



SIEMENS EDA

Tessent™ Shell User's Manual

Software Version 2021.3
Document Revision 23

Unpublished work. © 2021 Siemens

This material contains trade secrets or otherwise confidential information owned by Siemens Industry Software, Inc., its subsidiaries or its affiliates (collectively, "Siemens"), or its licensors. Access to and use of this information is strictly limited as set forth in Customer's applicable agreement with Siemens. This material may not be copied, distributed, or otherwise disclosed outside of Customer's facilities without the express written permission of Siemens, and may not be used in any way not expressly authorized by Siemens.

This document is for information and instruction purposes. Siemens reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Siemens to determine whether any changes have been made. Siemens disclaims all warranties with respect to this document including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement of intellectual property.

The terms and conditions governing the sale and licensing of Siemens products are set forth in written agreements between Siemens and its customers. Siemens' **End User License Agreement** may be viewed at: www.plm.automation.siemens.com/global/en/legal/online-terms/index.html.

No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Siemens whatsoever.

TRADEMARKS: The trademarks, logos, and service marks ("Marks") used herein are the property of Siemens or other parties. No one is permitted to use these Marks without the prior written consent of Siemens or the owner of the Marks, as applicable. The use herein of third party Marks is not an attempt to indicate Siemens as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A list of Siemens' trademarks may be viewed at: www.plm.automation.siemens.com/global/en/legal/trademarks.html. The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Support Center: support.sw.siemens.com

Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Revision History ISO-26262

Revision	Changes	Status/ Date
23	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Sep 2021
22	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Jun 2021
21	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Apr 2021
20	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Mar 2021

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact the Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Table of Contents

Revision History ISO-26262

Chapter 1

Tessent Shell Introduction	23
What Is Tessent Shell?	23
What Can You Do With Tessent Shell?	24
Tessent Shell Tcl Interface	26
Command Conventions	26
Command Completion	27
Dofile Transcription	28
Tcl Command Registration	29

Chapter 2

Tool Invocation, Contexts, Modes, and Data Models	31
Tool Invocation	31
Contexts and System Modes	33
Contexts	33
System Modes	35
Context and System Mode Combinations	35
Design Levels	36
Design Data Models	38
Flat Design Data Model	38
Hierarchical Design Data Model	38
ICL Data Model	40
Object Attributes	40

Chapter 3

Design Introspection and Editing	43
Design Introspection	44
Object Specification Format	44
Collections	44
Design Introspection Examples	47
Design Introspection Command Summary	51
Bundle Object Introspection	53
Bundle Object	53
Bundle Object Data Model	58
Introspection	61
Bundle Object Introspection Examples	65
Fanin and Fanout Tracing of Complex Signals and Bundle Object Tracing	68
Connectivity Through Complex Signals	72
Design Editing	77
Design Editing Examples	78

Design Editing Command Summary	83
Simulation Contexts	85
Simulation Context Overview	85
Introspection and Analysis Using Simulation Contexts	86
Automatic Design Mapping	89
ICL and TCD Post-Synthesis Update	89
Updating ICL Attributes From the Design	90
Matching Rules for Port Names in Post-Synthesis Update	91
Controlling the Name Mapping	91
ICL Objects vs. Design Objects Introspection	92
Chapter 4	
DFT Architecture Guidelines for Hierarchical Designs	95
Hierarchical DFT Overview	96
Physical Layout Regions in Hierarchical Test	96
Pattern Retargeting	97
Wrapped Cores and Wrapper Cells	97
Internal Mode and External Mode	98
On-Chip Clock Controller	99
Graybox Model	100
Top-Down Planning Before Bottom-Up Implementation	102
Clocking Architecture	102
Resource Availability	103
Test Scheduling	103
Specific Tasks That May Require Planning	106
Sample DFT Planning Steps	107
DFT Implementation Strategy	107
Chapter 5	
Tessent Shell Workflows	111
Tessent Shell Flow for Flat Designs	112
Overview of the RTL and Scan DFT Insertion Flow	113
First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan	116
Second DFT Insertion Pass: EDT, Hybrid TK/LBIST, and OCC	120
Loading the Design	121
Specifying and Verifying the DFT Requirements	123
Creating the DFT Specification	126
Generating the EDT, Hybrid TK/LBIST, and OCC Hardware	130
Extracting the ICL Module Description	130
Generating ICL Patterns and Running Simulation	131
Performing Synthesis	132
Performing Scan Chain Insertion (Flat Design)	133
Performing ATPG Pattern Generation	135
Simulating LBIST Faults	137
Considerations for Using Gate-Level Verilog Netlists	139
Tessent Shell Flow for Hierarchical Designs	141
Hierarchical DFT Terminology	142
How the DFT Insertion Flow Applies to Hierarchical Designs	144

Table of Contents

RTL and Scan DFT Insertion Flow for Physical Blocks	146
First DFT Insertion Pass: Performing Block-Level MemoryBIST	147
Second DFT Insertion Pass: Inserting Block-Level EDT and OCC	149
Specifying and Verifying the DFT Requirements: DFT Signals for Wrapped Cores ...	152
Performing Scan Chain Insertion: Wrapped Core	154
Verifying the ICL Model	157
Performing ATPG Pattern Generation: Wrapped Core	158
Running Recommended Validation Step for Pre-Layout Design Sign Off	163
RTL and Scan DFT Insertion Flow for the Top Chip	165
Top-Level DFT Insertion Example	166
First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan ...	168
Second DFT Insertion Pass: Inserting Top-Level EDT and OCC	171
Top-Level Scan Chain Insertion Example	175
Top-Level ATPG Pattern Generation Example	175
Performing Top-Level ATPG Pattern Retargeting	177
RTL and Scan DFT Insertion Flow for Sub-Blocks	180
DFT Insertion Flow for the Sub-Block	180
DFT Insertion Flow for the Next Parent Level	181
RTL and Scan DFT Insertion Flow for Instrument Blocks	184
DFT Insertion Flow for the Instrument Block	184
Instrument Block DFT Insertion Flow for the Next Parent Level	187
RTL and DFT Insertion Flow With Third-Party Scan	189
DFT Insertion Flow With Third-Party Scan Insertion	189
Wrapped Core DFT Insertion With Third-Party Scan	191
Top Chip DFT Insertion with Third-Party Scan	206
Tessent Shell Post-Layout Validation Flow	215
Overview of the Post-Layout Validation Flow	215
Soft Link TSDB and Post-Layout Netlist	216
Verify MemoryBIST, Boundary Scan and IJTAG Patterns	217
Verifying the Scan-Inserted ATPG Patterns	218
Post-Layout Validation When You Have Ungrouped IJTAG/OCC/EDT Logic	220
Test Bench Generation and Simulation in RTL Mode	224
Simulation Wrapper Creation	224
Test Bench Examples	227
Hybrid TK/LBIST Flow for Flat Designs	232
RTL and Scan DFT Insertion Flow With Hybrid TK/LBIST	232
Perform the First DFT Insertion Pass: Hybrid TK/LBIST	233
Perform the Second DFT Insertion Pass: Hybrid TK/LBIST	236
Perform Test Point Insertion	241
Perform Scan Chain Insertion (Hybrid Flow)	244
Performing ATPG Pattern Generation: Hybrid TK/LBIST	246
Performing LogicBIST Fault Simulation	249
Perform LogicBIST Pattern Generation	251
Running Multi-Load ATPG on Wrapped Core Memories with Built-In Self Repair	254
Overview of Multi-Load ATPG on Memories for Wrapped Cores With Built-in Self Repair	254
Performing Multi-Load ATPG Pattern Generation	255
Performing Multi-Load Top-Level ATPG Pattern Retargeting	258
Built-in Self Repair in Hierarchical Tessent MemoryBIST Flow	261

Performing Block Level BISR Insertion	261
Performing Chip Level BISR Insertion	264
Functional Repair Clock	264
Connection of the BISR Controller to Existing BISR Chains	264
Connection of the BISR Controller to an External Fuse Box	265
Connection of the BISR Controller to System Logic	266
Verification of Block and Chip Level BISR	266
Chapter 6	
Tessent Shell Automotive Workflow	269
Introduction to Tessent Automotive	270
Test Case Overview and Objective	271
Core-Level DFT Insertion for Automotive	276
DFT Insertion Flow for the Processor Core Physical Block	277
First DFT Insertion Pass: processor_core	278
Second DFT Insertion Pass: processor_core	281
Test Point, X-Bounding, and Scan Insertion: processor_core	285
ATPG Pattern Generation: processor_core	289
LogicBIST Fault Simulation: processor_core	291
LogicBIST Pattern Generation: processor_core	292
Interconnect Bridge/Open UDFM Generation: processor_core	294
Cell-Neighborhood UDFM Generation: processor_core	296
Automotive-Grade ATPG Pattern Generation: processor_core	300
DFT Insertion Flow for the GPS Baseband Physical Block	307
DFT Insertion Pass With In-System Test: gps_baseband	308
LogicBIST Fault Simulation With One NCP: gps_baseband	312
Interconnect Bridge/Open UDFM Generation: gps_baseband	312
Cell-Neighborhood UDFM Generation: gps_baseband	313
Automotive-Grade ATPG Pattern Generation: gps_baseband	313
Top-Level DFT Insertion for the Automotive Flow	314
First DFT Insertion Pass: Top with MemoryBIST, BISR, and Boundary Scan	314
Second DFT Insertion Pass: Top with EDT, OCC, and In-System Test	320
Scan Insertion for the Top Design	325
ATPG Pattern Generation for the Top Design	327
ATPG Pattern Retargeting for the Top Design	328
Interconnect Bridge/Open UDFM Generation for the Top Design	328
Cell-Neighborhood UDFM Generation for the Top Design	328
Automotive-Grade ATPG Pattern Generation for the Top Design	329
UDFM Generation for Cell-Aware ATPG	329
TCA Based Pattern Sorting	331
Functional Mode Fault Tolerance for Static IJTAG Signals	333
Chapter 7	
TSDB Data Flow for the Tessent Shell Flow	339
Core-Level or Flat TSDB Data Flow	339
Top-Level TSDB Data Flow	344

Chapter 8
Streaming Scan Network (SSN)..... 351

- Tessent SSN Workflows 352
 - Block-Level SSN Insertion and Verification 355
 - First DFT Insertion Pass: Performing Block-Level MemoryBIST 357
 - Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN 357
 - Synthesis for Block-Level Insertion..... 364
 - Creating the Post-Synthesis TSDB View (Block-Level) 365
 - Performing Scan Chain Insertion With Tessent Scan (Block-Level)..... 367
 - Verifying the ICL-Based Patterns After Synthesis (Block-Level)..... 370
 - Generating Block-Level ATPG Patterns 373
 - Top-Level SSN Insertion and Verification..... 380
 - First DFT Insertion Pass: Performing Top-Level MemoryBIST 382
 - Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN..... 385
 - Synthesis for Top-Level Insertion 393
 - Creating the Post-Synthesis TSDB View (Top-Level) 394
 - Performing Scan Chain Insertion With Tessent Scan (Top-Level) 394
 - Verifying the ICL-Based Patterns After Synthesis (Top-Level) 395
 - Generating Top-Level ATPG Patterns 395
 - Retargeting ATPG Patterns 397
 - Performing Reverse Failure Mapping for SSN Pattern Diagnosis..... 400
- Advanced Topics 405
 - Streaming-Through-IJTAG Scan Data 405
 - On-Chip Compare With SSN 408
 - Types of Clock Networks To Use With SSN 417
 - Broadcast to Identical SSN Datapaths 421
 - Yield Statistics on ATE With SSN..... 422
 - Manufacturing Patterns With SSN 427
 - Manufacturing Pattern Quick Reference 428
 - SSN Pattern Structure..... 429
 - How To Write Complete SSN Patterns 430
 - How To Write ssn_setup and ssn_end Procedures to Separate Files..... 432
 - How To Write SSN On-Chip Compare Patterns 435
 - How To Write Streaming-Through-IJTAG Patterns 436
 - SSN Debug on the Tester..... 438
 - Signoff Patterns With SSN 444
 - Block-Level Signoff Patterns 445
 - Top-Level Signoff Patterns 449
 - Signoff Pattern Quick Reference 452
 - SSN SDC Constraints in the Design Flow 453
 - Overview of SSN-Related SDC Procs and Constraints 454
 - SSN/SSH SDC Constraint Descriptions..... 462
- Tessent SSN Examples and Solutions 484
 - Third-Party OCCs With SSN 484
- SSN Frequently Asked Questions 486
- SSN Limitations..... 488

Chapter 9	
Tessent Examples and Solutions	491
How to Avoid Simulation Issues When Using the Two-Pin Serial Port Controller	491
How to Handle Clocks Sourced by Embedded PLLs During Logic Test	494
How to Design Capture Windows for Hybrid TK/LBIST	500
How to Use Boundary Scan in a Wrapped Core	502
How to Use an Older Core TSDB With Newly-Inserted DFT Cores	504
TAP Configuration	506
Insert a Stand-Alone TAP in a Design	506
Insert a TAP with an IJTAG Host Scan Interface	508
Insert a Compliance Enable TAP with an IJTAG Interface	509
Insert a Daisychained TAP	511
Insert a Primary TAP	512
Insert a Secondary TAP	514
Connecting to a Third-Party TAP	517
How to Set Up Third-Party Synthesis	517
How to Set Up Support for Third-Party OCCs	520
How to Configure Files for Third-Party OCCs	520
Test Logic Insertion	521
Configuration for Scan Insertion	523
Pattern Generation and Simulation	523
Post-Synthesis Update	527
ICL and TCD Post-Synthesis Update	527
Limitations Related to SystemVerilog Interface Arrays	528
Updating ICL Attributes From the Design	529
Matching Requirements for Port Names in Post-Synthesis Update	530
Design Name Mapping Commands	531
Design Name Mapping	531
Default Matching Rules for the <code>get_pins</code> , <code>get_ports</code> , and <code>get_instances -match_rtl_reg</code> Commands	531
Chapter 10	
Test Procedure File	533
Test Procedure File Creation	534
Test Procedure File Syntax	534
Test Procedure File Structure	539
#include Statement	539
Set Statement	540
Alias Definition	543
Timing Variables	545
Timeplate Definition	548
Multiple-Pulse Clocks	554
The <code>pulse_clock</code> Statement	554
Inferred Timing	555
Differences Between Default <code>add_clock</code> and 1x Multiplier Clock	556
Always Block	556
Procedure Definition	557
Clock Control Definition	567

Table of Contents

The Procedures	576
Test_Setup (Optional)	577
Shift (Required)	580
Alternate Shift Procedure (Optional)	582
Load_Unload (Required)	583
Shadow_Control (Optional)	586
Master_Observe (Sometimes Required)	588
Shadow_Observe (Optional)	588
Skew_Load (Optional)	589
Clock_run (Optional)	591
Capture Procedures (Optional)	593
Rules for Creating and Editing a Default Capture Procedure	594
Rules for Creating and Editing Named Capture Procedures	594
Slow and Load Types in the Cycle Statement	596
launch_capture_pair Statement	597
Clock_po (Optional)	598
Clock_sequential (Optional)	599
Init_force (Optional)	599
Test_end (Optional, all ATPG tools)	600
Sub_procedure	602
Additional Support for Test Procedure Files	604
Creating Test Procedure Files for End Measure Mode	605
Serial Register Load and Unload for LogicBIST and ATPG	609
Register Load and Unload Use Models	609
Static Versus Dynamic Register Variables	609
Test Procedure File Modifications	610
Dofile Modifications	613
Serial Load and Unload DRC Rules	616
P13 and P54	616
P66	616
W5	617
Procedure Examples	618
Notes About Using the stil2mgc Tool	622
Extraction of Strobe Timing Information from STIL (SPF)	622
The STIL ClockStructures Block	622
Test Procedure File Commands and Output Formats	623
Chapter 11	
Tessent Visualizer	625
Invoking Tessent Visualizer	626
Framework Overview	627
Tessent Visualizer Components and Preferences	630
Tables	631
Table Toolbar Features	632
Columns and Filters Editor	634
Table Filters	635
Data Sorting	637
Row Highlighting	637

Schematics	639
Toolbar	640
Schematic Symbols	643
Context Tables	655
Signal Net Tracing Strategies	665
Displayed Property	668
Tessent Shell Attributes	668
Mouse Gestures	670
Tessent Visualizer Preferences	671
Macros	672
Tooltips	673
Gate Report Settings	674
Saving and Restoring the Session State	675
Window Title Prefixes	675
Tessent Visualizer GUI Reference	677
Hierarchical Schematic	677
Flat Schematic	682
Instance Browser	684
Wave Generator	686
Cell Library Browser	688
DRC Browser	689
Pin Data	691
Transcript	692
Text/HDL Viewer	693
Diagnosis Report Viewer	695
Search Features	697
Using Tessent Visualizer to Debug Design Issues	699
Accessing Tutorials for Tessent Visualizer	703
Accessing Videos for Tessent Visualizer	704
Chapter 12	
Configuration Data Visualizer	707
Modifying the Contents of the Configuration Data Visualizer	707
Adding a Test Data Register to a SIB Example	708
Adding a Multiplexer to a SIB Example	708
Customizing the DFT Specification for EDT	710
Chapter 13	
Timing Constraints (SDC)	715
Generating and Using SDC for Tessent Shell Embedded Test IP	716
SDC File Generation with Tessent Shell	716
SDC Design Synthesis with Tessent Shell	718
Preparation Step 1: Sourcing SDC File	718
Preparation Step 2: Setting and Redefining Tessent Tcl Variables	718
Preparation Step 3: Verifying the Declaration of Functional Clocks	720
Preparation Step 4: Redefining Other Tessent Tcl Variables	721
Synthesis Step 1: Applying the SDC Constraints	722
Synthesis Step 2: Preparing the DFT Logic for Synthesis	722

Table of Contents

Synthesis Step 3: Synthesizing Your Design	723
Synthesis Step 4: Writing Out Your Final SDC	723
Synthesis Step 5: Writing Out Your Final Netlist	723
Running Layout with Tessent Shell DFT	723
SDC for Modal Static Timing Analysis	726
Checking Your Functional Logic Alone	726
Checking Your Embedded Test Logic	726
VHDL and Mixed Language Designs	728
VHDL Generate Blocks	728
SDC File Contents	730
tessent_set_default_variables	730
tessent_create_functional_clocks	731
tessent_<design_name>_set_dft_signals	731
<ltest_prefix>_disable	732
tessent_set_non_modal	732
tessent_<design_name>_kill_functional_paths	733
IJTAG Instrument	734
LOGICTEST Instruments	735
MemoryBIST Instrument	748
BoundaryScan Instrument	751
Hierarchical STA in Tessent	752
Mapping Procs	756
Synthesis Helper Procs	758
Example Scripts using Tessent Tool-Generated SDC	760
Example Design Compiler Synthesis Script	760
Example Genus Synthesis Script	761
Example PrimeTime STA Script	764
Appendix A	
The Tessent Tcl Interface	771
General Tcl Guidelines in Tessent Shell	771
Guidelines for Modifying Existing Dofiles for Use with Tcl	773
Special Tcl Characters	775
Custom Tcl Packages in Tessent Shell	777
Tcl Resources	778
Appendix B	
Synthesis Guidelines for RTL Designs with Tessent Inserted DFT	779
DFT Insertion With Tessent	780
Synthesis Guidelines	781
SystemVerilog and Port Name Matching	783
Appendix C	
Clocking Architecture Examples	785
Clocks Driven by Primary Inputs	785
Clock Generators Outside the Cores	786
Clock Generators Inside the Cores	786
Clock Sourced From A Core With Embedded PLL	787

Clock Mesh Synthesis	788
Appendix D	
Tessent Visualizer Keyboard Shortcuts	791
Appendix E	
Formal Verification	793
Constraints for Formality Scripts.....	793
Constraints for Conformal Scripts.....	794
Appendix F	
Transitioning from the Classic Point Tools	797
Transitioning from the Classic FastScan Point Tool.....	797
Transitioning from the Classic TestKompress Point Tool.....	798
Transitioning from the Classic DFTAdvisor Tool.....	799
Transitioning from the Classic Diagnosis Tool.....	800
Appendix G	
Getting Help	801
The Tessent Documentation System.....	801
Global Customer Support and Success	802
Index	
Third-Party Information	

List of Figures

Figure 2-1. Hierarchical Design Example	39
Figure 3-1. Hierarchical Design Example With Colors.	48
Figure 3-2. Complex Signals with Tessent Visualizer	74
Figure 3-3. Hierarchical Design Example	79
Figure 3-4. Inverter Inception	81
Figure 3-5. Tessent Visualizer Flat Schematic (gate level).	87
Figure 4-1. Wrapped Cores	97
Figure 4-2. Internal Mode	98
Figure 4-3. External Mode.	99
Figure 4-4. On-Chip Clock Controller.	100
Figure 4-5. Graybox Model	101
Figure 4-6. Compression Analysis Example	105
Figure 5-1. Two-Pass Insertion Flow for RTL, Flat Designs	114
Figure 5-2. DFT-Ready Design	115
Figure 5-3. After the First DFT Insertion Pass	115
Figure 5-4. After the Second DFT Insertion Pass	116
Figure 5-5. Flow for the Second DFT Insertion Pass	120
Figure 5-6. Flow for Loading the Design	121
Figure 5-7. Flow for Specifying and Verifying the DFT Requirements	123
Figure 5-8. Flow for Creating the DFT Specification	126
Figure 5-9. Flow for Inserting the Second-Pass Hardware	130
Figure 5-10. Flow for Extracting ICL	131
Figure 5-11. Flow for Generating and Simulating ICL Patterns	132
Figure 5-12. Two-Pass Insertion Flow for RTL, Gate-Level Designs	140
Figure 5-13. Hierarchical Design Levels.	143
Figure 5-14. Two-Pass Insertion Flow, Physical Blocks.	146
Figure 5-15. Flow for Specifying and Verifying the DFT Requirements for Wrapped Cores	152
Figure 5-16. ATPG Pattern Generation Flow for Wrapped Cores	159
Figure 5-17. Two-Pass Insertion Flow for RTL, Top Level	165
Figure 5-18. Top-Level Example, Before DFT Insertion	167
Figure 5-19. Top-Level Example, After DFT Insertion for Wrapped Cores.	167
Figure 5-20. Top-Level Example, After DFT Insertion at the Top Level.	168
Figure 5-21. Two-Pass Insertion Flow for RTL, Wrapped Cores, and Third-Party Scan . . .	191
Figure 5-22. DFT Signals and Multiplexer Logic Support Hierarchical ATPG	194
Figure 5-23. At-Speed Flows in the Wrapper Chain	196
Figure 5-24. Current Level Path to Child Core Wrapper Chains	200
Figure 5-25. Two-Pass Insertion Flow for RTL, Top Level, and Third-Party Scan	206
Figure 5-26. Top Level EDT Connection to Child Cores	210
Figure 5-27. Post-Layout Validation Flow	215
Figure 5-28. Post-Layout Validation Flow with Ungrouped IJTAG/OCC/EDT Logic	220

