* was not controlled. (D6-N)

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 topic, ATPG might generate vectors that fail functional simulation or fail on the tester, although in the TestMAX ATPG tool you would also have to override the TestMAX ATPG default settings.

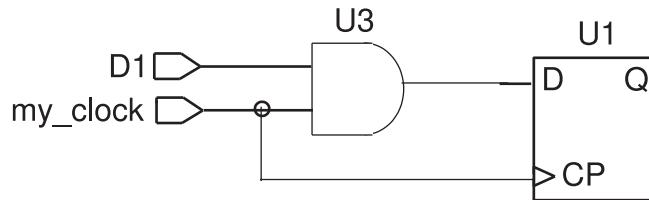
The violations are described in the following topics:

- [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 532](#), ATPG tools cannot determine the captured value. Modify the logic leading to the datapath to eliminate dependency on the clock.

Figure 532 Clock Signal Used As Data Input



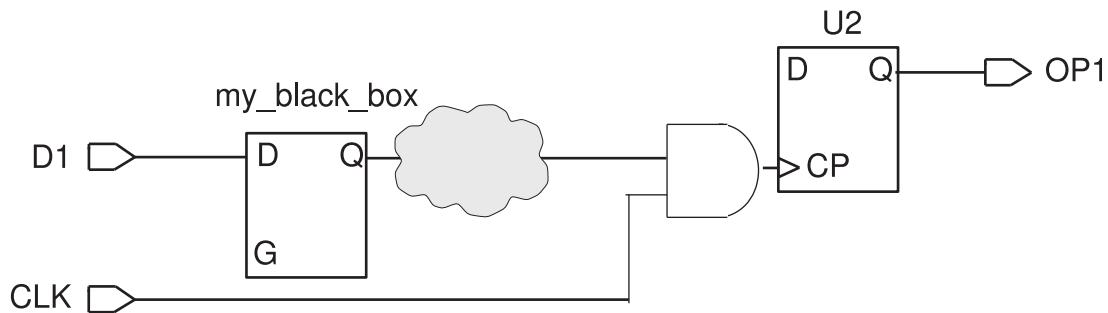
If the clock and data input to a register are interdependent, you might get the following warning message:

Warning: Clock C connects to clock and data inputs I1/I2 of DFF S.
(D11-N)

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 533](#).

Figure 533 Black Box Feeds Clock Input



See Also

- [Black Boxes on page 1124](#) for more information about how black boxes affect testability

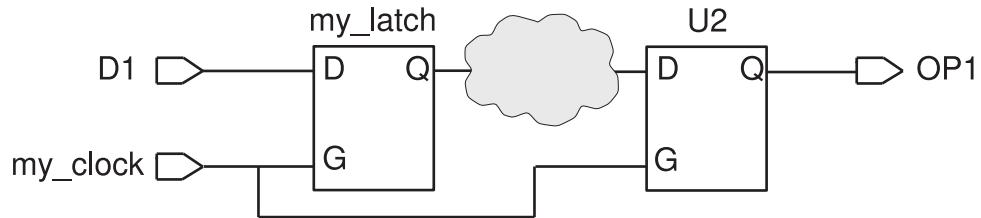
Source Register Launch Before Destination Register Capture

This topic 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 534](#). 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 534 Latch-Based Circuit With Source Register Launch Before Destination Register Capture



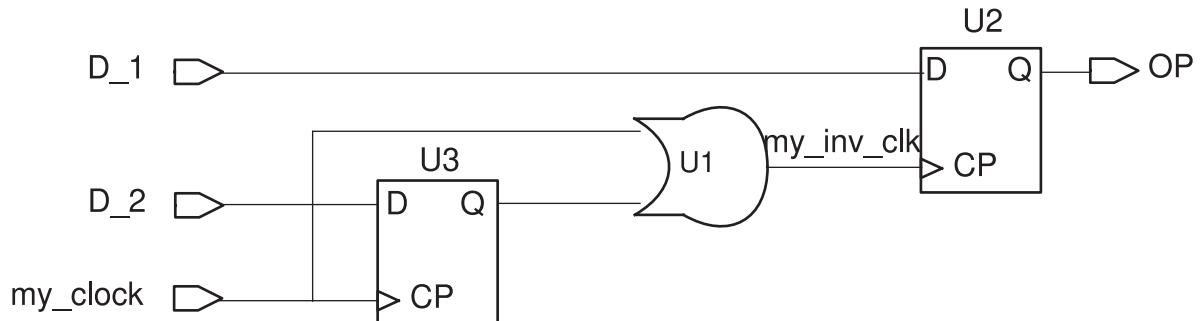
If multiple latches are enabled so that the latches feed through capture data, you get the following warning message:

Warning: Clock C cannot capture data with other clocks off. (D8-N)

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 535](#).