Figure 5-29. Two-Pass Insertion Flow With Hybrid TK/LBIST.	233
Figure 5-30. Two-Pass Insertion Flow for RTL, Wrapped Cores	255
Figure 5-31. Two-Pass Insertion Flow for RTL, Top Level	258
Figure 6-1. Integrated Tessent DFT Solution for Automotive	270
Figure 6-2. Initial Design for Automotive Test Case	273
Figure 6-3. DFT Integration at the Core Level for Automotive	273
Figure 6-4. DFT Integration at the Top Level for Automotive	274
Figure 6-5. DFT Insertion Flow for Automotive Test Case	275
Figure 6-6. DFT Insertion Flow for processor_core	277
Figure 6-7. Enhanced Clocking and DFT Controls After Second DFT Insertion Pass	282
Figure 6-8. ATPG Pattern Generation Flow for processor_core	289
Figure 6-9. Interconnect Bridge/Open UDFM Generation Flow for processor_core	295
Figure 6-10. Cell-Neighborhood UDFM Generation Flow for processor_core	298
Figure 6-11. ATPG Topoff Run Flow With a UDFM for processor_core	302
Figure 6-12. ATPG Run Flow With All UDFM for processor_core	305
Figure 6-13. DFT Insertion Flow for gps_baseband	307
Figure 6-14. Dofile Example for Top-Level Scan Chain Insertion, Automotive Flow	326
Figure 6-15. Cell-Aware UDFM Generation Flow	330
Figure 6-16. TCA Based Pattern Sorting Flow	332
Figure 6-17. Hardware Fault Tolerant Module Example	335
Figure 6-18. HFT Module in the Single-Pass Flow	336
Figure 6-19. HFT Modules in the Two-Pass Flow	337
Figure 7-1. TSDB Data Flow, Core Level, First Insertion Pass	341
Figure 7-2. TSDB Data Flow, Core Level, Second Insertion Pass	342
Figure 7-3. TSDB Data Flow, Core Level, Scan Insertion	343
Figure 7-4. TSDB Data Flow, Core Level, Pattern Generation	343
Figure 7-5. TSDB Data Flow, Top Level, First Insertion Pass	345
Figure 7-6. TSDB Data Flow, Top Level, Second Insertion Pass	346
Figure 7-7. TSDB Data Flow, Top Level, Scan Insertion	347
Figure 7-8. TSDB Data Flow, Top Level, ATPG Pattern Generation	348
Figure 7-9. TSDB Data Flow, Top Level, ATPG Pattern Generation With Pattern Retargeting 349	349
Figure 8-1. Directory Structure of Reference Testcase	353
Figure 8-2. SSN Block-Level Workflow	356
Figure 8-3. SSN Top-Level Workflow	381
Figure 8-4. Multiplexer Node for Bypassing the gps_baseband Blocks	389
Figure 8-5. Simplified Tessent Diagnosis Two-Step Flow	401
Figure 8-6. Streaming-Through-IJTAG Mode	406
Figure 8-7. Faults in Comparator Logic That Could Mask Real Faults	412
Figure 8-8. SSN Datapath in Tiling Architecture	420
Figure 8-9. Stepping Down the Clock Tree on the Last Pipeline Stage	421
Figure 8-10. Packet Rotation on the SSN Bus	423
Figure 8-11. Procedure and Data Structure of an SSN Pattern	430
Figure 8-12. SSN Procedures With test_setup and test_end in Pattern Order	431
Figure 8-13. SSN Patterns With the -maxloads 50 Option	432

List of Figures

Figure 8-14. SSN Pattern Files With Order for Application on Tester	434
Figure 8-15. SSN Pattern Files With Combined Setup Showing Tester Order	435
Figure 8-16. SSN Pattern Files With On-Chip Compare Self-Test	436
Figure 8-17. Tester Application Order for Streaming-Through-IJTAG	438
Figure 8-18. Order To Apply Patterns for SSN Pattern Failure Debugging	439
Figure 8-19. Order To Apply ICLNetwork Verify Patterns to the Tester	440
Figure 8-20. Order To Apply SSN Continuity Patterns to the Tester	441
Figure 8-21. Order To Apply SSN On-Chip Compare Self-Test Patterns	441
Figure 8-22. Order To Apply SSH Loopback Patterns	442
Figure 8-23. Order To Apply SSH Loopback, Chain, and Scan Patterns	443
Figure 8-24. FIFO MCP Example	470
Figure 8-25. SSH Clock Signal Generation	473
Figure 8-26. SSH Clock Group Constraints	476
Figure 8-27. SSH scan_en and edt_update Generation Circuit	476
Figure 8-28. scan_en Margin Definition Diagram	477
Figure 8-29. Gated Scan Clocks With All *extra_cycle* Variables Set to 1 (Default)	478
Figure 8-30. Gated Scan Clocks With All *extra_cycle* Variables Set to 0	479
Figure 8-31. Divided Shift Clocks With Ratio of 4 and *extra_cycle* Variables at 0	479
Figure 8-32. Divided Shift Clocks With Ratio of 4 and *extra_cycle* Variables at 1	479
Figure 8-33. SSH Scan Chain Interface Circuit	480
Figure 8-34. Localized Scan Timing With Shift-Only Mode OCC	485
Figure 9-1. TRST Pulse Generator inside TPSP	493
Figure 9-2. Example Chip with PLL Embedded Inside Lower Core	496
Figure 9-3. Active OCCs During Internal Test Modes of corec and coreb	497
Figure 9-4. Active OCCs During Internal Test Modes of corea and coreb	498
Figure 9-5. Active OCCs During Test Modes of Top Level	499
Figure 9-6. Wrapped Core Boundary Scan Flow	503
Figure 10-1. 200ns Timing Waveform	555
Figure 10-2. Shift Procedure	580
Figure 10-3. Timing Diagram for Shift Procedure	581
Figure 10-4. Load_Unload Procedure	584
Figure 10-5. Timing Diagram for Load_Unload Procedure	585
Figure 10-6. Shadow_Control Procedure	586
Figure 10-7. Master_Observe Procedure	588
Figure 10-8. Shadow_Observe Procedure	589
Figure 10-9. Skew_Load Procedure	590
Figure 10-10. Skew_load applied within Pattern	591
Figure 10-11. Full Clock Sequential Pattern	599
Figure 10-12. Init_force Procedure Usage	600
Figure 11-1. Tessent Visualizer Split View	629
Figure 11-2. Common Table Toolbar Features	632
Figure 11-3. Columns and Filters Editor	634
Figure 11-4. Selection Colors	638
Figure 11-5. Schematic With Overview Pane and Attributes Table	640
Figure 11-6. Schematic Elements: Toolbar, Address Bar, and Selection Buttons	641

Figure 11-7. Hierarchical Schematic Showing Pins, Ports, Instances, and Nets	643
Figure 11-8. Hierarchical Instance With Callout and Pin Data	644
Figure 11-9. Showing the Internal Connectivity of a Hierarchical Instance	645
Figure 11-10. Hierarchical Instance With DRC Callout	645
Figure 11-11. Hierarchical Cell	646
Figure 11-12. Hierarchical Instance Representing an Assign Statement.	646
Figure 11-13. Black-Boxed Instance Example	647
Figure 11-14. Objects in the Flat Schematic Tab	647
Figure 11-15. Persistent Buffer Symbol	648
Figure 11-16. Bus Index Displayed at Rip Point	649
Figure 11-17. Collapsed Buffers and Inverters With No Hidden Fanout	652
Figure 11-18. Expanded Buffers and Inverters With No Hidden Fanout	652
Figure 11-19. Collapsed Buffers/Inverters With Hidden Fanout	653
Figure 11-20. Expanded Buffers/Inverters With Hidden Fanout.	653
Figure 11-21. Expanded Buffers/Inverters With Fanout Revealed	654
Figure 11-22. Unfolding an Instance	655
Figure 11-23. Folded Cell Group.	655
Figure 11-24. Unfolded Cell Group.	655
Figure 11-25. Schematic Elements: Context Table	656
Figure 11-26. Tracer Context Table	657
Figure 11-27. Context Table for Instance Pins (Hierarchical Schematic).	658
Figure 11-28. Context Table for Instances (Hierarchical Schematic)	659
Figure 11-29. Context Table for Nets (Hierarchical Schematic).	660
Figure 11-30. Fanout Replaced by User-Added Primary Input.	661
Figure 11-31. Flat Sink	661
Figure 11-32. Fanout Replaced by Multiple User-Added Primary Inputs	662
Figure 11-33. Multiple Flat Sinks	662
Figure 11-34. Context Table for Multiple Flat Sinks	663
Figure 11-35. User-Added Primary Inputs	664
Figure 11-36. Netlist Driver and Context Table for User-Added Primary Inputs.	665
Figure 11-37. User-Added Primary Inputs (Flat Schematic).	665
Figure 11-38. Decision Point Strategy on Hierarchy Boundary at q[6] Port.	666
Figure 11-39. Choosing a Path From the Tracing Table	666
Figure 11-40. Decision Point Strategy.	667
Figure 11-41. Tessent Shell Attributes Table	669
Figure 11-42. Mouse Gestures in Tessent Visualizer Schematics.	670
Figure 11-43. Light, Dark, and High-Contrast Color Themes.	671
Figure 11-44. Preferences Dialog Box (Text Viewers Tab)	672
Figure 11-45. Macro Editor	673
Figure 11-46. Running a Macro.	673
Figure 11-47. Tooltip for a Collapsed Buffer Indicator.	674
Figure 11-48. Tooltip for a Filter Box	674
Figure 11-49. Tooltip for an Action Button.	674
Figure 11-50. Setting a Window Title Prefix	676
Figure 11-51. Visualizer Window Label	676

List of Figures

Figure 11-52. Hierarchical Instances With Pins Shown and Hidden.	678
Figure 11-53. Hierarchical Schematic Symbols	679
Figure 11-54. Bus Tracing in the Hierarchical Schematic.	680
Figure 11-55. Bus Sub-Ranges	681
Figure 11-56. Example of a Tied-Value Marker With Associated Pin Table	682
Figure 11-57. Flat Schematic (Cell Grouping On)	683
Figure 11-58. Instance Browser Elements.	685
Figure 11-59. Debugging in the Instance Browser	686
Figure 11-60. Wave Generator	687
Figure 11-61. A GTKWave Viewer Application Invoked From Wave Generator	688
Figure 11-62. Cell Library Browser	689
Figure 11-63. DRC Browser With Data Grouped By Rule	690
Figure 11-64. Flat Schematic Showing a DRC Violation	691
Figure 11-65. Pin Data Tab	692
Figure 11-66. Transcript Tab With an Interactive Command Line	693
Figure 11-67. Text/HDL Viewer	694
Figure 11-68. Diagnosis Report Viewer (Text View)	696
Figure 11-69. Diagnosis Report Viewer (Table View)	697
Figure 11-70. Search Tab: Instances	698
Figure 11-71. Search Tab: Pins	698
Figure 11-72. Search Tab: Gate Pins	699
Figure 11-73. Searching by Name	702
Figure 13-1. tessent_test_clock and tessent_shift_capture_clock Creation.	737
Figure 13-2. Generated Clock Muxed Multi-Clock Memories	750
Figure B-1. Problem Occurrence Creating the Gate Level Netlist	779
Figure C-1. OCCs for Clocks Driven by Primary Inputs	785
Figure C-2. OCCs With Clock Generators at Chip Level, Asynchronous Clocks	786
Figure C-3. OCCs With Clock Generators Inside the Cores, Asynchronous Clocks	787
Figure C-4. Clock Sourced From A Core With Embedded PLL, With MUX	787
Figure C-5. Clock Sourced From A Core With Embedded PLL, Without MUX	788
Figure C-6. Clock Mesh Synthesis	789

List of Tables

Table 1-1. Tessent Shell Command Conventions	26
Table 1-2. Commands for Tcl Command Registration	29
Table 2-1. Application-Specific Environment Variables	32
Table 2-2. Tessent Shell Contexts	33
Table 2-3. Tessent Shell System Modes	35
Table 2-4. Tessent Shell Context and System Mode Commands	36
Table 3-1. Commands That Interact With Collections	46
Table 3-2. Design Introspection Commands	51
Table 3-3. Attribute Introspection Commands	52
Table 3-4. base_class and base_type System Verilog Examples	59
Table 3-5. Complex Signals Design Introspection Commands	65
Table 3-6. Design Editing Commands	77
Table 3-7. Design Editing Command Summary	84
Table 4-1. Test Schedule Example	106
Table 5-1. Test Pattern Bit Name Assignments	228
Table 7-1. Core-Level TSDB Data Flow Inputs and Outputs	340
Table 7-2. Top-Level TSDB Data Flow Inputs and Outputs	344
Table 8-1. SSN Manufacturing Pattern Summary	427
Table 8-2. Manufacturing Pattern Quick Reference	428
Table 8-3. Block-Level Verification Pattern Summary	448
Table 8-4. Top-Level Verification Pattern Summary	451
Table 8-5. Signoff Pattern Quick Reference	452
Table 8-6. set_load_unload_timing_options Arguments and SDC Variables	474
Table 8-7. SSH Chain Interface Circuit SDC False Paths	481
Table 8-8. SSH Chain Interface Circuit SDC Multicycle Paths	482
Table 10-1. Reserved Punctuation Characters	535
Table 10-2. Procedure File Tool Command Summary	623
Table 11-1. Filtering Summary	636
Table 11-2. Schematic Toolbar Actions	641
Table 11-3. Marker Symbols	649
Table A-1. Common Dofile Issues and Solutions	774
Table A-2. Common Tcl Characters	775
Table D-1. Global Shortcuts	791
Table D-2. Schematic Shortcuts	791
Table D-3. Instance Browser Shortcuts	792
Table D-4. Filter Editing Shortcuts	792
Table D-5. Text Search Shortcuts	792

Chapter 1

Tessent Shell Introduction

Tessent™ Shell is a platform from which you can run all the Tessent tools. The platform includes shared design data, a common database, and powerful scripting utilities that provide a fully automated design-for-test (DFT) flow as well as the ability to customize the flow to fit specific requirements.

What Is Tessent Shell?	23
What Can You Do With Tessent Shell?	24
Tessent Shell Tcl Interface	26
Command Conventions.	26
Command Completion	27
Dofile Transcription	28
Tcl Command Registration.....	29

What Is Tessent Shell?

Tessent™ Shell is a DFT environment within which you can perform all the tasks required to insert DFT hardware, generate manufacturing test patterns, and perform post-silicon tasks such as diagnosis and yield analysis.

Tessent Shell provides a flexible design flow that supports a high level of automation or customization. Key aspects of the environment that support automation as well as enhance flexibility:

- **Shared Data Models** — Tessent Shell uses data models to store your design data. The Tessent environment shares these models across all tools and functions, and you can access the data stored within them to customize your design flow.

For standard automated flows, you do not need to be aware of this infrastructure. For those that need to extend the native tool commands, they have the same access to the design data model as the Tessent Shell tools do.

For more information about the data models, refer to “[Design Data Models](#)” on page 38.

- **Attributes** — Attributes are characteristics associated with design objects, such as library cells, pins, and modules, that are stored within data models. Some are predefined, and you can also create your own attributes. Using attributes increases visibility into your design, enabling you to manipulate and query the features of the design objects.
- **Design Introspection** — Introspection is the act of examining the design objects and attributes stored within the data models. Introspection enables you to retrieve the design

data you need so that you can use Tcl scripting techniques to automate your own custom designs flows.

For more information about design introspection, refer to “[Design Introspection](#)” on page 44.

- **Tcl** — Use this scripting language across all Tessent tools and functions. Through scripting, you can customize and extend the Tessent Shell standard flows as needed for your design requirements.

For more information about Tcl usage, refer to “[Tessent Shell Tcl Interface](#)” on page 26.

- **Tessent Shell Database (TSDB)** — The TSDB is a database that stores all the directories and files that Tessent Shell generates as you move through a design flow. The TSDB enhances flow automation by acting as the central location where Tessent tools can access the data it requires for the current task, whether that task be reading in a design, performing DRC, inserting logic test hardware, or performing ATPG pattern generation.

For more information about the TSDB, refer to “[TSDB Data Flow for the Tessent Shell Flow](#)” on page 339.

- **IJTAG Automation** — By default, the DFT instruments that you create with Tessent Shell are controlled through an IJTAG network and the IEEE 1687 protocol. Tessent Shell automatically inserts the IJTAG infrastructure. This simplifies test setup and control of all the instruments at all levels of hierarchy, and enables reuse in hierarchical designs and test-benching at any stage of chip development.

For more information about IJTAG, refer to the “[Tessent IJTAG User’s Manual](#).”

What Can You Do With Tessent Shell?

The Tessent Shell platform is a jumping-off point for accomplishing the full array of DFT tasks available within the Tessent tool suite. Specific DFT tasks may require different licenses, but all tools may be launched from and managed via Tessent Shell.

DFT tasks include:

- **Instrument Insertion** — Generate and insert logic test hardware such as EDT (embedded deterministic testing) and OCC (on-chip clock controller), in addition to LogicBIST, MemoryBIST, in-system test, and boundary scan.
- **Scan Analysis and Insertion** — Perform scan analysis and scan chain insertion.
- **ATPG** — Generate ATPG patterns and perform fault simulation, including compression technology.

- **Defect Diagnosis and Yield Analysis** — Perform test failure diagnosis to determine a defect’s most probable failure mechanism, as well as statistical analysis of diagnostic failures to find systemic defects.

Additionally, Tessent Shell provides general-purpose design editing support so that you can modify your design netlists as needed. You can work on the command line as described in “[Design Editing](#),” or use the [Tessent Visualizer](#) graphical interface.

If you are getting started with Tessent Shell, refer to “[Tessent Shell Workflows](#)” for information about the Tessent Shell workflow for RTL and scan DFT insertion.

Tessent Shell Tcl Interface

The Tessent Shell environment uses a Tcl-based interface that you can use from the command line or by writing dofile scripts.

The Tcl interface supports Tcl constructs such as variables, command substitution, flow control, and procedures. You can embed Tcl constructs in tool commands and embed tool commands within Tcl constructs the same as with any Tcl command.

For guidelines about using Tcl in Tessent Shell, and information about converting existing dofiles and using special characters, refer to “[The Tessent Tcl Interface](#)” appendix.

Command Conventions	26
Command Completion	27
Dofile Transcription	28
Tcl Command Registration	29

Command Conventions

Tessent Shell provides a unified Tcl-style command set and naming convention. Commands that begin with a certain first word (for example, “get” in get_attribute_value_list) perform operations on the current data model.

[Table 1-1](#) provides a summary listing by command first word of the Tessent Shell commands. Refer to the *Tessent Shell Reference Manual* for a complete list of commands and options. You can also use the “help” command with a wildcard to see a complete list of commands that start with the same word:

```
> help get_*
```

Table 1-1. Tessent Shell Command Conventions

Command First Word	Types of Operations Performed
append_ compare_ copy_ filter_ foreach_ index_ remove_ sizeof_ sort_	Operates on collections. Refer to “ Collections ” for more information.

Table 1-1. Tessent Shell Command Conventions (cont.)

Command First Word	Types of Operations Performed
get_	Returns data strings or collections of lists, objects, or object attributes from the design object model for introspection.
read_	Reads files into memory, such as libraries and netlists.
report_	Displays information about a specific item, such as the current context or licenses currently checked out.
set_	Specifies options, contexts, modes, attributes, and the current design. Refer to “ Contexts and System Modes ” and “ Object Attributes ” for more information.
write_	Writes design data to a file or set of files.

Command Completion

In the Tessent Shell interface, you can use the Tab key to complete command names, command option names, and command option values. If the command you type is ambiguous, pressing the Tab key lists all matching commands. Additionally, within a command, pressing the Tab key can match variable names with \$.

Many Tessent Shell commands are constructed as follows:

```
command_name command_options command_option_values
```

For example:

```
set_config_value -in_wrapper /TopBuilder(CHIP)
```

Using Tab key completion, you can complete each of the command parts (name, options, and option values).

For information about Tcl shell handling in Tessent Shell, refer to the [set_tcl_shell_options](#) command.

Command Name Tab Completion

As an example of command name completion, you can abbreviate the `get_attribute_list` command in the Tessent Shell interface by typing the following:

```
SETUP> g_a_l <tab>
```

Pressing the Tab key after the string completes and expands the command so that it looks like this:

```
SETUP> get_attribute_list
```

Command Option Name Tab Completion

Tab completion for command option names is available for most commands. For example:

```
SETUP> get_config_value -m<tab>
-meta_id      -meta_name    -meta_object

SETUP> get_config_value -meta_
```

Command Option Value Tab Completion

Tab completion for command option values is available for the following value types:

- Configuration data path
- Enum (Enumerated) values

Configuration Data Path Example

In this example, pressing the Tab key on the partial configuration data path completes the path:

```
SETUP> set_config_value -in_wrapper /TopB<tab>
SETUP> set_config_value -in_wrapper /TopBuilder(CHIP)/
```

Enum Value Example

In this example, using the Tab key lists the supported time values -time_unit option to the get_config_value command:

```
SETUP> get_config_value -time_unit <tab>
as_is fs    ms     ns     ps     s      us

SETUP> get_config_value -time_unit
```

Dofile Entries

You can use abbreviated commands (for example, **g_a_l** for **get_attribute_list**) in Tessent Shell dofiles, but it is best to always use the full command name. There is no fixed minimum typing for any command or option, and current minimum typing could change in future releases because of the addition of new commands or options. Using the full command names also adds readability to your dofiles.

Dofile Transcription

By default, the transcript style is Full, which means the tool writes out all commands read from a dofile before any Tcl evaluation occurs. The command appears in the transcript as entered but is commented out.

During Tcl evaluation, the Tcl commands are written to the transcript as follows:

- The tool writes all commands from the dofile with a “// *command*.” prefix. This includes Tessent Shell commands, Tcl commands, and complete loop and if/else constructs.
- The tool writes all commands embedded in Tcl constructs to the transcript as executed with a “// *sub_command*.” prefix. Tessent Shell completes the variable and command substitutions and writes the resolved values in the loop to the transcript.

Note

This does not apply to introspection commands. Introspection commands are not written to the transcript as subcommands.

- The tool indents nested dofiles when they are written to the logfile.
- The tool can read a Tcl routine in much the same manner as a dofile. If you issue the source command (>source *my_report_env*), then the Tcl commands in the routine are not transcribed.

Control the Tessent Shell transcription behavior using the [set_transcript_style](#) command.

For information about modifying existing dofiles for use with the Tessent Shell Tcl interface, refer to “[Guidelines for Modifying Existing Dofiles for Use with Tcl.](#)”

Tcl Command Registration

You can convert any Tcl procedure into a Tessent Shell application command. This enables you to implement functionality in Tcl and register that functionality as a command so that Tessent Shell handles it like any other built-in command. Like any built-in command, newly registered Tcl commands support Tab completion, and you can control their availability relative to the context and system mode. The help system also includes the new command.

The following commands enable you to register new commands and to define arguments and options for those commands.

Table 1-2. Commands for Tcl Command Registration

Command	Description
display_message	Controls messages produced by Tcl commands.
lock_current_registration	Locks all current Tcl commands.
register_tcl_command	Registers a Tcl command.
unregister_tcl_command	Unregisters a Tcl command.

When you first execute the new command, the tool performs syntactic and semantic checks, and provides the parsed results to your Tcl implementation of the command. The tool also provides a mechanism to automatically define and register user-defined commands at tool invocation.

For more information about creating your own application commands, refer to the examples included with the [register_tcl_command](#) description in the *Tessent Shell Reference Manual*.

Chapter 2

Tool Invocation, Contexts, Modes, and Data Models

In the Tessent Shell environment, setting the context and system mode orients the tool as to which task you want to perform. The tool uses design data models to store the relevant data about your design.

Tool Invocation	31
Contexts and System Modes	33
Design Data Models	38

Tool Invocation

When you invoke Tessent Shell, the tool starts up in setup mode.

You can invoke Tessent Shell from a Linux shell using the following syntax:

```
% tessent -shell
```

To use most Tessent Shell functionality, you must load a cell library after invocation using the `read_cell_library` command.

Refer to the **tessent** shell command description in the *Tessent Shell Reference Manual* for additional invocation options.

Tessent Startup File

During invocation, Tessent Shell reads the `.tessent_startup` startup file. You can use this startup file for both general and tool-specific settings. The default location for the startup file is the home directory: `$HOME/.tessent_startup`.

This startup file is common to the contexts listed in [Table 2-2](#) on page 33.

Tessent Shell Environment Variables

Tessent Shell provides environment variables for Tessent Shell environment operation in addition to application-specific environment variables. During invocation, the tool reads all environment variables that you have set.

The following table lists the environment variables that you can set for Tessent Shell environment operation.

Table 2-1. Application-Specific Environment Variables

Variable	Description
TESSENT_LICENSE_ORDER	Specifies the order that Tessent tools check out licenses. The list of licenses uses the same terminology accepted by the command “-license”.
TESSENT_STARTUP	Specifies the pathname of a directory that contains a Tessent tool startup file. Default: No default
TESSENT_TMP_LOCATION	Specifies the location at which the tool creates the <i>.tessent.tmp.hostname.process_id</i> scratch directory. For more information, see the command description in the <i>Tessent Shell Reference Manual</i> .
TESSENT_UNDERSCORE_COMMANDS_ONLY	Directs the tool to only enable commands that use the underscore style. When this environment variable is set to any value, the legacy commands that used spaces are disabled. For example, the tool would accept <code>analyze_drc_violation</code> but would not accept “analyze drc violation”.

Related Topics

[read_cell_library \[Tessent Shell Reference Manual\]](#)

[tessent \[Tessent Shell Reference Manual\]](#)

[Managing Mentor Tessent Software](#)

[set_context \[Tessent Shell Reference Manual\]](#)

[get_scratch_directory \[Tessent Shell Reference Manual\]](#)

Contexts and System Modes

One of the first tasks you perform after invoking Tessent Shell is setting the context and system mode for the current session.

The term “context” refers to a broad category of functionality that often corresponds to a specific point tool, product, or license feature, such as Tessent FastScan. Each context includes several system modes, which specifies the tool’s current state of operation. By setting the context and then the system mode, you indicate the type of tasks you want Tessent Shell to perform.

In addition, you must specify the design level at which you want Tessent Shell to execute tasks.

Contexts	33
System Modes	35
Context and System Mode Combinations	35
Design Levels	36

Contexts

The context specifies the functional category of tasks you want to perform with Tessent Shell.

[Table 2-2](#) lists the contexts that are currently available.

Table 2-2. Tessent Shell Contexts

Context	Description
dft	Editing and introspection of the following types of designs: gate-level Verilog, RTL Verilog, RTL System Verilog, and RTL VHDL.
dft -edt	EDT IP generation and optional insertion. This corresponds to the IP creation phase of Tessent TestKompress®.
dft -logic_bist -edt	Configuration, generation, and insertion of the EDT/LBIST hybrid controller IP. This corresponds to Tessent LogicBIST.
dft -no_sub_context	Specifies that a command is only available in the dft context when no sub-context, such as -scan and -edt, was specified. See the register_tcl_command in the <i>Tessent Shell Reference Manual</i> for more information.
dft -scan	Scan analysis and scan chain insertion. This corresponds to Tessent Scan. Additionally in this context you can prepare a BIST-ready design and include tasks such as X-bounding, MCP/FP handling, and test point insertion. These features correspond to Tessent ScanPro.

Table 2-2. Tessent Shell Contexts (cont.)

Context	Description
dft -test_points	Test point identification and insertion. The test point identification algorithm focuses on either pattern count reduction or improving random pattern testability of the design.
patterns -failure_mapping	Reverse map top-level failures to the core so that you can perform diagnosis with Tessent Diagnosis. Used after performing scan pattern retargeting within the patterns -scan_retargeting context.
patterns -ijtag	PDL command retargeting for IJTAG (IEEE 1687-2014) plus extraction of the ICL network from the design.
patterns -scan	Test pattern generation and good and fault simulation. This corresponds to Tessent FastScan and the test pattern generation phase of Tessent TestKompress. The patterns -scan context supports uncompressed scan ATPG/fault simulation, EDT ATPG/fault simulation, and Logic BIST fault simulation.
patterns -scan_diagnosis	Test failure diagnosis to determine a defect's failure mechanism and location. This corresponds to Tessent Diagnosis.
patterns -scan_retargeting	Scan pattern retargeting for retargeting core-level test patterns at the top level.
patterns -silicon_insight	Control, simulate, debug, and characterize BIST-related memories, logic, PLLs, and SerDes. This corresponds to Tessent SiliconInsight.

Context Specification

Set the Tessent Shell context using the [set_context](#) command. For example:

```
SETUP> set_context dft -scan
```

You must set the context after you invoke Tessent Shell and before you can enter most commands. The `set_context` command is available only in setup mode. You can use the [get_context](#) command to see the current context.

Prior to setting the context, you can only run a small set of setup commands. These commands are those that you would typically place inside the startup file.

Context and Licensing

When you set the context, the tool automatically acquires the appropriate license, if available. Alternatively, you can directly specify the license Tessent Shell acquires by using the [set_context](#) command with the optional `-license` switch.

You can control the order in which Tessent Shell checks out licenses by setting the `TESSENT_LICENSE_ORDER` environment variable. You provide, as the value of the environment variable, a space-separated list of licenses using the terminology described for “`set_context -license`.” For example:

```
setenv TESSENT_LICENSE_ORDER "Scan TestKompres IJTAG"
```

Any available licenses not explicitly listed in the value of the `TESSENT_LICENSE_ORDER` environment variable are appended to the list in their original order. The tool issues a warning if the environment variable contains a license that does not exist.

For more information about specifying a license feature, refer to the [set_context](#) command description in the *Tessent Shell Reference Manual*. For a complete list of license feature names, refer to [Managing Tessent Software](#).

System Modes

A system mode in Tessent Shell defines the operational state of the tool. The default system mode is `setup`. The available system mode depends on the current context of the tool.

[Table 2-3](#) lists the available system modes.

Table 2-3. Tessent Shell System Modes

System Mode	Description
setup	Used as the entry point into the tool. Used to define the current context and specify the design information.
analysis	Used to perform design analysis, test pattern generation, PDL retargeting, and simulation.
insertion	Used to perform design editing and introspection.

Change the Tessent Shell system mode using the [set_system_mode](#) command. For example:

```
SETUP> set_system_mode insertion
```

Context and System Mode Combinations

Together, the context and system modes imply a Tessent Shell task flow from `setup` to `analysis` to `insertion` when you are working with DFT designs, and from `setup` to `analysis` when you are working with patterns. For example, if you are working on a DFT design while in `setup` mode, you cannot perform scan analysis.

The following matrix lists the actions you can perform in the available contexts and system modes. The tool filters out the functionality that does not apply to the context/mode you are currently in and issues an error when you attempt to perform a task not within the scope of the current context/system mode.

Context	Setup Mode	Analysis Mode	Insertion Mode
dft	<ul style="list-style-type: none"> • Read and configure design • Prepare for design editing, IP creation, scan insertion, test point analysis and insertion 	<ul style="list-style-type: none"> • Scan analysis • Test point analysis • EDT IP creation • Hybrid TK/LBIST IP creation • MemoryBIST IP creation • Boundary scan IP creation • In-System Test IP creation 	<ul style="list-style-type: none"> • RTL or gate-level design editing with design introspection
patterns	<ul style="list-style-type: none"> • Read and configure design • Define test pattern generation type to perform 	<ul style="list-style-type: none"> • ATPG/fault simulation • PDL retargeting • Scan pattern retargeting • Execute an IJTAG test pattern • Perform SimDUT simulation 	(not applicable)

The tool provides a set of commands that enable you to interact with contexts and system modes.

Table 2-4. Tessent Shell Context and System Mode Commands

Command	Description
get_context	Returns the current context as specified by the set_context command. Also returns inferred subcontexts, such as whether patterns -scan is configured to perform LogicBIST or ATPG.
set_context	Sets the current context and its options.
set_system_mode	Sets the system mode.

Design Levels

When working with DFT designs—that is, you are working in context dft—you must set the design level at which you are performing tasks. There is no default setting for the design level.

The available design levels are chip, physical block, and sub-block. For flat designs, you always work at the chip level. For hierarchical designs, it is crucial to differentiate whether you are work at the top-level (chip) or at lower levels within physical blocks or sub-blocks. Refer to “[Hierarchical DFT Terminology](#)” for definitions of these terms.

For complete information, refer to the [set_design_level](#) command description in the *Tessent Shell Reference Manual*.

Design Data Models

Tessent Shell has the following data models: the hierarchical design data model, the flat design data model, and the ICL data model. Each of these data models contains one or more design objects, such as pins and modules, and each design object is associated with a set of attributes, such as an ID or bit-width of a bus.

For a complete description of the various data models, object types, and attributes, refer to the “[Data Models](#)” chapter in the *Tessent Shell Reference Manual*.


Flat Design Data Model	38
Hierarchical Design Data Model	38
ICL Data Model	40
Object Attributes	40

Flat Design Data Model

The flat design data model is an internal, flattened representation of a hierarchical design that the tool creates when entering analysis mode. You can also explicitly create the flat model using the `create_flat_model` command.

The flat design data model consists of gates that are connected together. A gate is an instance of a primitive module, and a `gate_pin` object represents a pin on a gate instance. `Gate_pin` objects have no unique instance name. Instead they have a unique ID that differentiates one from another. The format of the ID is two integers separated by a period character. The first integer represents the gate ID, and the second integer represents the pin index (where 0 is the output pin, 1 is first input pin, and so on). However, this ID does not remain in place from one netlist version to the next or from one Tessent Shell invocation to the next. For this reason, you should avoid hard-coding this ID into a script.

Note

 You can preserve hierarchical pins in the flat model by using the [set_attribute_options -preserve_boundary_in_flat_model](#) option.

For more information about the flat design data model and the `gate_pin` object type, refer to “[Flat Design Data Model](#)” in the *Tessent Shell Reference Manual*.

Hierarchical Design Data Model

Tessent Shell translates your design netlist into a hierarchical design data model in memory when you load the design into the tool using a command such as [read_verilog](#).

The design data remains in memory during the Tessent Shell session until you delete the model or exit the tool.

The hierarchical design data model contains the following object types:

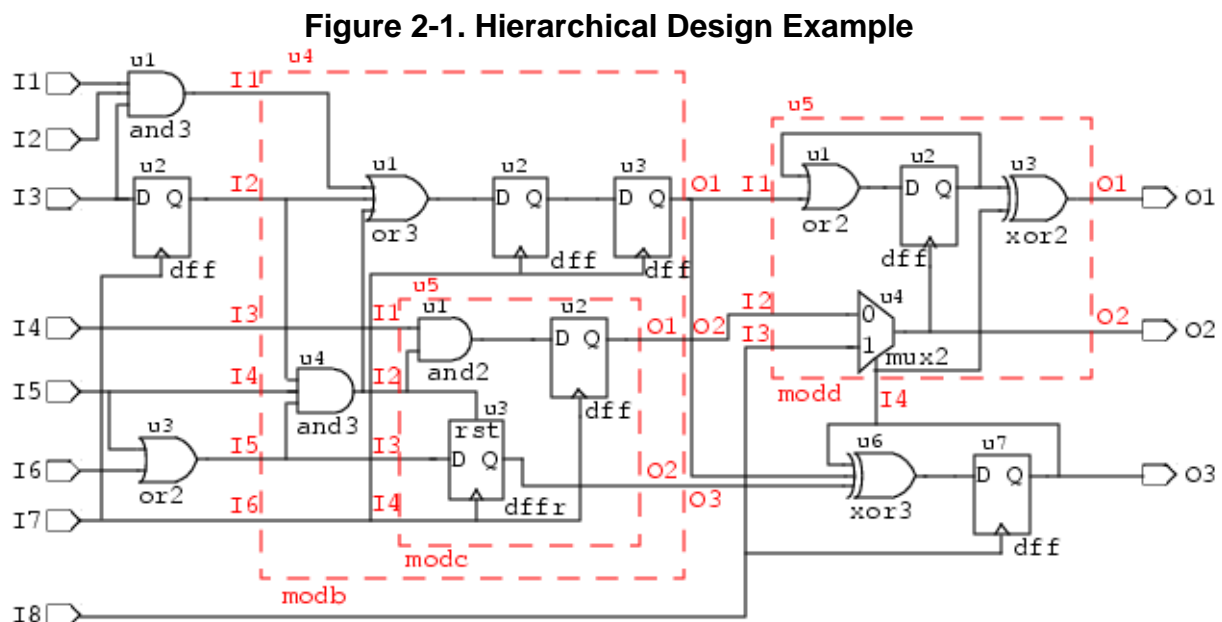
- Module — A basic building block of your design. A module can be a Verilog module, a Tessent library model, or a built-in primitive.
- Instance — A single instantiation of a module.
- Port — An input, output, or inout interface of a module.
- Pin — An input or output interface of an instance.
- Net — A wire that connects the pins of instances.
- Pseudo_port — Represents a user-added primary input or primary output.

These object types are described in detail in the “[Data Models](#)” chapter of the *Tessent Shell Reference Manual*.

Use the [set_current_design](#) command to designate one module within your design as the current design. Instances, pins, and nets are hierarchical objects defined relative to the current design.

[Figure 2-1](#) shows a hierarchical design upon which many of the examples in this chapter are based. The label above the elements is the instance name while the label below the elements is the module name. For example, the module name of the AND gate in the upper-left corner is `and3`, and its instance name is `u1`.

The dashed boxes are modules instantiated in the parent module. For example, module `modc` is instantiated inside of module `modb`. Its complete instance path is `u4/u5`.



ICL Data Model

The Instrument Connectivity Language (ICL) describes the elements that comprise the IJTAG network as well as their logical (though not necessarily physical) connections to each other and to the instruments at the endpoints of the network.

ICL bears a loose resemblance to a hierarchical netlist such as Verilog; it is organized by modules that may contain instances of other modules, and it describes the connections between the pins of the instances. ICL is not a complete netlist. The connections are port-to-port rather than through nets, and ICL uses abstraction rather than include the detailed physical construction of the circuitry. ICL represents only the behavioral operation of the network.

The ICL data model defines objects as one of various object types, such as `icl_module`, `icl_instance`, `icl_port`, and `icl_pin`.

Each `icl_module` object has its list of objects including the following:

- `icl_port`
- `icl_instance_in_module`
- `icl_pin_in_module`
- `icl_scan_register_in_module`
- `icl_scan_mux_in_module`
- `icl_one_hot_scan_group_in_module`
- `icl_alias_in_module`
- `icl_scan_interface_of_module`

Once the current design is set using the [set_current_design](#) command, the ICL data model is elaborated downward from the current design and `icl_instance` objects are created for each instance of `icl_module`, and `icl_pin` objects are created for each port of the modules associated to the `icl_instances`. The `icl_instance` and `icl_pin` objects are hierarchical objects and therefore only exist after the current design is set. The same holds for objects of type `icl_scan_register`, `icl_scan_mux`, `icl_one_hot_scan_group`, `icl_alias`, and `icl_scan_interface`.

For more information about the ICL data model and the `icl_module`, `icl_instance`, `icl_port`, and `icl_pin` object types, refer to “[ICL Data Model](#)” in the *Tessent Shell Reference Manual*.

Object Attributes

Each design object in a design data model has a list of characteristics, called attributes, attached to that object. For example, each pin has an attribute that specifies its hierarchical name and its parent instance. There are both predefined and user-defined attributes.

Predefined attributes provide access to information known to the tool such as the name of a module or the direction of a pin. All design objects have some predefined attributes, and every design object can have user-defined attributes as well. For a complete list of attributes for each type of data model, refer to “[Data Models](#)” chapter in the *Tessent Shell Reference Manual*.

The process for creating a new user-defined attribute is called *registration*. The predefined attributes, unlike the user-defined attributes, do not need to be registered.

You must register each new user-defined attribute and specify a default value before you can use the attribute. If you want to later change the default value, you must unregister and then re-register the attribute. For more information about the registration process, refer to the description of the [register_attribute](#) command in the *Tessent Shell Reference Manual*.

You can query any attribute and change any user-defined attribute. Most predefined attributes are read-only, although some are read-write, such as the `is_hard_module` attribute.

You can filter and sort objects based on the attributes and their values. The `filter_collection` and `sort_collection` commands perform these operations. For more information about filtering attributes, refer to “[Attribute Filtering Equation Syntax](#)” in the *Tessent Shell Reference Manual*.

Intuitively named “`get_`” commands return collections of objects from the data model and the model’s attributes that you can use for design introspection of various design objects.

Chapter 3

Design Introspection and Editing

Tessent Shell provides a full range of commands for examining (introspecting) and editing designs.

Along with the command-line interface, Tessent Shell provides a graphical user interface, Tessent Visualizer, that enables you to edit and introspect designs, and adjust object attributes. For details, refer to “[Tessent Visualizer](#)” on page 625.

Design Introspection	44
Design Editing	77
Simulation Contexts	85
Automatic Design Mapping	89
ICL Objects vs. Design Objects Introspection	92

Design Introspection

Tessent Shell introspection commands allow you to examine the design objects within a design data model. They provide access to the tool’s internal data structures, giving you the flexibility of Tcl scripting while keeping all compute-intensive processing in the tool’s backend. The commands operate on object specifications that you include as arguments to the commands, and they return collections.

Object Specification Format	44
Collections	44
Design Introspection Examples	47
Design Introspection Command Summary	51
Bundle Object Introspection	53

Object Specification Format

An object specification lists the design objects (such as instances, pins, or nets) that the specified command acts upon. The object specification can be the name of an object, a Tcl list of names of objects, or a collection of objects. For design objects that have unique IDs, like instances, you can use the IDs as the object names.

The following examples show the `add_schematic_objects` command with valid object specifications listed within square brackets or curly brackets.

```
add_schematic_objects [get_instances u*] -display hierarchical_schematic
```

```
add_schematic_objects [list u1 u2 u3] -display hierarchical_schematic
```

```
add_schematic_objects {u1 u2 u3} -display hierarchical_schematic
```

These commands display the results in Tessent Visualizer.

All Tessent Shell commands operating on user-specified objects accept object specifications as arguments.

Collections

Collections are an extension of Tcl that are specific to Tessent Shell applications. Native Tcl commands such as “foreach” and “puts” do not recognize collections. A collection represents a group of zero (an empty set) or more design objects.

Design introspection commands return collections of objects. The objects are stored in the internal data structures, and the tool returns a string handle to this collection. The string handle is a “@” character followed by a numeric ID (in this case, “@1”). The entire data volume remains in the tool’s internal data structures, so the Tcl interface is not overloaded with large amounts of data.

Introspection commands issued directly from the shell display the “name” attribute of the first 50 elements of the collection, because the name is more useful than displaying the collection’s ID. However, names are not displayed in non-interactive mode such as when executing a dofile.

The following example demonstrates the use of the [get_instances](#) and [get_name_list](#) commands to return collections of objects:

```
SETUP>set var1 [get_instances u* -hierarchical -of_modules MOD1]
{u1 u2 u3 u4 u5 u6 u7}

SETUP>puts $var1@1
SETUP>puts [get_name_list $var1]

u1 u2 u3 u4 u5 u6 u7
```

In this example, the first command returns a collection of names. The objects are stored in internal data structures, and the tool returns a string handle to this collection which is “@” followed by a number ID (“@1”).

In the following example, the collection created by the [get_instances](#) command is stored in the variable `instCollection` (which means that the collection is referenced).

```
set instCollection [get_instances -of_type cell]
```

Tessent Shell deletes the collection when you unset the variable `instCollection` or set the variable to a new value, or when the Tcl variable(s) referring to the collection go out of scope.

In the following example, the collection created by the command [get_pins](#) is passed to the [get_attribute_value_list](#) command. This means that the collection is referenced.

```
get_attribute_value_list [get_pins u1/a*] -name direction
```

Tessent Shell deletes the collection when the command [get_attribute_value_list](#) returns a value.

Collections can refer to objects that no longer exist because they have been deleted using editing commands, such as [delete_pins](#). The built-in attribute “`is_valid`” is set to false when an object has been deleted. All commands that accept collection pointers as input automatically ignore objects with the “`is_valid`” attribute set to false.

Persistence of Collections

The tool references a collection when the collection is stored in a Tcl variable or when it is passed to a command or procedure. The tool automatically deletes a collection when it is no longer referenced.

You should not use Tcl built-in commands that expect a Tcl list, such as the `foreach` command, with collections. When you pass a collection to this type of command, the command dereferences the collection and, as a result, deletes the collection.