Figure 535 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 warning message:

Warning: Clock input I of DFF S was not controlled. (D1-N)

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 warning messages is issued:

Warning: Bus gate N failed contention ability check for drivers $G1$ and $G2$. (D20-N)

Warning: Bus gate N failed Z state ability check. (D21-N)

Warning: Wire gate N failed contention ability check for drivers $G1$ and $G2$. (D22-N)

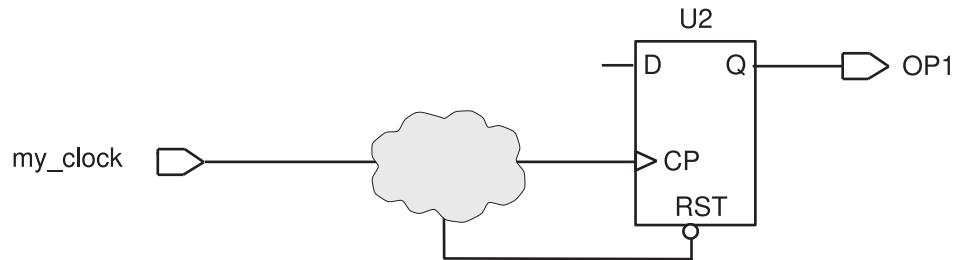
Clock Feeding Multiple Register Inputs

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 536](#) shows an example of this violation.

Figure 536 Clock Signal Feeds Register Clock Pin and Asynchronous Reset



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

Warning: D12 Clock C connects to clock/set/reset inputs (G1 / G2) of DFF I. (D12-N)

Violations That Reduce Fault Coverage

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

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

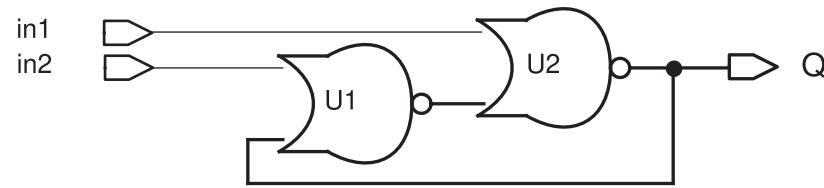
Combinational Feedback Loops

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 537](#) shows this type of loop.

Figure 537 Highlighted Combinational Feedback Loop



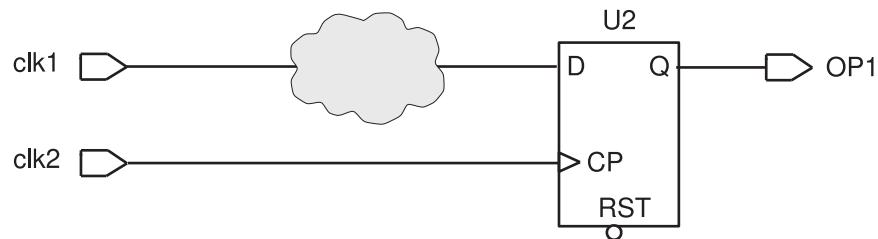
If your design contains a sensitizable feedback loop, you get the following warning message:

Warning: Feedback path network X is sensitizable through source gate G. (D23-N)

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 538](#), 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 538 Clock Interacting With Register Input



If a clock affects the data input of a register, you might get the following warning message:

Warning: Clock C connects to data input of DFF S. (D10-N)

Multiple Clocks That Feed Into Latches and Flip-Flops

The following topics describe the types of clock-gating configurations that can reduce fault coverage:

- [Latch Requires Multiple Clocks to Capture Data](#)
- [Latches Are Not Transparent](#)

See Also

- [Violations That Prevent Scan Insertion on page 1115](#) for more information about other clock-gating configurations that prevent scan insertion
- [Violations That Prevent Data Capture on page 1116](#) for more information about other clock-gating configurations that prevent data capture

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.