How to Transcript the Contents of Collections

Use the [get_name_list](#) command to return a list of the names of all the elements in the collection. So, to transcript the names in the log file, you can use the “puts” command with [get_name_list](#) (or any other Tessent [get_*](#) command):

Commands That Work With Collections or Tcl Lists

Tessent Shell provides commands that create, manipulate, and query collections. [Table 3-1](#) presents some Tessent Shell commands based on how they interact with collections.¹

Table 3-1. Commands That Interact With Collections

Commands that...	Command Name
Create collections or Tcl lists	get_attribute_list
	get_attribute_option
	get_attribute_value_list
	get_current_design
	get_fanins
	get_fanouts
	get_gate_pins
	get_instances
	get_modules
	get_name_list
	get_nets
	get_pins
	get_ports
Operate on collections	add_to_collection
	append_to_collection
	compare_collections
	copy_collection
	filter_collection
	foreach_in_collection
	index_collection
	remove_from_collection
sort_collection	

1. The table does not list all the applicable commands. Refer to the *Tessent Shell Reference Manual*.

Table 3-1. Commands That Interact With Collections (cont.)

Commands that...	Command Name
Read and write attributes of objects found within collections or Tcl lists	get_attribute_list
	get_attribute_value_list
	report_attributes
	set_attribute_value

Design Introspection Examples

Design introspection enables you to create procedures to show design data of interest, create and set attributes, create and manipulate collections, create custom reports, and other tasks.

Example of Creating a Procedure to Show Flip-Flop Fanouts

The following example is based on [Figure 2-1](#) on page 39 and creates a procedure called `show_fanouts` that returns the fanouts of a specified flip-flop.

```
proc show_fanouts {flop} {
    set ff_fanout [get_fanouts $flop/Q]
    puts "$flop/Q is connected to: [get_name_list $ff_fanout]"
}
```

The following two examples show how to use the `show_fanouts` procedure you just created.

> show_fanouts u2

```
u2/Q is connected to : u4/u4/A0 u4/u1/A1
```

**> foreach_in_collection ff [get_instances * -of_module dff* -hierarchical] \
{show_fanouts [get_name_list \$ff]}**

```
u2/Q is connected to: u4/u1/A1 u4/u4/A0
u7/Q is connected to: O3 u6/A0 u5/u3/A1 u5/u4/S0
u4/u2/Q is connected to: u4/u3/D
u4/u3/Q is connected to: u6/A1 u5/u1/A1
u4/u5/u2/Q is connected to: u5/u4/A0
u5/u2/Q is connected to: u5/u1/A0 u5/u3/A0
u4/u5/u3/Q is connected to: u6/A2
```

Example of Displaying the Number of Input Ports

The following example displays the number of input ports on modules (objects with names that begin with “mod” in [Figure 2-1](#)).

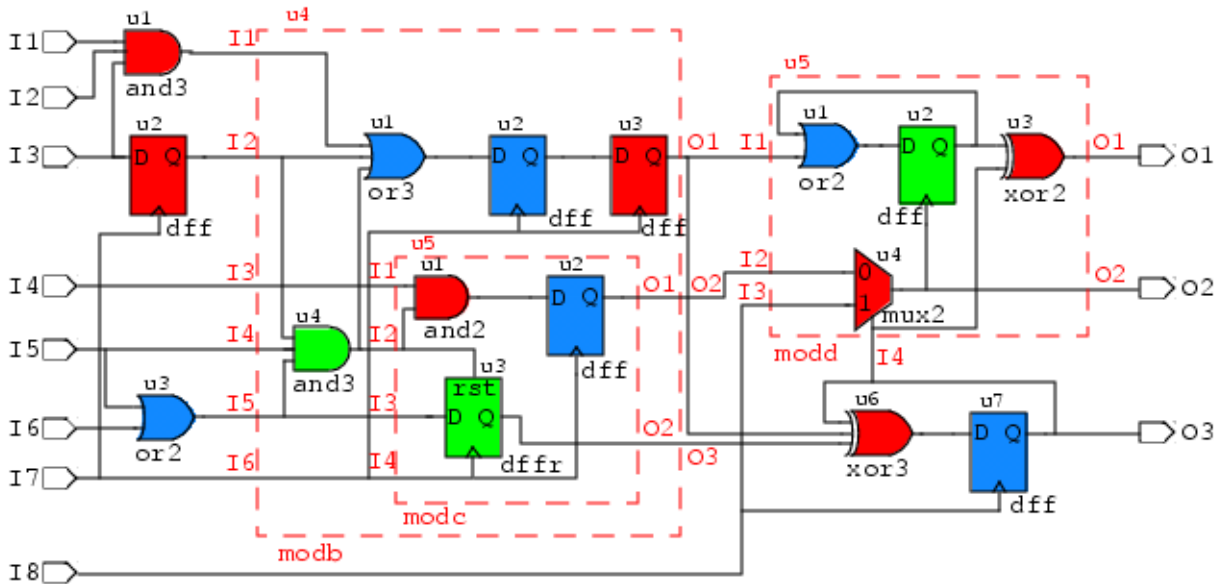
> set mods [get_modules mod* -of_type design]

**> foreach_in_collection itr \$mods \{puts "The number of input ports of \
[get_attribute_value_list \$itr -name name] \
is [sizeof_collection [get_ports -of_module \$itr -direction input]]"}**

The number of input ports of modb is 6
 The number of input ports of modc is 4
 The number of input ports of modd is 4

The design in [Figure 3-1](#) is identical to [Figure 2-1](#), with the addition of colors to help you understand the examples that follow.

Figure 3-1. Hierarchical Design Example With Colors



Example of Creating Attributes and Manipulating Collections

The following example is based on [Figure 3-1](#) and shows the complete process of creating and setting user-defined attributes, and then creating and manipulating collections.

```
register_attribute -name color -value_type string -default "blue" -enum "red green blue" \  
  -object_types instance
```

Set the “color” attribute of each instance to match [Figure 3-1](#).

```
set_attribute_value {/u4/u4 /u4/u5/u3 /u5/u2} -name color -value "green"
```

```
{u4/u4 u4/u5/u3 u5/u2}
```

```
set_attribute_value {u1 u2 u6 /u4/u5/u1 /u4/u3 /u5/u4 /u5/u3} -name color -value "red"
```

```
{u1 u2 u6 u4/u5/u1 u4/u3 u5/u4 u5/u3}
```

```
set_attribute_value {u3 u7 /u4/u1 /u4/u2 /u4/u5/u2 /u5/u1} -name color -value "blue"
```

```
{u3 u7 u4/u1 u4/u2 u4/u5/u2 u5/u1}
```

Create collections of instances with the same color values.

```
set_all_inst [get_instances -of_type cell]
```



```
{u1 u2 u3 u6 u7 u4/u1 u4/u2 u4/u3 u4/u4 u4/u5/u1 u4/u5/u2 u4/u5/u3 u5/u1
u5/u2 u5/u3 u5/u4}
```

```
set red_inst [filter_collection $all_inst {color == "red"}]
```

```
{u1 u2 u6 u4/u3 u4/u5/u1 u5/u3 u5/u4}
```

```
set blue_inst [filter_collection $all_inst {color == "blue"}]
```

```
{u3 u7 u4/u1 u4/u2 u4/u5/u2 u5/u1}
```

```
set green_inst [filter_collection $all_inst {color == "green"}]
```

```
{u4/u4 u4/u5/u3 u5/u2}
```

Manipulate the collections and create additional collections.

```
puts "Number of red cells in the design: [sizeof_collection $red_inst]"
```

```
Number of red cells in the design: 7
```

```
puts [compare_collections $red_inst $blue_inst]
```

```
1
```

```
set red_green [add_to_collection $red_inst $green_inst]
```

```
{u1 u2 u6 u4/u3 u4/u5/u1 u5/u3 u5/u4 u4/u4 u4/u5/u3 u5/u2}
```

```
set sort_red [sort_collection $red_inst name -descending]
```

```
{u6 u5/u4 u5/u3 u4/u5/u1 u4/u3 u2 u1}
```

Example of Displaying Attribute Values of a Collection

After Tessent Shell has executed the previous commands, the following commands display the colors assigned to gates in the collection named ANDs.

```
set ANDs ""
```

```
append_to_collection ANDs [get_instances -of_type cell -filter {module_name == "AND"}]
```

```
foreach_in_collection itr $ANDs {puts "The color value for \  
[get_attribute_value_list $itr -name name] is [get_attribute_value_list $itr -name color] "}
```

```
The color value for u1 is red
The color value for u4/u4 is green
The color value for u4/u5/u1 is red
```

As an alternate way to register the attribute in the previous example, you can use the `register_attribute` command as shown here:

```
register_attribute -name is_red -value_type bool -object_types "instance" \  
-description "True if color is red."
```

So, instead of using the color attribute with enumerated values of red, green, and blue as shown previously, you could instead register three Boolean type attributes of `is_red`, `is_green`, and `is_blue`. This enables you to use Boolean values for filtering and tracing. For example:

```
> set red_inst [filter_collection $all_inst {is_red}]
```

Example of Changing an Attribute of a Collection

The following command example is based on [Figure 3-1](#) and shows how to change the color attribute of a collection of DFFs to green.

```
set DFFs [get_instances -of_modules dff]

# display all of the instances that have a color attribute of blue

set all_blue [filter_collection $all_inst "color == blue"]
{u7 u4/u2 u4/u5/u2}

# change all instances in the collection DFFs to have a "color" attribute value of green

set_attribute_value $DFFs -name color -value green

# now check the results
set all_green [ filter_collection [$all_inst {color == "green"} ]
{u2 u7 u5/u2 u4/u4 u4/u2 u4/u3 u4/u5/u2 u4/u5/u3}
```

Example of Removing Duplicate Objects from a Collection

The following example shows how to create a collection containing three duplicate objects and then remove the duplicate objects from the collection.

```
set t [get_instances {u3 u3 u3}]
{u3 u3 u3}

sizeof_collection $t
3

set tt [add_to_collection "" $t -unique]
{u3}

sizeof_collection $tt
1
```

Example of Creating a Custom Report

The following dofile example creates a collection of all instance objects with a leaf name starting with “u” and then displays ten instance names per row.

```

set cnt 0
set line ""
foreach_in_collection element [get_instances u* -hierarchical] {
  if {$cnt < 10} {
    append line "[get_single_name $element] "
    incr cnt
  } else {
    puts $line
    set line ""
    append line "[get_single_name $element] "
    set cnt 1
  }
}
if {$cnt > 0} {puts $line}

```

Design Introspection Command Summary

The design introspection commands include the `get_` commands as well as various commands for manipulating attributes.

Design Introspection Command Summary

[Table 3-2](#) lists the common design introspection commands. Refer to the *Tessent Shell Reference Manual* for more information.

Table 3-2. Design Introspection Commands

Command Name	Description
get_common_parent_instance	Returns the common ancestor of all instances, pins, or nets found in <code>object_spec</code> .
get_current_design	Returns a collection of one element that specifies the top-level design when previously set.
get_design_level	Returns the design level previously defined with the <code>set_design_level</code> command.
get_fanins	Returns a collection of all objects found in the fanin of the specified pin, net, or port objects.
get_fanouts	Returns a collection of all requested objects found in the fanout of the specified pin, net, or port object.
get_gate_pins	Returns a collection of all <code>gate_pin</code> objects found below the current design in the flat model.
get_instances	Returns a collection of instances relative to the current design.
get_modules	Returns a collection of modules.
get_nets	Returns a collection of nets relative to the current design.
get_pins	Returns a collection of pins relative to the current design.

Table 3-2. Design Introspection Commands (cont.)

Command Name	Description
get_ports	Returns a collection of ports on a given module.

Attribute Introspection Command Summary

Table 3-3 lists all of the commands that introspect attributes.

Table 3-3. Attribute Introspection Commands

Command Name	Description
get_attribute_list	Lists registered attributes.
get_attribute_option	Returns the current setting of an attribute's option.
get_attribute_value_list	Returns the attribute's value.
get_name_list	Returns the name attribute of the specified objects.
get_single_name	Returns a string with the name of the element specified by the <code>object_spec</code> argument.
register_attribute	Registers a new attribute by adding it to the attribute manager.
report_attributes	Prints a report for the registered attributes.
reset_attribute_value	Resets the attribute to its default value.
set_attribute_options	Configures attribute options.
set_attribute_value	Defines the attribute's value.
unregister_attribute	Makes an attribute unusable by removing it from the attribute manager.

Bundle Object Introspection

The Bundle Object Hierarchy is an extension of the module instance hierarchy, and the signals (pins, ports, and nets) are considered as a tree of sub signals. The root bundle is the signal itself, the leaf objects are the bit-blasted elements of the given signal, and the nodes are the middle bundle objects.

To support this bundle object concept new object types are added to [Tessent Shell Hierarchical Design Data Model](#). The leaf objects are represented by net, pin, port object types; the sub and root bundle objects are represented by net_bundle, pin_bundle, and port_bundle objects types.

The bundle object is set of bits that represents a sub signal of a given net or port. This concept is applied to complex and simple signals for RTL and no RTL flow. The difference between simple and complex signals is in the number of levels that can be explored through the introspection commands.

Each complex signal has a select part. For example, for the following complex signal:

`B [5] .c [2] .d`

The portion in red ([5].c[2].d) is the select part, and the “B” is the root bundle. See “[Bundle Object](#)” on page 53 for complete details. The select part of the signals helps to explore the Bundle Object Hierarchy level-by-level.

Tessent Shell supports introspection of RTL complex signals through a bundle object, which is a set of bits that represent a sub signal of a given RTL net or port. Using Tessent Shell introspection commands enable exploration inside the bundle object and selection of different bundle parts of the signals using the hierarchical bitselect.

Bundle Object.	53
Bundle Object Data Model	58
Introspection.	61
Bundle Object Introspection Examples	65
Fanin and Fanout Tracing of Complex Signals and Bundle Object Tracing.	68
Connectivity Through Complex Signals	72

Bundle Object

A bundle object is a set of bits that represents a whole signal or its sub signals of a given RTL port or net.

The bundle is made up of Tessent Shell [Hierarchical Design Data Model](#) object types that you introspect using Tessent Shell introspection commands relative to the current design. The following illustrates this concept:

```
Root bundle (Signal itself)
  |--> Node (Sub Signal)
  |   |--> Leaf (Bit-blasted elements)

Example:net_bundle (Bundle Object Type)
  |--> net_bundle (Bundle Object Type)
  |   |--> net (Bit-blasted Object Type)

pin_bundle (Bundle Object Type)
  |--> pin_bundle (Bundle Object Type)
  |   |--> pin (Bit-blasted Object Type)

port_bundle (Bundle Object Type)
  |--> port_bundle (Bundle Object Type)
  |   |--> port (Bit-blasted Object Type)
```

The following object types contain several common attributes that are also associated with the bit-blasted object types (net, pin, and port):

- [net_bundle](#)
- [pin_bundle](#)
- [port_bundle](#)

All attributes added to port, pin, and net are also added to port_bundle, pin_bundle, and net_bundle.

For example, the naming is common between a bit-blasted object type and a bundle object type.

See “[Bundle Object Data Model](#)” on page 58 for complete information.

Naming Attributes

The following table shows the list of the principal name attributes associated with net, port, and pin object types.

Attribute	Description	RTL View	Synthesized View
name	The active name. The name depends on the selected view. This includes the select part in the case of signals (port, pin, and net). The -rtl mode has the RTL view and the synthesized view. The -no_rtl mode only has the netlist view.	The full RTL name including all the select part.	The full synthesis name including all the select part (flatten position in case of complex signals).
parent_bundle_name	The bundle parent of the port, pin, or net.	The parent name of the selected RTL object.	The parent name of the selected synthesized object.
post_synthesis_name	The synthesis name, created by the quick synthesis.	The computed equivalent post synthesis name.	Equal to “name” attribute.
pre_synthesis_name	RTL name, the original name before synthesis.	Equal to “name” attribute.	Full RTL name, the original name before synthesis.
root_bundle_name	The active name without the select part.	The base RTL name without the select part.	The base synthesis name without the select part.

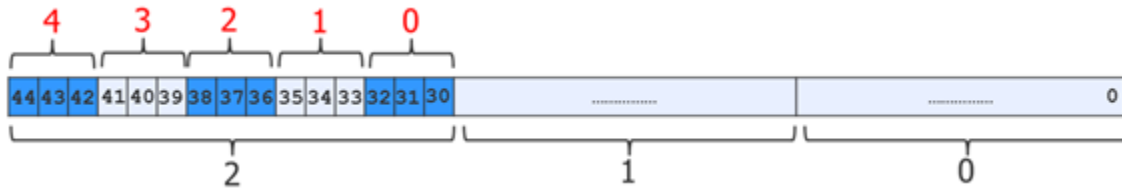
Bundle Naming Examples

Example 1

The `post_synthesis_name` attribute for complex signals is usually the base name with the flatten position as the index. System Verilog interface signals are the exception. For example, in case of the following input port “data_in” declared as array of structure in System Verilog:

```
typedef struct{
    logic a;
    logic [1:0] b;
} data_t;
input data_t [2:0] [4:0] data_in;
```

The following figure illustrates the representation in the flatten dimension:



Name Attribute (RTL View)	post_synthesis_name Attribute
data_in[2][1].a	data_in[35]
data_in[2][1].b	data_in[34:33]
data_in[2][1].b[1]	data_in[34]

The following table shows the different naming of the signal “data_in[2][1].b[1]”

Attribute	RTL View	Synthesis View
name	data_in[2][1].b[1]	data_in[34]
pre_synthesis_name	data_in[2][1].b[1]	data_in[2][1].b[1]
post_synthesis_name	data_in[34]	data_in[34]
root_bundle_name	data_in	data_in

Example 2

In this example, the “net1” from Example 1 is declared inside nested generate blocks:

```
typedef struct{
    logic a;
    logic [1:0] b;
} data_t;

genvar i, j;
generate
    for (i=0; i < 2; i=i+1) begin
        for (j=0; j < 2; j=j+1) begin : MEM_j
            data_t [2:0] [4:0] net1;
            .....
        end
    end
endgenerate
```

The following table shows the different naming of “genblk1[0].MEM_j[0].net1[2][1].b[1]”

Attribute	RTL View	Synthesis View
name	genblk1[0].MEM_j[0].net1[2][1].b[1]	\genblk1[0].MEM_j[0].net1 [34]

Attribute	RTL View	Synthesis View
pre_synthesis_name	genblk1[0].MEM_j[0].net1[2][1].b[1]	genblk1[0].MEM_j[0].net1[2][1].b[1]
post_synthesis_name	\genblk1[0].MEM_j[0].net1 [34]	\genblk1[0].MEM_j[0].net1 [34]
root_bundle_name	genblk1[0].MEM_j[0].net1	\genblk1[0].MEM_j[0].net1

Example 3

This example illustrates an interface declared as a net to connect two sub instances.

```
interface intf;
  logic a;
  logic b;
  modport in (input a, output b);
  modport out (input b, output a);
endinterface

module top;
  intf i ();
  u_a m1 (.i1(i));
  u_b m2 (.i2(i));
endmodule

module u_a(intf.in i1);
endmodule
module u_b(intf.out i2);
endmodule
```

The following tables shows the different naming of the signals “i.a” and “m1/i1.a”.

For net: “i.a”

Attribute	RTL View	Synthesized View
name	i.a	\i.a
object_type	net	net
pre_synthesis_name	i.a	i.a
post_synthesis_name	\i.a	\i.a
root_bundle_name	i	i

For pin: “m1/i1.a”

Attribute	RTL View	Synthesized View
name	m1/i1.a	m1/i1.a
object_type	pin	pin
pre_synthesis_name	m1/i1.a	m1/i1.a
post_synthesis_name	m1/i1.a	m1/i1.a

Attribute	RTL View	Synthesized View
root_bundle_name	m1/i1	m1/i1.a

Example 4

The values of the name attributes are language independent and have the same values for both VHDL and Verilog RTL objects. The following example in VHDL uses the record datatype:

```

library ieee;
package record_pkg is

    -- Outputs from the FIFO.
    type t_FROM_FIFO is record
        wr_full  : std_logic;           -- FIFO Full Flag
        rd_empty : std_logic;           -- FIFO Empty Flag
        rd_dv    : std_logic;
        rd_data  : std_logic_vector(7 downto 0);
    end record t_FROM_FIFO;
    .....
end package record_pkg;
use work.record_pkg.all; -- USING PACKAGE HERE!
entity top is
    port (
        i_clk  : in  std_logic;
        i_fifo : in  t_FROM_FIFO;
        ..... );
end top;

architecture behave of top is

begin
    .....
end behave;

```

The following table shows the different naming of the port “i_fifo.rd_data(5)” in VHDL. VHDL uses “()” to represent the index, but Tessent Shell uses “[]” instead.

Attribute	RTL View	Synthesis View
name	i_fifo.rd_data[5]	i_fifo[5]
pre_synthesis_name	i_fifo.rd_data[5]	i_fifo.rd_data[5]
post_synthesis_name	i_fifo[5]	i_fifo[5]
root_bundle_name	i_fifo	i_fifo

Bundle Object Data Model

Bundle objects utilize the Tessent Shell Hierarchical Design Data Model object types.

For each object type, there are attributes associated with the design objects found on, in, and below the current design (set with the [set_current_design](#) Tessent Shell command). A bundle object has a select part and type information. Otherwise a bit-blasted object has a bit select and also type information. An object type can be a complex type in case of bundle object, like struct, enum, typedef, multi-dimensional array. In the case of a bit-blasted object it is always a 1 bit datatype. It can be register, wire, logic, bit, tri, for example. Several attributes are associated with the following Hierarchical Design Data Model object types:

- [Net](#)
- [Net Bundle](#)
- [Pin](#)
- [Pin Bundle](#)
- [Port](#)
- [Port Bundle](#)

Each of these object types is covered in detail in the [Tessent Shell Reference Manual](#).

base_class and base_type Attributes

Each bundle object type contains a base_class and base_type attribute as follows:

- **base_class** — String character that represents the class category of the base type. It helps to complete the base_type definition. Possible values are: = 4_state_signed | 4_state_unsigned | 2_state_signed | 2_state_unsigned | enum | struct_packed | struct_unpacked | union_backed | union_unpacked | interface | interface_modport | other.
- **base_type** — String character that represents the base type of the selected object. The base type is the type of the object without any dimensions. It represents the base type of the sub bundle type if the selected object is a sub bundle object. It can be a base type of the root type if the selected object is a root bundle. The base_type is a key or user name that help to understand the base type of this bundle. Possible values are: logic | register | wire | bit | supply0 | supply1 | tri | triand | trior | tri0 | tri1 | wand | wor | byte | shortint | integer | longint | *user_name* | other

The following examples shows the values for base_class and base_type bundle object type attributes in System Verilog:

Table 3-4. base_class and base_type System Verilog Examples

Examples	base_type and base_class Values
Packed struct having typedef name “data_t”	base_type= data_t, base_class= struct_packed
Unpacked struct having typedef name “data_t”	base_type= data_t, base_class= struct_unpacked

Table 3-4. base_class and base_type System Verilog Examples (cont.)

Examples	base_type and base_class Values
shortint	base_type = shortint, base_class = 2_state_signed
unsigned shortint	base_type = shortint, base_class = 2_state_unsigned
integer	base_type = integer, base_class = 4_state_signed
int	base_type = integer, base_class = 2_state_signed
unsigned integer	base_type = integer, base_class = 4_state_unsigned
unsigned int	base_type = integer, base_class = 2_state_unsigned
reg	base_type = register, base_class = 4_state_unsigned
logic	base_type= logic, base_class= 4_state_unsigned
bit	base_type= logic, base_class= 2_state_unsigned
supply0, supply1	base_type= supply0 or supply1, base_class = 2_state_unsigned
tri, wand, triand, wor, trior, tri0, tri1	base_type = tri, wand, triand.... For all those datatypes, the base_class=4-state_unsigned
module top (INTF1 Bus)	bundle “Bus” has as base_type=INTF1, base_class=interface
module top (INTF1.secondary Bus)	bundle “Bus” has as base_type=INTF1.secondary, base_class=interface_modport

Net Bundle

The Net Bundle object type has net_bundle datatype and attributes. For complete details, refer to [net_bundle](#) in the *Tessent Shell Reference Manual*.

Pin Bundle

The Pin Bundle object type has pin_bundle datatype and attributes. For complete details, refer to [pin_bundle](#) in the *Tessent Shell Reference Manual*.

Port Bundle

The Port Bundle object type has port_bundle datatype and attributes. For complete details, refer to [port_bundle](#) in the *Tessent Shell Reference Manual*.

ICL Data Model

The ICL data model is a separate data model that reflects the design logic. The ICL data model defines objects as one of the following four data types: icl_module, icl_instance, icl_port, and icl_pin.

When using bundle objects, the naming style may be different between the ICL data model and the design logic data model. For this reason, bundle object-specific attributes are required to bind the ICL objects (`icl_module` and `icl_port`) to design logic (module and port).

`icl_module`

The following `icl_module` built-in attribute applies to design objects:

- **`tessent_design_module`** — The attribute reflects the name of the design module that corresponds to the ICL module at the time of ICL extraction.

`icl_port`

The following `icl_port` built-in attribute applies to design objects:

- **`tessent_design_gate_ports`** — The attribute reflects the `post_synthesis_name` of the design port corresponding to the ICL port at the time of ICL extraction.
- **`tessent_design_rtl_ports`** — An attribute that specifies the `pre_synthesis_name` of the design port corresponding to the ICL port at the time of ICL extraction.

Introspection


Using bundle object types, Tessent Shell enables you to introspect and explore any kind of signal within the design data model.

Both simple datatypes and complex datatypes can be explored using the introspection commands listed in “[Complex Signal Introspection Commands](#)” on page 65.

Any signal objects (ports, pins, and nets) are considered as a tree of sub signals. The given tree is referred to as a Bundle Object Hierarchy. The root bundle is the signal; the leaf objects are the bit-blasted elements of the given signal; the root, leaf, and middle bundle are all nodes of the tree. The middle bundles and the leaf objects are the sub bundle objects. To support this concept there are object types in the Hierarchical Design Data Model. The leaf objects are represented by `net`, `pin`, and `port` object types, the root and middle bundles are represented by `net_bundle`, `pin_bundle`, and `port_bundle` objects types. See “[Bundle Object Data Model](#)” on page 58.

The concept of bundle object type is applied to complex and simple signals for RTL and no RTL flow. The difference between simple and complex signals is in the number of levels that can be explored through the introspection commands. The select part of the signals helps to explore the Bundle Object Hierarchy level by level.

Tip

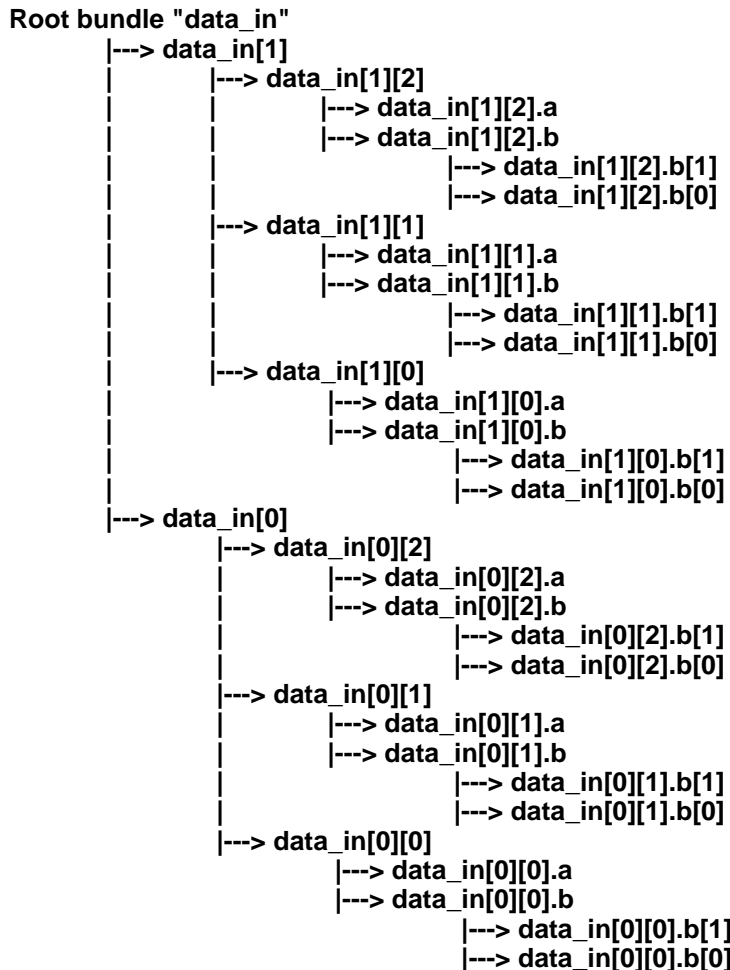
-  When performing introspection of complex signals, see also “[Object Specification Format](#)” on page 44 and “[Collections](#)” on page 44.
-

How Complex Signal Introspection Works

Consider the bundle hierarchy of the “data_in” port as follows:

```
typedef struct{
    logic a;
    logic [1:0] b;
} data_t;
input data_t [1:0][2:0] data_in;
```

The following shows the hierarchy of the “data_in” port in the Tessent Shell design data model.



Referring to the figure, using the *name_patterns* argument of the introspection commands, the tool splits each pattern into the different levels so that a local sub-pattern is applied to each local level.

For the bundle part, the characters “.” and “[<index>]” enable access to sub bundle hierarchy, “:” to access field sub bundle objects of struct or SVinterface and “[<index>]” to access array element sub bundle objects. For example, the pattern “data_in[0][1].b[*]” is really five sub-patterns; “data_in”, “[0]”, “[1]”, “.b”, and “[*]”, representing five levels of bundle hierarchy.

For simple wildcard patterns, the '*' is used the exact same way as it is used in instance path matching. It does not cross hierarchy. When specified within a field name it does not cross the "." that precedes or follows the field. When applied within [], it only matches elements within the []. In other words delimiters must be explicit.

For regular expression patterns, an individual regular expression only matches elements within the sub-pattern in which they are found. Again, it does not cross the adjacent delimiters.

Without the -regexp switch in the introspection command, the tool treats the bundle hierarchy delimiters ("[]" and ".") as normal delimiters and serves to break the pattern into multiple sub-patterns. When you include the -regexp switch in the introspection command, the tool treats the bundle hierarchy delimiters as regular expression meta-characters (unless you have escaped those delimiters). That is, in regexp mode the delimiters must be explicit and, unlike simple wildcard patterns, you must escape them.

For example, to match "data_in[1]", you need to use "data_in\[.*\]". The sequence '.*' matches any character zero or more times. For more details, see the [Example With regexp](#).

See "[Glob and Regular Expression Pattern Matching Syntax](#)" in the *Tessent Shell Reference Manual* for complete information on pattern matching.

Note



The examples use the "get_ports ... -bundle" command. For "get_nets ... -bundle" and "get_pins ... -bundle", the behavior is the same.

Examples

These examples use the "data_in" hierarchy described in the preceding.

```
SETUP> get_ports data_* -bundle
```

```
data_in
```

```
SETUP> get_ports data_in* -bundle
```

```
data_in
```

```
SETUP> get_ports data_in[*] -bundle
```

```
{data_in[1]} {data_in[0]}
```

```
SETUP> get_ports data_in[0][*] -bundle
```

```
{data_in[0][2]} {data_in[0][1]} {data_in[0][0]}
```

```
SETUP> get_ports data_in[0] -bundle
```

```
{data_in[0]}
```

```
SETUP> get_ports data_in[0][*].b -bundle
```

```
{data_in[0][2].b} {data_in[0][1].b} {data_in[0][0].b}
```

```
SETUP> get_ports data_in[1][2].* -bundle
```

```
{data_in[1][2].a} {data_in[1][2].b}
```

```
SETUP> get_ports data_in[1][*] -bundle
```

```
{data_in[1][2]} {data_in[1][1]} {data_in[1][0]}
```

Note



Integer, enum, and others are bit-blasted in unbundled objects in introspection commands when the `-bundle` switch is omitted.

Example With regexp

In the following example the second command call is equivalent to the first, but uses `-regexp`. You should enclose the regexp in double braces (`{{ { } }}`) to avoid having to use the `“\”` everywhere, that is, `“{{data_in\[.*\]\.da.*\[.*\]\.*/}`”.

With wildcard syntax:

```
SETUP> puts [join [get_name_list [get_ports data_in[*].da*[*].* ]] "\n"]
```

```
data_in[1].data[1].data1[1][1]
data_in[1].data[1].data1[1][0]
data_in[1].data[1].data1[0][1]
data_in[1].data[1].data1[0][0]
data_in[1].data[1].data2[1][2]
...
data_in[0].data[0].data2[0][2]
data_in[0].data[0].data2[0][1]
data_in[0].data[0].data2[0][0]
```

With regular expression syntax:

```
SETUP> puts [join [get_name_list [get_ports {{data_in\[.*\]\.da.*\[.*\]\.*/} -regexp]] "\n"]
```

```
data_in[1].data[1].data1[1][1]
data_in[1].data[1].data1[1][0]
data_in[1].data[1].data1[0][1]
data_in[1].data[1].data1[0][0]
data_in[1].data[1].data2[1][2]
...
data_in[0].data[0].data2[0][2]
data_in[0].data[0].data2[0][1]
data_in[0].data[0].data2[0][0]
```


Complex Signal Introspection Commands

The following commands support introspection of complex signals:

Table 3-5. Complex Signals Design Introspection Commands

Command Name	Description/Usage
get_bundle_objects	Returns a collection of bundle/leaf objects in the current design specified by the name_patterns list in conjunction with the specified options. Those bundle objects (leaf in case of descendants) are ancestors or descendants of the objects which match the specified name_patterns depending upon the specified options.
get_nets ... -bundle	When the -bundle switch is specified, constrains the tool to return net bundle/leaf objects. When the pattern matches a bundle, the command returns the bundle. When it matches leaf objects, it returns those leaf objects.
get_pins ... -bundle	When the -bundle switch is specified, constrains the tool to return pin bundle/leaf objects. When the pattern matches a bundle, the command returns the bundle. When it matches leaf objects, it returns those leaf objects.
get_ports ... -bundle	When the -bundle switch is specified, constrains the tool to return port bundle/leaf objects. When the pattern matches a bundle, the command returns the bundle. When it matches leaf objects, it returns those leaf objects.

Bundle Object Introspection Examples

In the following System Verilog code, several kinds of datatypes are illustrated: struct, array of interface, enumeration.

```
typedef enum logic [1:0] {
    STATE_1, STATE_2, STATE_3, STATE_4, STATE_5, STATE_6
} State_t;

typedef struct packed {
    logic [2:0] eventbus;
} event_bus_packet_t;

typedef struct packed {
    State_t state_encoded;
    event_bus_packet_t dt_lcp_pulse;
    logic [3:0] eventbus_encoded;
} event_bus_split_packet_t;

interface MSBus;
    event_bus_split_packet_t Addr;
    logic [1:0] Data;
    logic RWn;
    logic Clk;
    modport Secondary (input Addr, RWn, Clk, output Data);
endinterface

module top(MSBus.Secondary Bus[2], input wire i_clk);
    RAM TheRAM (.MemBus(Bus));
endmodule
```

Example 1

The following command returns all bit-blasted ports of “top” module (default behavior):

```
SETUP> get_ports -of_modules top
```

```
{Bus[0].Addr.state_encoded[2]} {Bus[0].Addr.state_encoded[1]}
{Bus[0].Addr.state_encoded[0]} {Bus[0].Addr.dt_lcp_pulse.eventbus[2]}
{Bus[0].Addr.dt_lcp_pulse.eventbus[1]}
{Bus[0].Addr.dt_lcp_pulse.eventbus[0]} {Bus[0].Addr.eventbus_encoded[3]}
{Bus[0].Addr.eventbus_encoded[2]} {Bus[0].Addr.eventbus_encoded[1]}
{Bus[0].Addr.eventbus_encoded[0]} {Bus[0].RWn} {Bus[0].Clk}
{Bus[0].Data[1]} {Bus[0].Data[0]}
{Bus[1].Addr.state_encoded[2]} {Bus[1].Addr.state_encoded[1]}
{Bus[1].Addr.state_encoded[0]} {Bus[1].Addr.dt_lcp_pulse.eventbus[2]}
{Bus[1].Addr.dt_lcp_pulse.eventbus[1]}
{Bus[1].Addr.dt_lcp_pulse.eventbus[0]} {Bus[1].Addr.eventbus_encoded[3]}
{Bus[1].Addr.eventbus_encoded[2]} {Bus[1].Addr.eventbus_encoded[1]}
{Bus[1].Addr.eventbus_encoded[0]} {Bus[1].RWn} {Bus[1].Clk}
{Bus[1].Data[1]} {Bus[1].Data[0]}
```

Example 2

The following commands return all root bundle ports of “top” module or sub bundle objects according to name patterns used:

```
SETUP> set b [get_ports -of_modules top -bundle]
```

```
Bus i_clk
```

```
SETUP> get_ports Bus -bundle
```

```
Bus
```

```
SETUP> get_ports B* -bundle
```

```
Bus
```

```
SETUP> get_ports Bus* -bundle
```

```
Bus
```

```
SETUP> get_ports Bus[0].* -bundle
```

```
{Bus [0] .Addr} {Bus [0] .RWr} {Bus [0] .Clk} {Bus [0] .Data}
```

```
SETUP> get_ports Bus[0].Addr* -bundle
```

```
{Bus [0] .Addr}
```

```
SETUP> get_ports Bus[0].Addr.* -bundle
```

```
{Bus [0] .Addr .state_encoded} {Bus [0] .Addr .dt_lcp_pulse .eventbus}  
{Bus [0] .Addr .eventbus_encoded}
```

```
SETUP> get_ports Bus[0].Addr.*
```

```
{Bus [0] .Addr .state_encoded[2]} {Bus [0] .Addr .state_encoded[1]}  
{Bus [0] .Addr .state_encoded[0]} {Bus [0] .Addr .dt_lcp_pulse .eventbus [2]}  
{Bus [0] .Addr .dt_lcp_pulse .eventbus [1]}  
{Bus [0] .Addr .dt_lcp_pulse .eventbus [0]} {Bus [0] .Addr .eventbus_encoded[3]}  
{Bus [0] .Addr .eventbus_encoded[2]} {Bus [0] .Addr .eventbus_encoded[1]}  
{Bus [0] .Addr .eventbus_encoded[0]}
```

```
SETUP> get_ports Bus[*] -bundle
```

```
{Bus [0]} {Bus [1]}
```

```
SETUP> get_ports Bus[*].* -bundle
```

```
{Bus [0] .Addr} {Bus [0] .RWr} {Bus [0] .Clk} {Bus [0] .Data} {Bus [1] .Addr}  
{Bus [1] .RWr} {Bus [1] .Clk} {Bus [1] .Data}
```

```
SETUP> get_ports Bus[0]
```

```
{Bus [0] .Addr .state_encoded[2]} {Bus [0] .Addr .state_encoded[1]}  
{Bus [0] .Addr .state_encoded[0]} {Bus [0] .Addr .dt_lcp_pulse .eventbus [2]}  
{Bus [0] .Addr .dt_lcp_pulse .eventbus [1]}  
{Bus [0] .Addr .dt_lcp_pulse .eventbus [0]} {Bus [0] .Addr .eventbus_encoded[3]}  
{Bus [0] .Addr .eventbus_encoded[2]} {Bus [0] .Addr .eventbus_encoded[1]}  
{Bus [0] .Addr .eventbus_encoded[0]} {Bus [0] .RWr} {Bus [0] .Clk}  
{Bus [0] .Data [1]} {Bus [0] .Data [0]}
```

```
SETUP> set b1 [get_bundle_objects $b -children]
```

```
{Bus [0]} {Bus [1]}
```

```
SETUP> set b2 [get_bundle_objects [index_collection $b1 0] -children]
```

```
{Bus [0] .Addr} {Bus [0] .RWr} {Bus [0] .Clk} {Bus [0] .Data}
```

```
SETUP> get_bundle_objects -of_objects {Bus[0].Addr} -root
```

```
{Bus}
```

```
SETUP> get_bundle_objects -of_objects {Bus[0].Addr} -leaf
```

```
Bus [0] .Addr .state_encoded [2] } {Bus [0] .Addr .state_encoded [1] }  
{Bus [0] .Addr .state_encoded [0] } {Bus [0] .Addr .dt_lcp_pulse .eventbus [2] }  
{Bus [0] .Addr .dt_lcp_pulse .eventbus [1] }  
{Bus [0] .Addr .dt_lcp_pulse .eventbus [0] } {Bus [0] .Addr .eventbus_encoded [3] }  
{Bus [0] .Addr .eventbus_encoded [2] } {Bus [0] .Addr .eventbus_encoded [1] }  
{Bus [0] .Addr .eventbus_encoded [0] }
```

```
SETUP> set b3 [get_bundle_objects [index_collection $b2 0] -children]
```

```
{Bus [0] .Addr .state_encoded} {Bus [0] .Addr .dt_lcp_pulse}  
{Bus [0] .Addr .eventbus_encoded}
```

```
SETUP> get_attribute_value_list [$b3] -name base_class
```

```
enum struct_packed 4-state_unsigned
```

```
SETUP> get_attribute_value_list [$b3] -name select_part_value_list
```

```
{0 Addr state_encoded} {0 Addr dt_lcp_pulse} {0 Addr eventbus_encoded}
```

```
SETUP> get_ports {Bus[0].Clk} -bundle
```

```
{Bus [0] .Clk}
```

```
SETUP> get_ports {Bus[0].Addr.state_encoded}
```

```
{Bus [0] .Addr .state_encoded [2] } {Bus [0] .Addr .state_encoded [1] }  
{Bus [0] .Addr .state_encoded [0] }
```

```
SETUP> get_bundle_objects -of_objects {Bus[0].Addr.state_encoded} -parent
```

```
{Bus [0] .Addr}
```

```
SETUP> get_bundle_objects -of_objects {Bus[0].Addr} -parent
```

```
{Bus [0] }
```

```
SETUP> get_bundle_objects -of_objects {Bus[0]} -parent
```

```
{Bus}
```

```
SETUP> get_bundle_objects -of_objects {Bus} -parent
```

```
{}
```

Fanin and Fanout Tracing of Complex Signals and Bundle Object Tracing

Using Tessent Shell, you can introspect and trace any kind of signals (simple and complex) through their fanins and fanouts. It is possible also to trace through bundle objects. The `get_fanins/get_fanouts` commands accept as input any kind of objects, bundle or not and with

simple types or complex types. The ending and intermediate nodes can also be an object or complex type. For example, they can trace from and to, and through any kind of complex signals in System Verilog and VHDL.

The following commands support this introspection:

- [get_fanins](#)
- [get_fanouts](#)

The following shows a top-level RTL module (*top1.sv*) with SV Interface objects. The tool models SV Interfaces as a named bundle of wires. Within the tool, those are considered as normal bundle ports, pins, and nets. You access the bundle object using the [get_nets](#), [get_pins](#), and [get_ports](#) commands.

```
module top (clk, reset, ce, we, addr, datai, datao, cnto, datao2, en);

    localparam WID = 8;
    localparam AWID = 5;
    localparam LEN = 2**AWID;
    localparam GEN = 3;

    input en;
    input clk, reset, ce, we;
    input [AWID-1:0] addr;
    input [WID-1:0] datai;
    output [GEN*12-1:0] datao, datao2;
    output [GEN*WID-1:0] cnto;

    and enAnd2 (resetEn, reset, en);
    and enAnd3 (weEn, we, en);
    and enAnd4 (ceEn, ce, en);

    param_mem_if miff(
        .clk(clk),
        .reset(resetEn),
        .we(weEn),
        .ce(ceEn),
        .dati(datai),
        .addr(addr),
        .dato(datao)
    );

    genvar i;

    generate
        for (i=0; i<GEN; i++) begin: gen_mem
            SYNC_1RW_16x8 mem1 (.CLK(clk), .D(datai), .Q(datao2[(WID*(i+1)
                -1):i*WID]), .WE(we), .OE(ce), .RE(ce), .A(addr[3:0]));
        end
    endgenerate

    generate
        for (i=0; i<GEN; i++) begin: gen_mem_wrapper

            param_mem mem_inst (.mif(miff));
        end
    endgenerate

    generate
        for (i=0; i<GEN; i++) begin: gen_cnt
            param_counter #(WID(WID)) counter_inst (
                .clk(clk),
                .reset(reset),
                .enable(ce&we),
                .count(cnto[(WID*(i+1)-1):i*WID]));
        end
    endgenerate
endmodule
```

The following shows an example command sequence to introspect the fanins and fanouts in this design:

```
SETUP> set_context dft -rtl
SETUP> read_cell_library my_cell_library.lib
SETUP> set_design_sources -format verilog -y {verilog_source_directory} -ext {v}
SETUP> set_design_sources -format tcd_memory -y {verilog_source_directory} \
-ext {lplib lib}
SETUP> read_verilog param_mem.sv -format sv2009
SETUP> read_verilog param_counter.sv -format sv2009
SETUP> read_verilog top1.sv -format sv2009
SETUP> set_current_design top
SETUP> set_design_level physical_block
SETUP> add_input_constraints en -C1
SETUP> puts [join [get_attribute_value_list \
[get_fanins gen_mem_wrapper[0].mem_inst/mem1/CLK ] -name name] "\n"]
```

```
clk
```

Here, from SV Interface sub bundle pins, the tool reaches the output pin of primitive. (Continuing from the previous example.)

```
SETUP> puts [join [get_name_list \
[get_fanouts gen_mem_wrapper[2].mem_inst/mif.ce ]] "\n"]
```

```
enAnd4/OUT
```

```
SETUP> puts [join [get_name_list \
[get_fanouts gen_mem_wrapper[2].mem_inst/mif.ce ] -name name] "\n"]
```

```
gen_mem_wrapper [2] .mem_inst/mem1/OE
gen_mem_wrapper [2] .mem_inst/mem1/RE
gen_mem_wrapper [2] .mem_inst/mem2/OE
gen_mem_wrapper [2] .mem_inst/mem2/RE
```

Here, from SV Interface sub bundle pins, the tool reaches the SV interface sub bundle net “miff.ce”. (Continuing with the previous example.)

```
SETUP> puts [join [get_name_list \
[get_fanins gen_mem_wrapper[2].mem_inst/mif.ce -stop_on net] \
-name name] "\n"]
```

```
miff.ce
```

```
SETUP> puts [join [get_name_list \
[get_fanouts gen_mem_wrapper[2].mem_inst/mif.ce -stop_on net] -name name] "\n"]
```

```
gen_mem_wrapper [2] .mem_inst/mif.ce
```

Additional Examples

The following command sequence shows introspection of bundle objects through both fanins and fanouts as well as ports and pins:

```
SETUP> set bitblasted_port [get_ports jack]
SETUP> set bitblasted_conn [get_fanouts $bitblasted_port]
SETUP> puts [join [get_name_list $bitblasted_conn] "\n"]
```

```
i1/jim[2]
i2/jim[1]
i1/jim[1]
i1/joe[2]
i1/jim[0]
i2/joe[2]
```

```
SETUP> set bundle_port [get_ports jack -bundle]
SETUP> set bundle_conn [get_fanouts $bundle_port]
SETUP> puts [join [get_name_list $bundle_conn] "\n"]
```

```
i1/jim
```

```
SETUP> set bitblasted_pin [get_pins i1/jim]
SETUP> set bitblasted_conn [get_fanins $bitblasted_pin]
SETUP> puts [join [get_name_list $bitblasted_conn] "\n"]
```

```
jack[2]
jack[1]
jack[0]
```

```
SETUP> set bundle_pin [get_pins i1/jim -bundle]
SETUP> set conn [get_fanins $bundle_pin]
SETUP> puts [join [get_name_list $bundle_conn] "\n"]
```

```
jack
```

Connectivity Through Complex Signals

Prior to the Tessent 2019.3 release, the connectivity described through complex RTL pins and nets were not visible until the RTL module was synthesized using the Quick Synthesis engine.