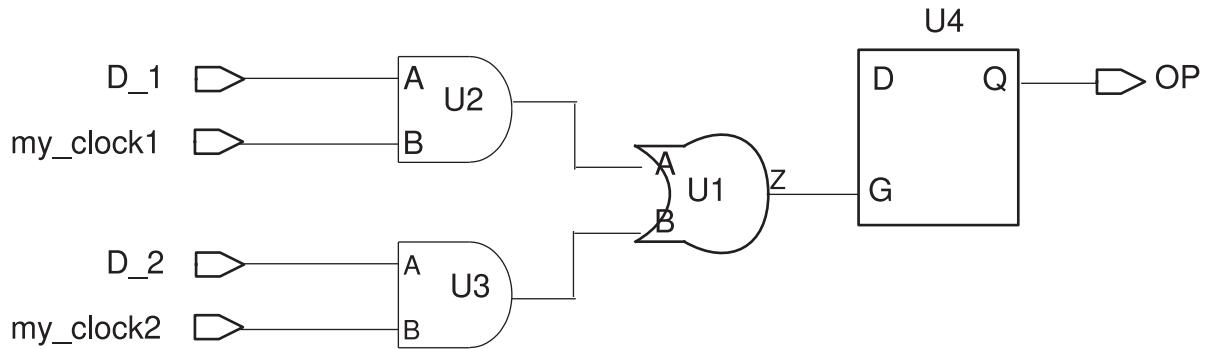
If your design has an OR gate with clock and data inputs, the output clock of the OR gate has extra pulses that depend on the data input. If your design has an AND gate with more than one clock input, the output of the AND gate never generates a clock pulse. Both of these cases are violations, and DFT 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 539](#).

Figure 539 Valid Latch Clock Gating



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

Warning: Clock C cannot capture data with other clocks off. (D8-N)

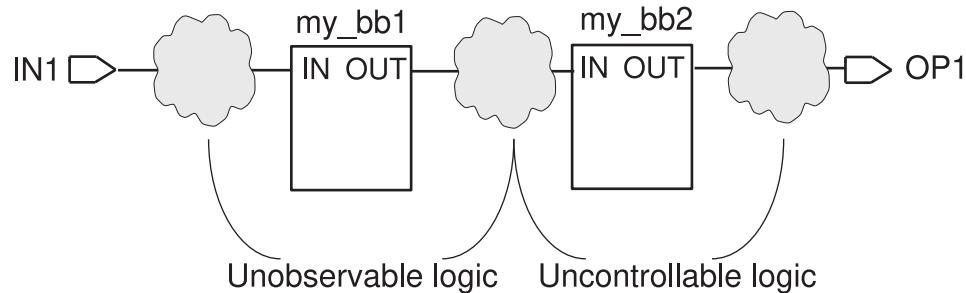
Latches Are Not Transparent

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 540](#) shows an example.

Figure 540 Effect of Black Boxes on Surrounding Logic



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

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

Limitations

Note the following limitations:

- The `set_svf` command is not supported in the RTL test DRC flow. You should comment out any `dft_drc` command that performs test DRC checking on elaborated RTL before you perform design synthesis, which generates the verification setup file.
- The `compile_ultra -gate_clock -scan` command is not supported in the RTL test DRC flow. When the `create_test_protocol` command is run on the elaborated RTL, subsequent `compile_ultra -gate_clock -scan` commands might not properly incorporate clock-gating cells into the scan chains. You should comment out any `create_test_protocol` commands performed on elaborated RTL before you perform design synthesis with this command.

Glossary

Associated internal clocks

A set of one or more user-defined internal clocks in a clock network, each of which is the root of a skew subdomain within its parent clock network.

ATPG

Automated Test Pattern Generation, the generation of scan data sequences used for scan testing, with the goal of achieving as much test coverage as possible using the smallest possible number of patterns. The test patterns contain nonfunctional data selected to detect faults on nets in the design.

ATE

Automated Test Equipment, industrial equipment used to test semiconductor devices by applying input stimuli, observing the device response, and comparing it against the expected response.

BIST

Built-in self-test, design-for-test logic where both the scan data generation and the scan data comparison logic are included in the design.

BSD

Boundary Scan Design, which refers to test logic that implements IEEE Std 1149.1.

Burn-in mode

A mode that continuously runs autonomous self-test. The scan and capture activity stresses the tested logic and causes continuous power draw during self-test. Burn-in operation can be configured to stop or continue if self-test fails.

CDR

Core data register, an optional register in an IEEE 1500 implementation that can be loaded with a value after loading a corresponding instruction into the wrapper instruction register (WIR).

Clock chain

A special scan chain segment associated with an on-chip clocking (OCC) controller whose scanned-in values control the pulse sequence of a controlled clock.

Clock gating

A method of reducing power by shutting off clocks to circuits that are not being used.