During ICL extraction or pre-DFT DRC, the fan-in and fanout of objects to be checked were pre-traced internally and if any complex objects were found along the way, such as traversing a net or a pin that is not a simple scalar or one dimensional bus, the module was synthesized to expose the connectivity to the DRC engine. As soon as a module with a System Verilog interface was synthesized, all modules connected to this interface also needed to be synthesized which often resulted in having to synthesize the majority of the design.

With the Tessent 2019.3 release and later, the connectivity through complex signals is visible to the [get_fanins/get_fanouts](#) commands on the hierarchical design view and to the [trace_flat_model](#) command on the flat model. Only if the path pre-tracing found RTL logic that represents logic gates does the tool synthesize the module. Also, the tool is now able to maintain the RTL view, and the connectivity to and from the surrounding modules even if a module

having an SV interface needs to be synthesized. This greatly limits the amount of RTL modules needed to be synthesized to pre-DFT DRC memoryBIST and performing ICL extraction.

When Tessent Shell flattens the design, only the leaf objects of complex signals are translated to gate pins using the post synthesis names (see the `post_synthesis_name` attribute on the `Port`, `port_bundle`, `Pin`, `pin_bundle`, `Net`, and `net_bundle` object types for more information).

For objects found inside unsynthesized modules, the hierarchical design objects uses the pre-synthesis names (see the `pre_synthesis_name` attribute on the `Port`, `port_bundle`, `Pin`, `pin_bundle`, `Net`, and `net_bundle` object types for more information).

The gate pins objects, however, are always named using the post synthesis name of the leaf complex signal objects whether or not the module is synthesized or not.

To simplify the introspection between the hierarchical design objects and the flat model gate_pin objects, the tool provides several switches on the introspection commands. With complex signal, it is no longer possible to assume the two objects have the same name.

- `get_gate_pins` -of_pins *pin_objects* |-of_ports *port_objects* |-of_nets *net_objects*
- `get_nets` [*name_patterns*] -of_gate_pins *gate_pin_objects*
- `get_pins` [*name_patterns*] -of_gate_pins *gate_pin_objects*
- `get_ports` [*name_patterns*] -of_gate_pins *gate_pin_objects*

The following example demonstrates loading an RTL design and performing tracing on the hierarchical data module using the `get_fanins` command. It also shows creating and tracing the flat model using the `trace_flat_model` command:

```
SETUP> set_context dft -rtl
SETUP> read_cell_library my_cell_library.lib
SETUP> open_tsdb InternalCore_tsdb
SETUP> read_design InternalCore -design_id rtl
SETUP> read_verilog RTL_Directory/Interfaces.sv -format sv2009
SETUP> read_verilog RTL_Directory/Core.sv -format sv2009
SETUP> set_current_design core
SETUP> set_design_level physical_block
SETUP> puts [get_name_list [get_fanins genloop[2].icw0/ic/OutBus[1]]]

{ InBus [1] }

SETUP> create_flat_model
// Flattening process completed, gates=79, PIs=12, POs=13, CPU time=0.00 sec.
// -----
// Begin circuit learning analyses.
// -----
// Learning completed, CPU time=0.00 sec.

SETUP> puts [get_name_list [trace_flat_model -from [get_gate_pins -of_pins \
[get_pins genloop[2].icw0/ic/OutBus[1]]] -direction backward \
-controllability connected]]
```

```
{InBus [1]}
```

This example shows the use of the “`get_gate_pins -of_pin`” and “`get_pins -of_gate_pins`” to illustrate that the matching objects do not have the same name:

```
SETUP> get_gate_pins -of_pins [get_pins genloop[2].icw0/ic/OutBus[1]]
```

```
{{\genloop[2].icw0 /ic/OutBus [1]}}
```

```
SETUP> get_gate_pins -of_pins genloop[2].icw0/ic/OutBus[1]
```

```
{{\genloop[2].icw0 /ic/OutBus [1]}}
```

```
SETUP> get_pins -of_gate_pins {{\genloop[2].icw0 /ic/OutBus[1]}}
```

```
{genloop[2].icw0/ic/OutBus [1]}
```

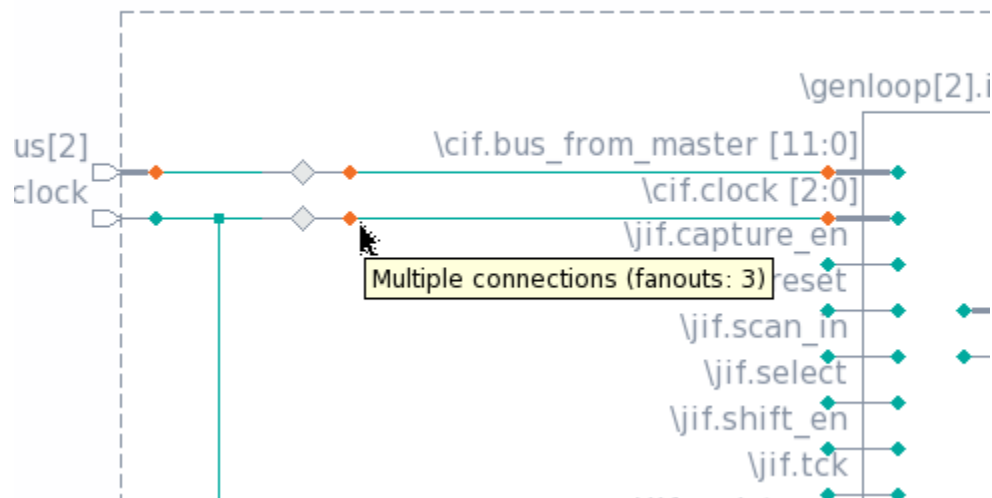
```
SETUP> get_pins -of_gate_pins [get_gate_pins {{\genloop[2].icw0 /ic/OutBus[1]}}
```

```
{genloop[2].icw0/ic/OutBus [1]}
```

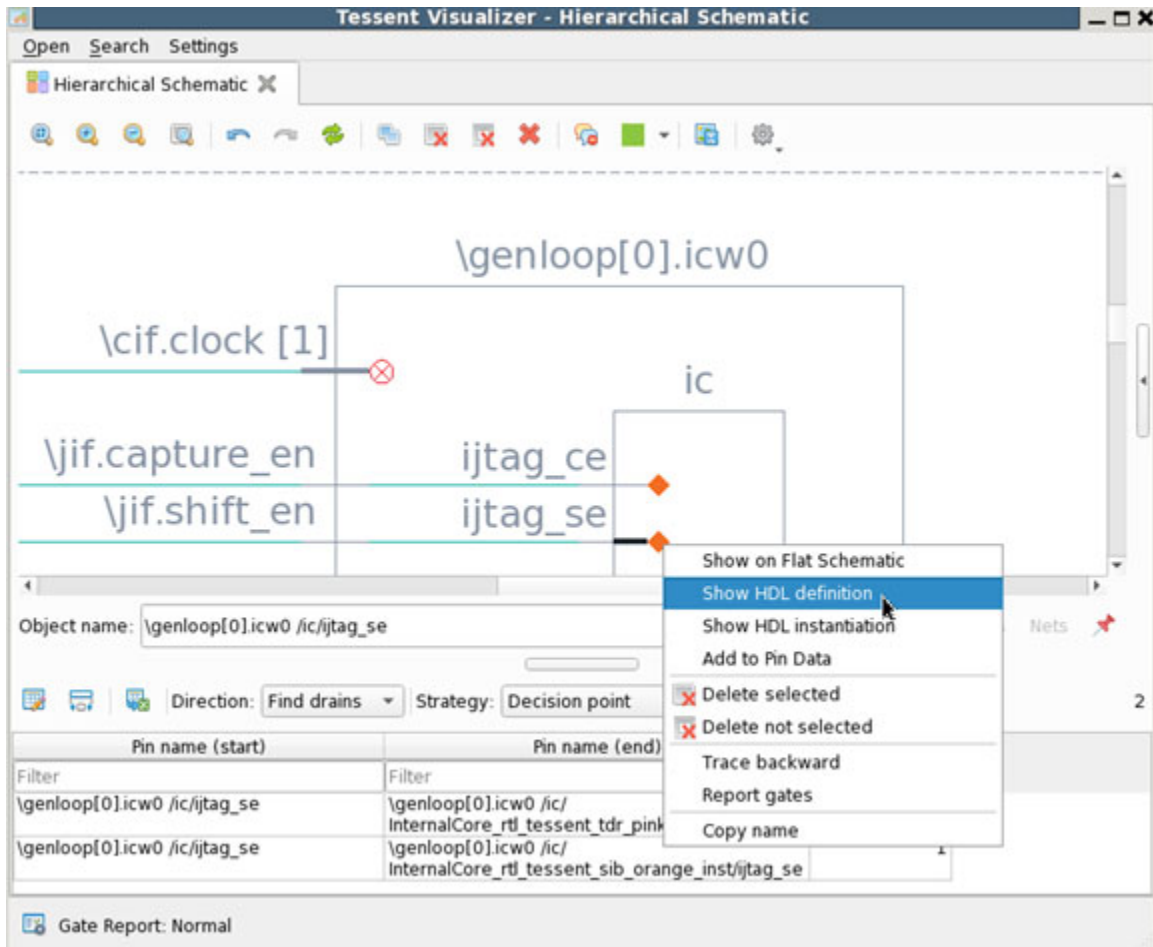
Tessent Visualizer Tracing and Visualization

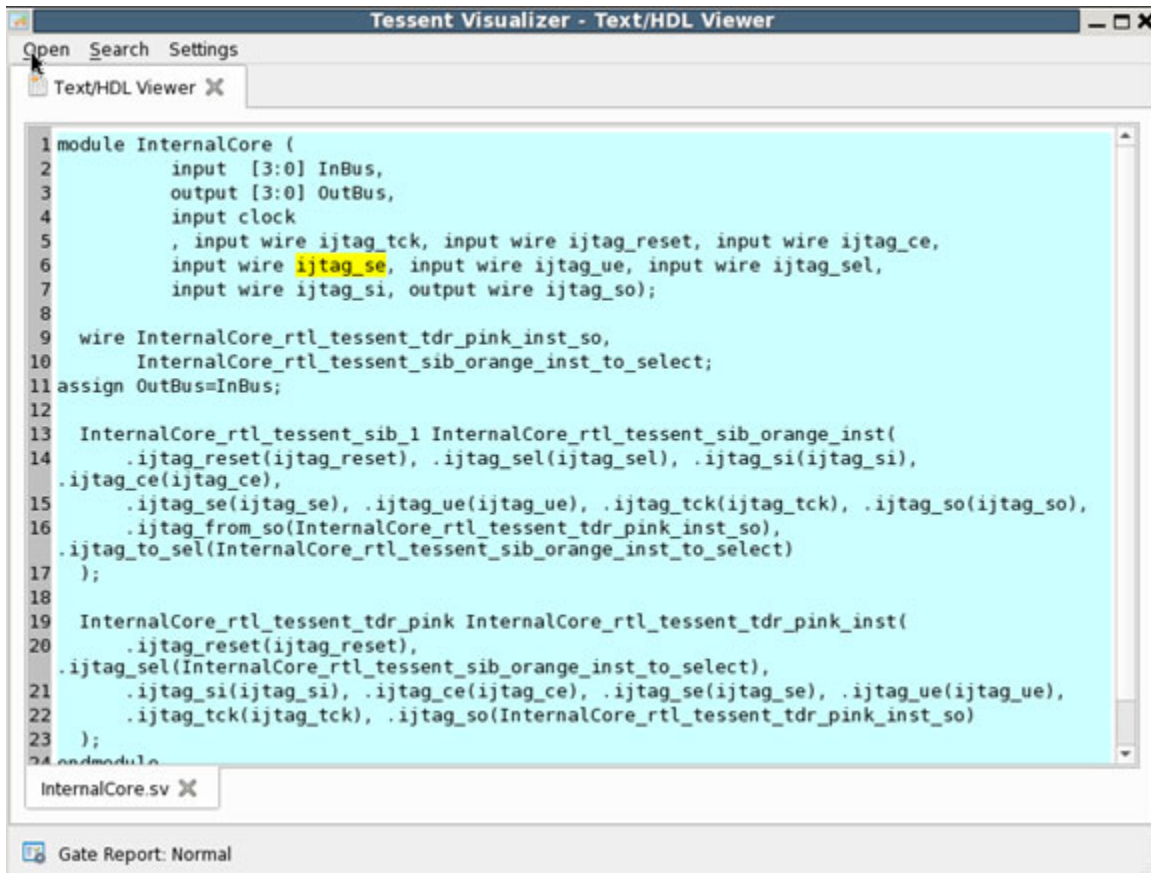
Tessent Visualizer provides a platform to trace and visualize bundle objects in the flat model. The Tessent Visualizer GUI shows the `post_synthesis_name` of the ports and pins even when the RTL view of the module is active. When you switch to the synthesized view of a module, its footprint remains unchanged in Tessent Visualizer. The RTL logic that is not shown in the RTL view becomes logic cell instances in the synthesized view.

Figure 3-2. Complex Signals with Tessent Visualizer



Tessent Visualizer facilitates cross-probing pins in the schematic to their associated RTL netlist.





The screenshot shows a window titled "Tessent Visualizer - Text/HDL Viewer" with a menu bar containing "Open", "Search", and "Settings". Below the menu bar is a tab labeled "Text/HDL Viewer". The main area displays Verilog code for an "InternalCore" module. The code includes input and output declarations, a clock input, and several internal module instantiations. The text "ijtag_se" is highlighted in yellow on line 6. At the bottom of the window, there is a status bar that reads "Gate Report: Normal".

```
1 module InternalCore (
2     input  [3:0] InBus,
3     output [3:0] OutBus,
4     input clock
5     , input wire ijtag_tck, input wire ijtag_reset, input wire ijtag_ce,
6     input wire ijtag_se, input wire ijtag_ue, input wire ijtag_sel,
7     input wire ijtag_si, output wire ijtag_so);
8
9     wire InternalCore_rtl_tessent_tdr_pink_inst_so,
10    InternalCore_rtl_tessent_sib_orange_inst_to_select;
11 assign OutBus=InBus;
12
13    InternalCore_rtl_tessent_sib_1 InternalCore_rtl_tessent_sib_orange_inst(
14        .ijtag_reset(ijtag_reset), .ijtag_sel(ijtag_sel), .ijtag_si(ijtag_si),
15        .ijtag_ce(ijtag_ce),
16        .ijtag_se(ijtag_se), .ijtag_ue(ijtag_ue), .ijtag_tck(ijtag_tck), .ijtag_so(ijtag_so),
17        .ijtag_from_so(InternalCore_rtl_tessent_tdr_pink_inst_so),
18        .ijtag_to_sel(InternalCore_rtl_tessent_sib_orange_inst_to_select)
19    );
20
21    InternalCore_rtl_tessent_tdr_pink InternalCore_rtl_tessent_tdr_pink_inst(
22        .ijtag_reset(ijtag_reset),
23        .ijtag_sel(InternalCore_rtl_tessent_sib_orange_inst_to_select),
24        .ijtag_si(ijtag_si), .ijtag_ce(ijtag_ce), .ijtag_se(ijtag_se), .ijtag_ue(ijtag_ue),
25        .ijtag_tck(ijtag_tck), .ijtag_so(InternalCore_rtl_tessent_tdr_pink_inst_so)
26    );
27 endmodule
```


For complete information on using Tessent Visualizer, see “[Tessent Visualizer](#)” on page 625.

Design Editing

Tessent Shell design editing commands enable you to modify your design after reading in the RTL or gate-level netlist. Tessent Shell supports gate-level netlist editing or RTL design editing with full language support, including multiple logical libraries, VHDL, Verilog, and System Verilog. Parameterized modules are also fully supported.


The Design editing commands allow you to manipulate a design's modules, instances, nets, ports, and pins, either interactively or through Tcl scripting.

Note

 There are language restrictions. See “[HDL Limitations in the Tessent Shell Flow](#)” in the *Tessent Shell Reference Manual* for details.

Design editing commands work with collections and introspection commands, as well as native Tcl commands, to automate many tasks. [Table 3-6](#) presents the common design editing commands based on the function they perform—that is, whether they create, modify, or remove elements and so on. For more information on collections and introspection commands, see “[Design Introspection](#).”

Caution

 Take special care when dealing with non-unique design scopes, such as multiple instances of the same module or the inner part of a generate loop. See “[Example of Handling Non-Unique Design Scopes](#)” on page 83 for details.

Note


 The design editing commands are not available when using some product licenses, such as Tessent FastScan and Tessent Scan.

Table 3-6. Design Editing Commands

Commands that...	Command Name
Read netlists and specify logical libraries	read_verilog
	read_vhdl
	set_logical_design_libraries
Create design elements	create_connections
	create_instance
	create_module
	create_net
	create_pin
	create_port

Table 3-6. Design Editing Commands (cont.)

Commands that...	Command Name
Remove design elements	delete_connections
	delete_instances
	delete_nets
	delete_pins
	delete_ports
Modify the design	copy_module
	intercept_connection
	replace_instances
	uniquify_instances
Set or get editing options	get_insertion_option
	set_insertion_options

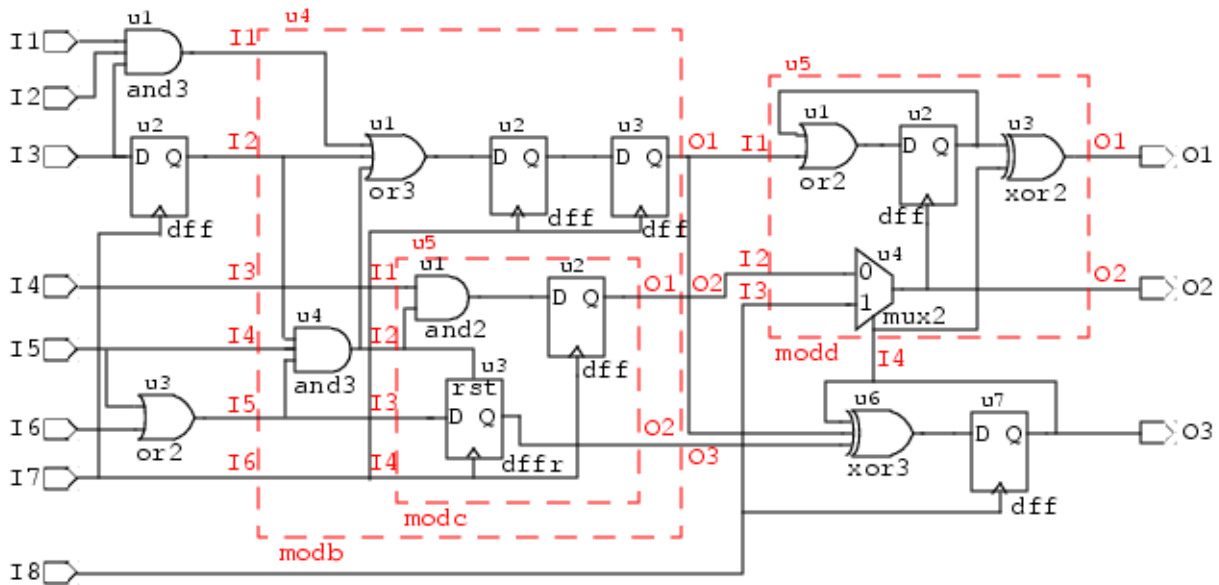
Design Editing Examples..... 78
Design Editing Command Summary..... 83

Design Editing Examples

There are many ways to create, modify, and remove elements from your design in a Tcl scripting environment.

[Figure 3-3](#) is an example design, which is followed by related examples that show some of the ways you can edit designs within Tessent Shell.