Codec

The combination of the decompressor and compressor in a compressed scan flow. A single design can have multiple codecs, but each codec consists of its own decompressor/compressor pair.

Compressed scan

A scan methodology that uses more scan chains (called compressed scan chains) than scan-in/scan-out pairs. A decompressor decompresses the scan-in data to drive the greater number of scan chains. A compressor compresses the scan chain data to drive the lesser number of scan-outs. The combination of the decompressor and compressor wrapped around the scan chains is called the *codec*.

Compressed scan chains

In a compressed scan flow, the scan chains that are driven by the decompressor and drive the compressor. There are a greater number of compressed scan chains than scan-in/scan-out pairs.

Compressor

In a compressed scan flow, the part of a codec that compresses the scan chain data to drive the lesser number of scan-outs.

Core

A design block that is DFT-inserted and has CTL model information about the inserted DFT structures. Cores are used in hierarchical scan synthesis flows.

Core wrapping

See *wrapped core*.

CTL model

Core Test Language model, describes the characteristics of a DFT-inserted design in STIL (IEEE Std 1450) format. This model includes information such as: DFT signals, scan chains, test clocks and clock control, and test modes.

Decompressor

In a compressed scan flow, the part of a codec that decompresses the scan-in data to drive the greater number of scan chains.

DFT

Design-for-Test, pertains to logic that helps the testability of a design.

DFTMAX scan compression

Scan compression implemented by the TestMAX DFT tool that uses combinational codecs to yield high scan compression ratios.

DFTMAX Ultra scan compression

Scan compression implemented by DFTMAX Ultra compression that uses streaming (sequential) codecs to yield very high scan compression ratios, even down to a single scan-in/scan-out pair.

DFT partition

A DFT specification that allows the sequential cells in a design to be separated into multiple independent partitions for the purpose of DFT insertion.

DRC

Design Rule Checking, checking a design against a rule set that ensures good testability and reporting any violations of those rules.

EXTEST mode

Outward-facing test mode used to test logic external to (outside) a core, independent of the logic inside the core. It uses the wrapper chain to drive scan-controllable values at the output wrapper cells and capture the values at the input wrapper cells.

FSM

Finite state machine, a logic construct that moves through states (like navigating elements in a flowchart diagram) to implement a certain logic behavior.

Hierarchical DFT insertion

Refers to a flow that performs DFT insertion in a lower level block (known as a *core*), then incorporates that block's scan structures into a higher level.

This term pertains to how DFT insertion is performed in the tool flow. Do not confuse this term with *hierarchical testing*, which pertains to how manufacturing test is run on the ATE.

Hierarchical testing

Refers to the process of testing different hierarchy levels of the design independently. Wrapped cores are often used to enable hierarchical testing.

This term pertains to how manufacturing test is run on the ATE. Do not confuse this term with *hierarchical DFT insertion*, which pertains to how DFT insertion is performed.

HSS

Hierarchical scan synthesis. See *hierarchical DFT insertion*.

Internal chains

The scan chains inside a block (uncompressed or compressed), as opposed to other kinds of chains, such as boundary scan chains or core-wrapping chains.

Internal clock

A scan clock, defined on an internal pin in the design, that is the root driver of a *skew subdomain*. Internal clocks can be automatically determined by the tool at multi-input gate outputs or specified manually at associated internal clock pins within the clock network.

INTEST mode

Inward-facing test mode used to test logic internal to (inside) a core, independent of the logic outside the core. It uses the wrapper chain to drive scan-controllable values at the input wrapper cells and capture the resulting values at the output wrapper cells.

Inward-facing test mode

See INTEST.

Leading edge

The first edge of a clock waveform definition. It is a rising edge for a return-to-zero clock, and it is a falling edge for a return-to-one clock.

LFSR

Linear feedback shift register, a shift register whose next data word is a linear XOR function of its current data word. The PRPG uses an LFSR to generate pseudorandom data.

MISR

Multiple-input signature register, a recirculating shift register that XORs scan-captured data values into the loop. After capturing all values, the MISR contains a signature value for that test.

OCC controller

On-chip clocking controller, a DFT design structure that controls a free-running on-chip clocking source (such as a PLL output clock) in test mode.

Outward-facing test mode

See EXTEST.

Pad cell

A special cell at the chip boundaries that allows communication with other integrated circuits outside the chip, as opposed to an internal core cell, which makes up the core of an integrated circuit.