Figure 3-3. Hierarchical Design Example



Example of Creating a Module From Scratch

The following example shows how to create module C (modc) in [Figure 3-3](#) as a standalone module.

```

set_context dft -no_rtl
set_system_mode insertion
create_module modc
set_current_design modc

create_port I1 -on_module modc -direction input
create_port I2 -on_module modc -direction input
create_port I3 -on_module modc -direction input
create_port I4 -on_module modc -direction input
create_port O1 -on_module modc -direction output
create_port O2 -on_module modc -direction output

create_instance u1 -of_module and2
create_instance u2 -of_module dff
create_instance u3 -of_module dffr

create_connection I1 u1/A0
create_connection I2 u1/A1
create_connection I2 u3/R
create_connection I3 u3/D
create_connection I4 u3/CLK
create_connection I4 u2/CLK
create_connection u1/Y u2/D
create_connection u2/Q O1
create_connection u3/Q O2

write_design -output_file MODC.v -replace
    
```

Example of Replacing a Module With Another Module

The following example replaces the module definition for instance u5 (modd) to modc. The original module and the new module have the same pinout, and the connecting nets remain the same after issuing the `replace_instances` command. After replacing instance u5 with modc, that instance u5/u1 changes from an or2 gate to an and02 gate.

```
INSERTION> get_attribute_value_list u5 -name module_name
modd

INSERTION> get_fanin u5/I1 -stop_on net
{u4_O1}

INSERTION> replace_instances u5 -with_module modc
{u5}

INSERTION> get_fanin u5/I1 -stop_on net
{u4_O1}

INSERTION> set_system_mode setup
SETUP> set_design_level sub_block
SETUP> set_sys_mode analysis

// Flattening process completed, cell instances=15, gates=45, PIs=8
// POs=3, CPU time=0.00 sec.
// -----
// Begin circuit learning analyses.
// -----
// Learning completed, CPU time=0.00 sec.

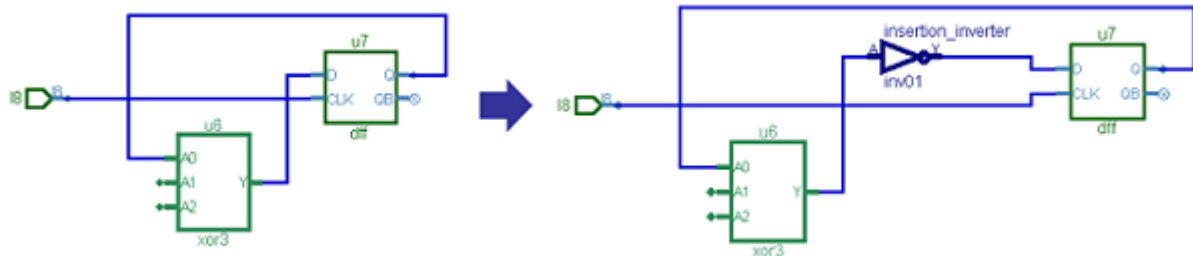
ANALYSIS> report_gates u5/u1

// /u5/u1 and02
//   A0   I  /u4/u3/Q
//   A1   I  /u4/u5/u2/Q
//   Y    O  /u5/u2/D
```

Example of Intercepting a Connection

Figure 3-4 shows an example of inserting an inverter by intercepting the existing connection between the u7/D pin and the u6/Y pin. After the interception, the inverter A pin is connected to the u6/Y pin and the inverter Y pin is connected to the u7/D pin.

Figure 3-4. Inverter Inception



```
INSERTION> intercept_connection u7/D -cell_function_name inverter
```

```
{insertion_inverter}
```

Example of Adding Input and Output Pads to a Design

The following dofile example adds input and output pads to a design by creating a collection of port names from the design, creating a collection of pad names that match the ports, then connecting the ports and pads together in the design. The example also adds the TAP's I/O ports, including the respective pads.

```
# Setting the context to netlist editing
set_context dft -no_rtl

# Reading the library and all necessary Verilog files
read_cell_library ../libs/libs/adk.atpg
read_verilog      ./counter_block2_edt_top_gate.v
read_verilog      ../libs/pads/*.v

# Telling the tool, which module 'top' is
set_top_module "counter_block2"
set_current_design $top_module

# Now for the editing
set_system_mode insertion

# First, insert the pad cells for all inputs of the netlist
set inputPortList [ get_ports -of_module $top_module -direction input ]
foreach_in_collection inputPort $inputPortList {

    # Creating the name of the pad cell:
    # The following lines take care of '[' and ']' in the portname,
    # substituting these by '_'. This makes the Tcl-life easier
    set portName [lindex [get_name_list $inputPort] 0]
    set padName  [ string map { \[ _ \] _ } ${portName}_PAD ]

    # Adding the pad cell
    create_instance $padName -of_module INPAD

# Connecting it up in the netlist
# The first line moves the existing net from the input port to the output
# of the pad cell. The second line creates a new net and connects the
# input port to the input of the pad cell
# Note: We are not connecting anything else here.
# We leave this to boundary scan insertion
    move_connection -from $inputPort -to $padName/FP
    create_connection $inputPort $padName/IO
}

# Second, insert the pad cells for all outputs of the netlist
set outputPortList [ get_ports -of_module $top_module -direction output ]
foreach_in_collection outputPort $outputPortList {

    # Creating the name of the pad cell:
    # The following lines take care of '[' and ']' in the portname,
    # substituting these by '_'.
    set portName [lindex [ get_name_list $outputPort ] 0]
    set padName  [ string map { \[ _ \] _ } ${portName}_PAD ]

    # Adding the pad cell
    create_instance $padName -of_module OUTPAD_2S

# Connecting it up in the netlist
# The first line moves the existing net from the IO port to the input
# of the pad cell. The second line creates a new net and connects the
# IO port to the output of the pad cell.
# Note: We are not connecting anything else here.
# We leave this to boundary scan insertion
    move_connection -from $outputPort -to $padName/TP
```

```
    create_connection $outputPort $padName/IO
}

# Next, add the JTAG IO pins and their respective PAD cell. Connecting #
them up.

set inputPortList { TDI TMS TRST TCK }
set outputPortList { TDO }

foreach portName $inputPortList {

    # Adding the pad cell
    set padName      ${portName}_PAD
    create_instance $padName -of_module INPAD

    # Creating an Input IO pin and connecting the pad
    create_port $portName -direction input
    create_connection $portName $padName/IO
}

foreach portName $outputPortList {

    # Adding the pad cell
    set padName      ${portName}_PAD
    create_instance $padName -of_module OUTPAD_EN1

    # Creating an output pin and connecting the pad
    create_port $portName -direction output
    create_connection $portName $padName/IO
}

# Write the new netlist
write_design -output_file "${top_module}.v" -replace
exit
```

Example of Handling Non-Unique Design Scopes

When you want to make edits to non-unique design scopes, such as multiple instances of the same module or the inner part of a generate loop, you must exercise caution.

When using the [delete_connections](#) or [move_connections](#) commands inside a non-uniquified design, only perform the move or disconnection once per module or once per generate loop count.

For an example, see how the `parent_instance` and the `leaf_name_hash` attributes are used in Example 5 of the [get_dft_cell](#) command description. (For more information on the `leaf_name_hash` attribute, see “[Instance](#)” in the *Tessent Shell Reference Manual*.)

Design Editing Command Summary

The design editing commands allow you to work with modules, connections, instances, nets, pins, and ports.

The commands listed in the following table are commonly used design editing commands. Refer to the *Tessent Shell Reference Manual* for more information.

Table 3-7. Design Editing Command Summary

Command Name	Description
<code>copy_module</code>	Creates an exact copy of a design module and gives it a new name, which the tool can use as part of <code>create_instance</code> and <code>replace_instances</code> operations.
<code>create_connections</code>	Creates a connection between pin, net, or port objects.
<code>create_instance</code>	Instantiates a module (design or cell type) inside of a design module that is part of the current design.
<code>create_module</code>	Creates a new design module.
<code>create_net</code>	Creates a net inside an instance of a design module.
<code>create_pin</code>	Creates a pin on an instance of a design module.
<code>create_port</code>	Creates a port on a design module.
<code>delete_connections</code>	Disconnects the specified pin objects.
<code>delete_instances</code>	Removes an instance of a module.
<code>delete_nets</code>	Removes net objects inside an instance of a design module.
<code>delete_pins</code>	Removes pin objects on an instance of a design module.
<code>delete_ports</code>	Removes port objects on a design module.
<code>get_insertion_option</code>	Introspects the default values of options affecting many design editing commands.
<code>intercept_connection</code>	Using the <code>get_dft_cell</code> command, obtains a cell with the specified function name and uses it to intercept a connection to a pin, port, or net.
<code>move_connections</code>	Moves a net connected on one pin or port to another pin or port. The first pin is left open after the move.
<code>rename_instance</code>	Renames the leaf name of an instance object.
<code>replace_instances</code>	Replaces the module object used in an instantiation.
<code>set_insertion_options</code>	Specifies default values of options affecting many design editing commands.
<code>uniquify_instances</code>	If the module of the specified instance has other instantiations in the design, the tool copies the module and the specified instance becomes an instance of the copied module.

Simulation Contexts

Tessent Shell provides simulation contexts to aid your design analysis and introspection efforts. You can use these contexts to create “simulation scratch pads” for rapid investigation of good-machine behavior of specific portions of your design.

Simulation Context Overview	85
Introspection and Analysis Using Simulation Contexts	86

Simulation Context Overview

Tessent Shell provides commands for creating and managing simulation contexts. From a particular user-defined simulation context, you can use these commands to apply stimulus forces to specified gate_pins, run simulation for a specific number of cycles, and then introspect gate_pins for simulation values.

Commands for Managing Simulation Contexts

Use the following commands to create and manage user-defined simulation contexts, as well as use predefined simulation contexts:

- [add_simulation_context](#) — Creates a new user-defined simulation context.
- [copy_simulation_context](#) — Copies the simulation values and forces from one simulation context to another.
- [delete_simulation_contexts](#) — Deletes one or more user-defined simulation contexts.
- [get_current_simulation_context](#) — Returns the name of the current simulation context.
- [get_simulation_context_list](#) — Returns the available simulation contexts in a Tcl list.
- [report_simulation_contexts](#) — Lists the available simulation contexts and indicates the current simulation context.
- [set_current_simulation_context](#) — Sets the current simulation context.

Commands for Managing Stimulus and Simulating within Simulation Contexts

The following commands are for managing stimulus (simulation forces and clock pulses) and running limited simulations within a simulation context:

- [add_simulation_forces](#) — Forces one or more gate_pin objects to the specified value.
- [delete_simulation_forces](#) — Removes forces from one or more gate_pin objects.
- [report_simulation_forces](#) — Lists the active forces on the specified gate_pin objects.

- [simulate_clock_pulses](#) — Pulses one or more clocks within the current simulation context.
- [simulate_forces](#) — Simulates the queued forces in the current simulation context.

Commands for Introspection and Analysis within Simulation Contexts

The following commands are for introspection and analysis within simulation contexts, which includes examining simulation results:

- [get_current_simulation_context](#) — Returns the name of the current simulation context.
- [get_simulation_context_list](#) — Returns the available simulation contexts in a Tcl list.
- [get_simulation_value_list](#) — Returns the simulation values on the specified gate_pin objects.
- [report_simulation_contexts](#) — Lists the available simulation contexts and indicates the current simulation context.
- [report_simulation_forces](#) — Lists the active forces on the specified gate_pin objects.
- [set_gate_report](#) — Specifies the information displayed by the report_gates command. This command enables reporting of simulated values in the current simulation context.
- [trace_flat_model](#) — Traces the flat model within the current simulation context. This command always operates within the current simulation context.

Attributes for gate_pin Objects

For a complete list of attributes, refer to the “[Data Models](#)” chapter in the *Tessent Shell Reference Manual*.

Introspection and Analysis Using Simulation Contexts

Simulation contexts allow you to perform various types of design analysis and introspection. The four predefined simulation contexts are: `stable_after_setup`, `stable_load_unload`, `stable_shift`, and `stable_capture`. After setting a simulation context, you can introspect gate_pins for simulation values based on that specific simulation context. For example, the `trace_flat_model` command can do design tracing based on values specified for the current simulation context.

To use any of the following techniques in this chapter, you must be doing the following:

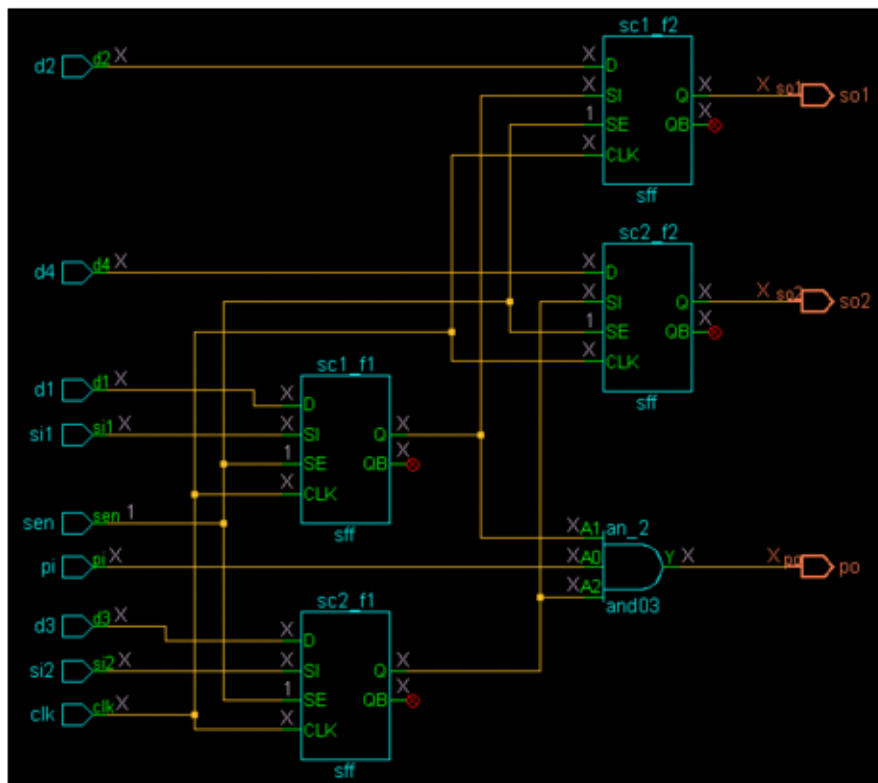
- Operating in the analysis system mode of Tessent Shell.

- Operating on a flattened design, and have read in the test procedure file (if available). At this point, the values specified in the setup, shift, capture, and load_unload procedures are in place in the design when you set the corresponding simulation context as current.

Example

For example, if you want to trace the design based on the values specified in the shift procedure, use the “set_current_simulation_context stable_shift” command. When you use this command with a small design containing two 2-cell scan chains, the values on the gate_pins display in the Tessent Visualizer Flat Schematic as shown in [Figure 3-5](#).

Figure 3-5. Tessent Visualizer Flat Schematic (gate level)



In this case, no values are forced except for sen (scan enable), which is set to 1. The four possible simulation values are 0, 1, X, and Z.

In addition to viewing values in the Flat Schematic, you can get the current simulation values of gate_pins using any of the following techniques:

- Use the get_simulation_value_list command and specify the gate_pin object(s):

```
ANALYSIS> set_current_simulation_context stable_shift
ANALYSIS> get_simulation_value_list sc2_f1/D
```

X

- Check the value of the simulation_value attribute for the gate_pins:

```
ANALYSIS> get_attribute_value_list [get_gate_pin sc2_f1/D]
-name simulation_value
```

```
X
```

- Issue the command “set_gate_report simulation_context,” after which you can use the report_gates command and the Flat Schematic to display the simulated values from the current simulation context:

```
ANALYSIS> set_gate_report simulation_context
ANALYSIS> report_gates sc2_f1
```

```
// /sc2_f1 sff
//      D      I (X) /d3
//      SI      I (X) /si2
//      SE      I (1) /sen
//      CLK     I (X) /clk
//      Q       O (X) /an_2/A2 /sc2_f2/SI
//      QB      O (X)
```

You can use simulation context functionality for different types of analysis. For example, by default the trace_flat_model command uses the stable_after_setup values as a simulation background. You can now change it to stable_capture, for example, if you want to trace based on sensitized paths during capture. Another example is to copy an existing simulation context to a new one, force some simulation value changes, then evaluate the results in the circuit after simulating the forces or clock pulses.

Automatic Design Mapping

Logic synthesis may cause name and footprint changes in the design for both complex and regular ports.

ICL and TCD Post-Synthesis Update	89
Updating ICL Attributes From the Design	90
Matching Rules for Port Names in Post-Synthesis Update	91
Controlling the Name Mapping	91

ICL and TCD Post-Synthesis Update

Logic synthesis flattens (bit-blasts) ports. This changes the design footprint. Additionally, post-synthesis names differ from those in the initial RTL.

Tessent in `-no_rtl` mode uses the post-synthesis netlist when you run `set_current_design`. The ICL and TCD of the current design updates automatically to match the new design objects that are loaded from the post-synthesis netlist. The following ICL model attributes are updated to enable binding of the ICL objects and design objects later in the flow:

- `tessent_design_instance` (for the instance path)
- `tessent_design_gate_ports` (for ports)
- `tessent_design_rtl_ports` (for ports)

The `tessent_design_instance` attribute of the ICL instance object is automatically updated when you use any of the following commands:

- `set_current_design`
- `write_design`
- `update_icl_attributes_from_design`

The `tessent_design_gate_ports` attribute of the ICL port object is automatically updated when you use any of the following commands:

- `set_current_design`
- `get_ports -of_icl_ports`
- `get_pins -of_icl_pins`
- `get_pins -of_icl_ports`
- `update_icl_attributes_from_design`
- `write_design`

For ICL ports and instances, if the current ICL model does not include the complete ICL of child instances under the physical blocks, the ICL matching performed by `set_current_design` is insufficient. The ICL matching is completed later in the flow using the following commands in these cases:

- `get_ports -of_icl_ports`
- `get_pins -of_icl_pins`
- `get_pins -of_icl_ports`
- `update_icl_attributes_from_design`
- `write_design`

Updating ICL Attributes From the Design

The `update_icl_attributes_from_design` command updates ICL port, instance, and module attributes for a design's gate views. The following attributes are updated:

- **tessent_design_gate_ports** — This attribute refers to the design port corresponding to the ICL port. The attribute value is the name of the design port as it appears in the netlist.
- **tessent_design_instance** — This attribute refers to the design instance corresponding to the ICL instance. The value is the hierarchical name relative to the parent ICL instance.
- **tessent_design_rtl_gate_port_mapping** — This module attribute maps the RTL name to the netlist name for those ports that appear in the following attribute name lists:
 - `forced_high_input_port_list`
 - `forced_low_input_port_list`
 - `forced_high_output_port_list`
 - `forced_low_output_port_list`
 - `forced_high_internal_input_port_list`
 - `forced_low_internal_input_port_list`
 - `tessent_ground_port_list`
 - `tessent_power_port_list`
 - `tessent_clock_domain_labels`

The attribute value is a list of names. The names at the even index positions in the list are the RTL names of the ports in the attribute name lists, and the names at the odd index positions are the netlist names of those ports.

The `update_icl_attributes_from_design` command has one Boolean option: `-verbose`. This option enables warnings when a port name cannot be found or matched in the netlist. The command is invoked automatically by the `write_design -tsdb` command. It also runs automatically as part of the `insert_test_logic` command just after insertion.

Matching Rules for Port Names in Post-Synthesis Update

The following matching rules are supported when looking up a port name in a netlist:

1. Apply the transformation performed by `dc_shell` with `change_names -rules verilog` to remove the escaping and replace special characters with an underscore (“_”). A single underscore matches multiple underscores.
2. Apply the transformation performed by Cadence© Genus™ Synthesis Solution to get the gate name from the RTL name. All strings and numerical indices in the RTL name are preserved. Only the delimiters are changed.
3. Apply the Genus transformation described in rule #2 but with escaping removed and special characters replaced with an underscore (“_”) as in rule #1.
4. Apply bit-blasted escaped names generated by a layout tool. The bit select is incorporated into the escaped name.
5. Apply end-user-specified matching rules.

Controlling the Name Mapping

Use the `add_rtl_to_gates_mapping` and `delete_rtl_to_gates_mapping` commands to control the name mapping. Use the `report_rtl_to_gates_mapping` command to obtain information about the current mapping rules.

See the [Tessent Shell Reference Manual](#) for information on these commands.

Default Matching Rules for the `get_pins` and `get_ports` Commands

- **Component Names** — All strings between non-escaped delimiters (component names) in the name to be looked up (lookup name) must match exactly to corresponding strings in a netlist name unless they contain special characters (see below). The recognized delimiters are ‘.’, ‘/’, ‘[]’, ‘_’.

An example lookup name:

```
abc.def
```

The ‘abc’ string in the name to be looked up must exactly match the string before the first delimiter in a netlist name. The ‘def’ string in the name to be looked up must exactly match the string after the last delimiter in the same netlist name.

- **Escaping** — Escape characters are not matched. They are only used to direct the matching procedure.
- **Delimiters** — Delimiters in the lookup name are used only to extract the component names.

All component names after the first are assumed to have a leading delimiter and optionally a trailing delimiter in a netlist name. There are two cases to consider when matching the delimiters for a component name in the netlist:

- a. Port name in netlist is escaped.

Possible matches for component name delimiters are as follows:

- Leading and trailing delimiters of [].
- Leading delimiter of ‘_’ and trailing delimiter of ‘_’ or null.
- Leading delimiter of ‘.’ or ‘/’ and trailing delimiter of null.


- b. Port name in netlist is not escaped.

Possible matches for component name delimiters are as follows:

- Leading delimiter of ‘_’ and trailing delimiter of ‘_’ or null.

- **Special Characters in the Lookup Name** — When matching to a non-escaped name in the netlist, any special characters (not allowed by the language without escaping) in the lookup name are replaced with the ‘_’ character. Matching allows truncation in the netlist name down to two trailing ‘_’ characters.

Note

 This truncation rule applies only to ‘_’ characters derived from special characters, not delimiters.

- **Bit Select Lookup Name** — A bit select lookup name (last component name is an index) can match a one-dimensional vector with the final bit select applied to the match or match directly to a bit select.

Related Topics

[add_rtl_to_gates_mapping](#)

[delete_rtl_to_gates_mapping](#)

[report_rtl_to_gates_mapping](#)

ICL Objects vs. Design Objects Introspection

Several Tessent Shell commands enable you to introspect ICL objects and design objects in an ICL context.

For example, you can retrieve design modules and instances from ICL modules and instances:

- `get_module -of_icl_module icl_module_spec`
- `get_instance -of_icl_instance icl_instance_spec`

You can retrieve ICL modules and instances from design modules and instances:

- `get_icl_module -of_module module_spec`
- `get_icl_instance -of_instance instance_spec`

Certain additional command-line switches for these commands are necessary when your design includes complex ports, or simple ports with escaped names requiring name changes during logic synthesis.