Pipelined scan data

A feature that inserts additional scan registers, called *pipeline registers*, at the scan-in and scan-out ports of the design to accommodate long wires between the scan chain input and the first flip-flop and between the last flip-flop and the scan chain output.

Pipeline registers inserted at the scan-in and scan-out ports are called *head* and *tail* pipeline registers, respectively.

Pipelined scan enable

A feature that adds a pipeline register to the scan-enable signal. It is used for transition delay testing by making use of launch-on-extra-shift (LOES). This method of transition delay testing requires additional circuitry to manipulate the scan-enable signals.

PLL

Phase-locked loop, an analog design block that creates a stable on-chip latency-adjusted clock from a free-running (and possibly less stable) input clock.

Pre-DFT DRC

DRC checking run before DFT insertion, evaluates the readiness of the design for DFT insertion. Certain types of pre-DFT DRC violations will result in scan cells being excluded from scan chains.

PRPG

Pseudo-random pattern generator, uses an LSFR and an XOR phase shifter to generate a stream of pseudorandom data values that have the appearance of random values, but are actually a function of a seed value.

Post-DFT DRC

DRC checking run after DFT insertion, evaluates the implemented DFT functionality of the design for correct operation.

Return-to-one clock

A clock whose value is normally high, with an active-low pulse during the clock period.

Return-to-zero clock

A clock whose value is normally low, with an active-high pulse during the clock period.

Scan cell

A sequential cell that has both functional and scan-shift modes of operation.

Scan compression

See *compressed scan*.

SCANDEF

A DEF file that uses a set of scan-specific constructs to describe how a design's scan chains can be reordered and repartitioned by a layout tool.

Scan group

A group of scan cells to be kept together in a scan chain.

Seed value

The initial value loaded into a PRPG. Different seed values result in different pseudorandom value sequences.

Shadow wrapper

A "wrapper" around an untestable block or macrocell that allows surrounding logic to be tested. Known values are forced at the outputs, and to improve coverage, the values at the inputs are captured. It can be easily inserted using automatic test point insertion.

Shared codec I/O

A feature that allows the scan-in and scan-out connections of DFTMAX cores or codecs to be shared to reduce scan I/O requirements.

Shift register

A sequence of sequential cells in the design whose functional operation is to shift bits through the register like a scan chain. Shift registers only need their first (head) element scan-replaced to be stitched into a scan chain.

Skew subdomain

Part of a parent clock network that is considered to have different clock skew characteristics than the rest of the clock network. Lock-up latches are inserted

whenever a scan chain crosses a skew subdomain boundary. A skew subdomain is driven by an *internal clock* pin.

SPF

STIL Protocol File, a file written out by DFT Compiler to describe DFT aspects of the design, such as: test ports, test clocks, primary input constraints, scan chains, codecs, and test modes. It is used by TestMAX ATPG (or other ATPG tool) for DRC and ATPG.

STIL

Standard Test Interface Language, documented in IEEE Std 1450, which is a language used to describe the DFT capabilities of a design. It is a standard for simplifying the number of test vector formats that automated test equipment (ATE) vendors and computer-aided engineering (CAE) tool vendors must support.

Standard scan

A scan methodology in which there is a one-to-one relationship between each scan-in/scan-out pair and each scan chain.

TCM

Test control module, a DFT design structure that selects the current test mode. The input is a vector of test-mode selection signals that can have binary, one-hot, or user-defined encodings. The output is a set of decoded one-hot enable signals, one for each test mode encoding.

TMCDR

Test-mode core data register, a special core data register (CDR) in an IEEE 1500 implementation that takes the place of traditional port-driven test-mode signals.

Trailing edge

The last edge of a clock waveform definition. It is a falling edge for a return-to-zero clock, and it is a rising edge for a return-to-one clock.

Weighted capture groups

An OCC-based capture method that uses comparator logic to enable one capture group in each pattern, based on user-specified probabilities.

UPF

Unified Power Format, a standard set of commands used to specify the low-power design intent for electronic systems, an alternative name of the IEEE 1801 Standard for Design and Verification of Low Power Integrated Circuits.

WIR

Wrapper instruction register, a required register in an IEEE 1500 implementation that controls what data register is selected for access.

Wrapped core

A core that has a wrapper chain along the I/O boundary of the design. Wrapped cores are used to implement hierarchical testing capability, which allows different hierarchy levels of the design to be tested independently.

Wrapper chain

A special scan chain in a core-wrapped design that consists of wrapper cells along the I/O boundary of the design. It can operate in inward-facing or outward-facing modes during testing to isolate the logic inside the core from logic outside the core.