And you can also retrieve design pins and ports from ICL pins and ports:

- `get_pins -of_icl_pins icl_pin_objects`
- `get_ports -of_icl_ports icl_port_objects`

You can also retrieve icl pins and ports from design pins and ports:

- `get_icl_pins -of_pins icl_pin_objects`
- `get_icl_ports -of_ports icl_port_objects`

Related Topics

[get_modules](#)

[get_instances](#)

[get_icl_modules](#)

[get_icl_instances](#)

[get_pins](#)

[get_ports](#)

[get_icl_pins](#)

[get_icl_ports](#)

Chapter 4

DFT Architecture Guidelines for Hierarchical Designs

Most chips follow hierarchical place-and-route because of the increase in design sizes. For DFT, you can also use hierarchical design methodologies to perform test insertion, test generation, and diagnosis. This use of the hierarchical design methodologies is called hierarchical DFT or hierarchical test.

If the design is implemented as one chip with flat place-and-route, then perform DFT implementation once for the entire chip. Hierarchical DFT implementation is only applicable for block-based designs in which the tool run time consumption is not practical for implementing DFT only from the chip level.

Hierarchical DFT has several advantages:

- Divide and conquer technique that helps with parallelization of tasks.
- Reduced memory footprint and CPU requirements to perform the given task.
- Faster pattern generation and simulation run times.
- DFT performed earlier at block level and out of design critical path.

For information about the Tessent Shell two-pass DFT insertion flow for hierarchical designs, refer to “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141.

Hierarchical DFT Overview	96
Top-Down Planning Before Bottom-Up Implementation	102
DFT Implementation Strategy	107

Hierarchical DFT Overview

Hierarchical DFT is the process of using hierarchical design methodologies to perform test insertion, test generation, and diagnosis. Within the Tessent Shell environment, multiple components, such as physical layout regions and wrapped cores, are used to implement DFT in hierarchical designs.

Physical Layout Regions in Hierarchical Test	96
Pattern Retargeting	97
Wrapped Cores and Wrapper Cells	97
Internal Mode and External Mode	98
On-Chip Clock Controller	99
Graybox Model	100

Physical Layout Regions in Hierarchical Test

Hierarchical designs typically have several physical layout regions, and these layout regions are instantiated into other physical layout regions. Usually, the instantiations occur at the chip level, so you have two levels of hierarchy (or sometimes more for some designs). From the DFT perspective, each layout region is where you perform test insertion, with the test insertion process performed independently for each region. You can implement DFT within the layout regions in parallel, thus improving throughput.

In some designs there may be more than two levels of physical hierarchy. From the DFT perspective, each layout region is where you perform test insertion, with the test insertion process performed independently for each region.

You must insert DFT into the lower-level designs before inserting it at the next higher level (at the chip level).

The following terms describe the layout regions of a hierarchical design:

- **Physical block, block, core, child core** — These terms are interchangeable and refer to layout regions below the chip level that you can instantiate within a chip, or across multiple chips. Blocks are logical entities that remain intact through tapeout. You perform synthesis on these blocks independent of the rest of the chip design.

The term “wrapped core” is a specialized type of block. Refer to “[Wrapped Cores and Wrapper Cells](#)” on page 97 for details.

- **Chip** — The chip is the top-level physical block — that is, the entire design — in which you typically find the pad I/O macros and clock controllers.

The chip and upper-level blocks in designs with more than two hierarchical levels are often referred to as the parent blocks and the tasks related to them as occurring at the parent level.

Pattern Retargeting

Pattern retargeting is the process by which Tessent Shell preserves the ATPG patterns associated with cores for purposes of reuse when testing the logic at the chip (or parent) level. You do not have to regenerate the patterns when you process the chip. Instead, you retarget the core patterns to the chip level. Every instantiation of a core includes its associated ATPG patterns.

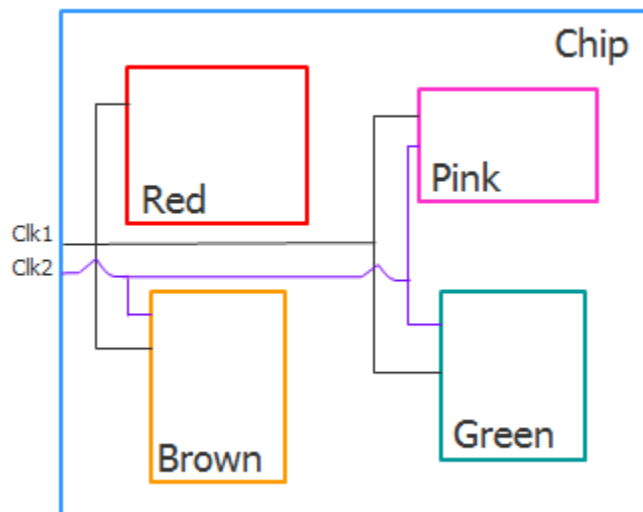
The pattern retargeting process consists of generating patterns for all cores, retargeting the core-level test patterns to the chip level, and generating patterns for the top-level chip logic only. This is also referred to as ATPG pattern retargeting or scan pattern retargeting. For details, refer to “[Scan Pattern Retargeting](#)” in the *Tessent Scan and ATPG User’s Manual*.

Wrapped Cores and Wrapper Cells

In hierarchical DFT, you must wrap physical blocks to make them reusable through pattern retargeting. The term *wrapped cores* applies to these physical blocks. The purpose of wrapping a core is to isolate its internal logic.

In the following figure, the physical layout regions from a DFT perspective are physical blocks. Tessent performs test insertion on the Red, Pink, Brown, and Green physical blocks first. Clk1 and Clk2 are asynchronous clocks.

Figure 4-1. Wrapped Cores



The wrapping mechanism that you use for these blocks makes use of the functional flops that are already present at the boundary of these blocks. These flops are called shared wrapper cells (or sometimes wrapper cells) because they share the responsibility of their original functional use as I/O flops with isolating the core.

Optionally, you can add dedicated wrapper cells. These wrapper cells are added on primary inputs or primary outputs of a physical block that fan out from or fan into large logic cones. By adding the dedicated wrapper cell, you can test the logic cone during internal testing of the core.

Shared and dedicated wrapper cells form the wrapper chains. Logic located within the wrapper chains is tested during internal testing of the core, called *intest*. Logic located outside the wrapper chains is tested during the external testing of the core, called *extest*.

Internal Mode and External Mode

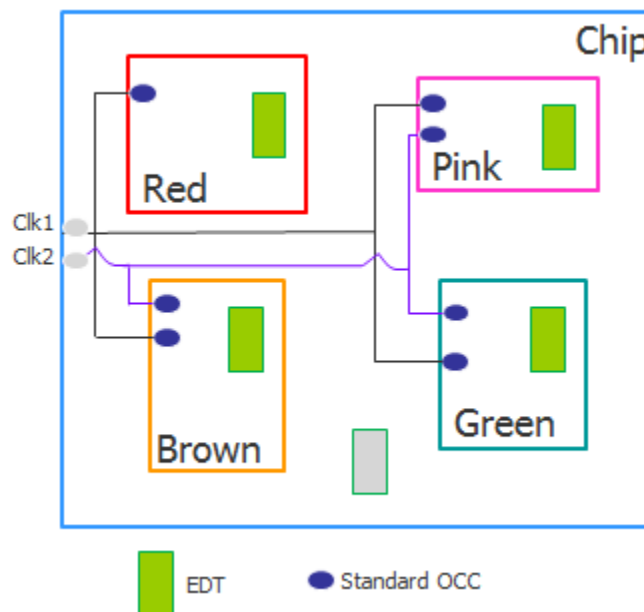
For purposes of pattern retargeting, Tessent Shell differentiates between a wrapped core's internal circuitry and its external circuitry.

Internal mode is the view into the wrapped core from the wrapper cells. That is, the logic completely internal to the core. Tessent Shell retargets internal mode ATPG patterns during ATPG pattern generation for the chip-level design.

External mode indicates the view out of the wrapped core from the wrapper cells. That is, the logic that connects the wrapped core to external logic. Tessent Shell uses the external mode to build graybox models, which are used by the internal modes of their parent physical blocks.

As shown in the following figure, based on chip pin availability, you can test the Red, Pink, Brown, and Green cores in internal mode at the same time when the on-chip clock controllers (OCCs) and EDT logic blocks inside the wrapped cores are active.

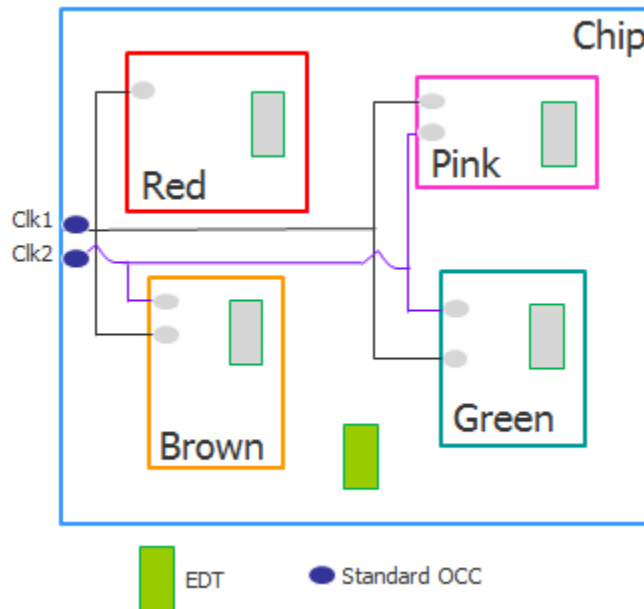
Figure 4-2. Internal Mode



You can test one or more wrapped cores in internal mode based on several factors, such as availability of chip-level pins, tester channels, tester memory, and power dissipation. The OCCs and EDT compression logic at the chip level are inactive during internal mode.

In external mode, the wrapped cores at the given physical hierarchy participate. The OCCs and EDT logic blocks inside the cores are inactive, and the OCCs and EDT logic blocks outside the wrapped cores are active. In a typical external mode configuration, one EDT logic block is used to test the logic outside the wrapper chains, as shown in the following figure.

Figure 4-3. External Mode



External mode also includes the wrapper chains within the wrapped cores. Tessent tests the logic outside of the wrapper chains during external mode.

On-Chip Clock Controller

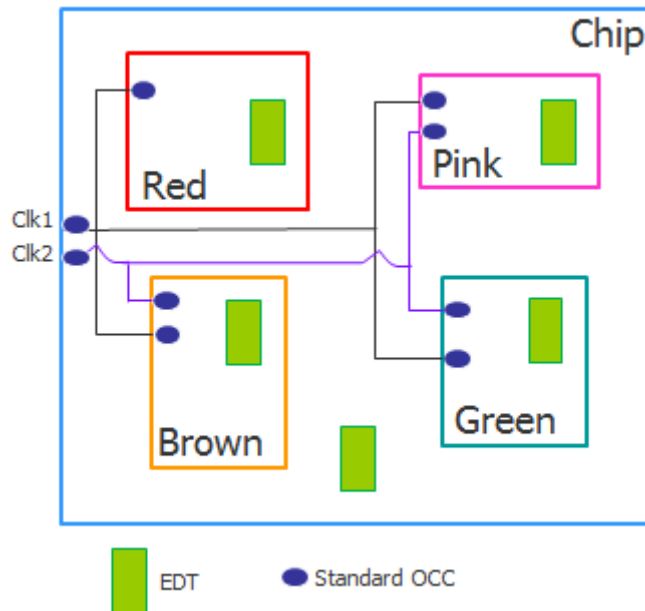
During hierarchical test, the existence of wrapped cores, whose patterns you want to retarget to the chip level, requires a mechanism to enable local, self-contained clock control. This mechanism is the on-chip clock controller (OCC), which you insert into each wrapped core. OCCs inside the wrapped cores make the ATPG patterns self-contained, which enables them to be retargeted.

The OCC is initialized for a clock waveform, and then the applicable patterns are retargeted to the parent- or chip-level design, as applicable. Block-level patterns with block-level OCCs can be retargeted and merged with other block patterns at the parent- or chip-level design independent of the clock waveforms.

Each wrapped core can contain one or more EDT logic blocks that contain the compression logic. Similarly there may be several options for testing the external mode of the cores. In the

following figure, OCC insertion occurs for each clock domain entering the cores. At the chip level, OCCs are also required for the clock domains at this level. Insert one EDT logic block for each internal mode of the core and one EDT logic block for the core's external mode.

Figure 4-4. On-Chip Clock Controller



To help facilitate test setup and automation, use a 1149.1 TAP controller along with an IJTAG network. You may also need to insert a Test Access Mechanism (TAM) to guide how these wrapped cores may be tested.

Standard OCCs include built-in clock selection, clock-chopping, and clock-gating functionality. For more information about standard OCCs, as well as parent and child OCCs, refer “[Tessent On-Chip Clock Controller](#)” in the *Tessent Scan and ATPG User’s Manual*.

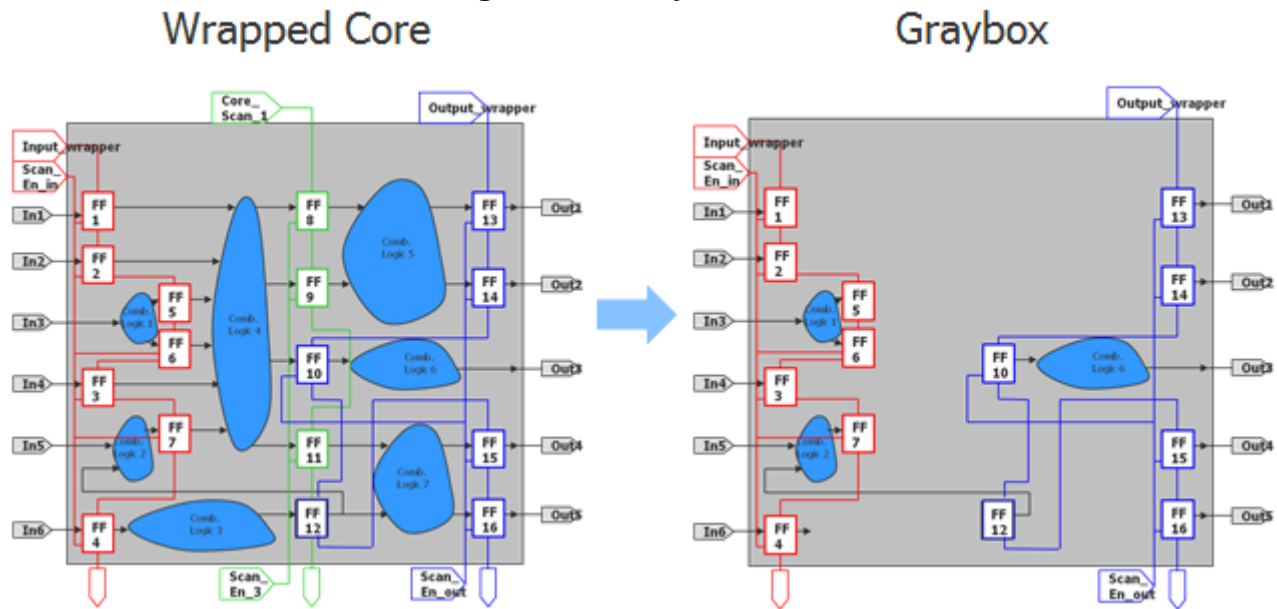
Graybox Model

Graybox models are wrapped core models that preserve the core’s external mode logic, which includes the wrapper chains, along with the portion of the IJTAG network that needs to be tested along with the logic at the next physical level. The purpose of graybox models during hierarchical test is to retain the minimum logic required to generate ATPG patterns for the internal modes of the parent physical blocks. When testing the next physical level, using graybox models enables Tessent Shell to load the design faster than loading the full-chip design.

For details, refer to “[Graybox Overview](#)” in the *Tessent Scan and ATPG User’s Manual*.

The following figure shows a graybox model in which the internal logic of the wrapped core is removed. This graybox model implementation has separate input and output wrapper chains with separate scan enable signals for each wrapper chain type.

Figure 4-5. Graybox Model



The input and output wrapper chains do not have to be separate, and the scan enable signals also do not need to be separate. You can use one scan enable signal with other mechanisms to enable how the wrapper chains operate.

Top-Down Planning Before Bottom-Up Implementation

To optimize your DFT implementation for hierarchical designs, top-down planning is necessary. Top-down planning enables you to properly budget and allocate the resources required for hierarchical test.

As described in “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141, the process for inserting DFT into hierarchical designs is performed in a bottom-up process starting from the lowest level block. However, prior to insertion, it is best to plan the implementation in a top-down manner, starting at the chip level.

To address the design's challenges, consider the following factors:

- Optimizing pattern count to reduce test time
- Optimizing scan volume to fit within allocated tester memory
- Reducing chip pin requirements to fit within tester pin allocation

The planning process requires you to think about priorities such as test time, test quality, flow compatibility, and design schedule. During the planning stage consider the priorities and also the necessary tradeoffs to meet those priorities. In addition, be aware that how you implement DFT at the chip level impacts DFT implementation at the core level.

Clocking Architecture	102
Resource Availability	103
Test Scheduling	103
Specific Tasks That May Require Planning	106
Sample DFT Planning Steps	107

Clocking Architecture

In functional mode, the clock architecture can include clocks that are divided or multiplied. The original, divided, and multiplied clocks can be synchronous with each other within a core, and sometimes they can be synchronous across hierarchical cores. Divided clocks can also be asynchronous across hierarchical cores.

When generating ATPG patterns at the core level that you want to retarget, the core must include an OCC to control clocking as described in “[On-Chip Clock Controller](#)” on page 99. How you implement OCCs in the cores depends on various factors:

- **How the clocks are balanced** — The most common methods for clock balancing are clock tree synthesis and clock mesh synthesis. The method you use can influence what type of OCCs you insert (standard, parent, child).

- **OCC behavior during intest and extest** — How the OCCs need to behave during intest and extest of the cores also dictates how you should implement the OCCs.

Some OCCs are only required to perform clock chopping or clock-gating functionality. Sometimes OCCs may be necessary to mux clocks in for test purposes. You can use chip-level clocking to determine the architecture and OCC types (standard, parent, child) that you should use within the wrapped cores.

Refer to “[Clocking Architecture Examples](#)” on page 785 for examples of clocking architectures with OCC.

Resource Availability

Resource availability plays a significant role in how you implement DFT. For example, at the chip level, the number of pins available for test limits the number of EDT channel pins that you can use. The EDT channel pins may not be associated with only one EDT controller; they could be connected to multiple EDT controllers.

Likewise, the tester may have limited channel pins available for connecting to EDT channel pins on the chip, which means that you may need to consider the tester’s memory allocation for storing the test patterns. Available memory on the tester dictates whether you need to split the pattern set or test multiple cores in parallel.

You want to test the wrapped cores dictates how you create the test-access mechanism (TAM). TAMs carry scan data in and out of the chip for each group of wrapped cores you intend to run in parallel. You need a TAM if you have limited chip pins, and you are reusing those chip pins to connect to multiple EDT controllers inside the wrapped cores.

The TAM logic is dependent on how and when the cores get tested. The TAM schedules the tests for the wrapped cores at the top level, and it enables access to the chip level so that you can run these tests. The TAM also needs input from floor planning and placement of these cores to avoid routing congestion; you need to account for the proximity of the cores that you want to test in parallel and for the location of the chip pins that connect to them.

Test Scheduling

Deciding which wrapped cores need to be tested at the same time—that is, in parallel—depends on a number of factors.

- Number of patterns to be executed
- Availability of chip pins
- Tester pin storage limitations
- Number of identical cores that need to be tested

- Cores with similar pattern count that can be grouped together for execution
- Power dissipation budget and allocation
- Optimal compression at core level
- Channel sharing of cores within modular EDT or sub-chip architectures; cores need to be tested where channels are shared

To achieve the optimal test schedule for a chip with a given number of hierarchical cores, generate an optimized compression for the internal mode of the cores. Within each core, compression could be based on asymmetric channel configurations. For designs with no X sources, fewer output channels are required.

A utility called Compression Advisor can help you obtain the optimal compression. You can access this utility from a Siemens Digital Industries Software application engineer. In addition, you can use the [analyze_compression](#) command to optimize compression. These tools require gate-level scan stitched netlists. BIST logic used for memories needs to be scan stitched and included for the best compression estimation.

The following example shows a schedule based on compression analysis. Suppose 64 pins are available at the chip level for use as channel pins. There are 12 instances of blk_a, eight instances of blk_b, 4 instances of blk_c, and 1 instance each of blk_d, blk_e, and blk_f. The highlighted instances provide the best utilization of resources.

Figure 4-6. Compression Analysis Example

12 instances

Block Name	Input Channels	Output Channels	Shift Length	# Patterns	# Tester Cycles
blk_a	40	2	255	2584	658920
blk_a	28	3	256	3044	779264
blk_a	52	1	255	2692	686460
blk_a	40	2	205	2505	513525
blk_a	40	2	156	2409	375804

8 instances

Block Name	Input Channels	Output Channels	Shift Length	# Patterns	# Tester Cycles
blk_b	56	1	256	1934	495104
blk_b	48	2	256	1939	496384
blk_b	40	3	256	1915	490240
blk_b	32	4	256	1905	487680
blk_b	32	4	202	1828	369256
blk_b	32	4	154	1411	217294

4 instances

Block Name	Input Channels	Output Channels	Shift Length	# Patterns	# Tester Cycles
blk_c	24	10	258	4475	1154550
blk_c	16	12	261	7360	1920960
blk_c	8	14	270	3895	1051650
blk_c	12	13	264	4233	1117512

Block Name	Input Channels	Output Channels	Shift Length	# Patterns	# Tester Cycles
blk_d	28	28	263	2285	600955
blk_d	36	20	260	1996	518960
blk_d	40	16	258	2005	517290
blk_d	38	18	259	1579	408961

Block Name	Input Channels	Output Channels	Shift Length	# Patterns	# Tester Cycles
blk_e	16	16	264	2233	589512
blk_e	26	6	257	2818	724226
blk_e	24	8	258	2112	544896
blk_e	20	12	261	2112	551232
blk_e	22	10	259	2033	526547

Block Name	Input Channels	Output Channels	Shift Length	# Patterns	# Tester Cycles
blk_f	16	16	264	2631	694584
blk_f	22	10	259	2462	637658
blk_f	26	6	257	2415	620655
blk_f	28	4	257	2416	620912
blk_f	31	1	255	2408	614040
blk_f	30	2	256	1918	491008

Suppose you plan to group identical cores instances for testing in parallel. For the blk_a instances, the 40 input channel pins can be broadcast. That is, you can provide the same input data to all the cores. Two output channels from each instance are required for a total of 24 additional pins. When testing the 12 blk_a instances together, you use the 64 available chip pins.

Table 4-1. Test Schedule Example

Scan Config	% Total Faults/ Group	Input Channels	Output Channels	Pins Used/ Config	Shift Length	No. of Patterns	No. of Tester Cycles
12(blk_a)	68.59	40	24	64	156	2409	375804
8(blk_b)	8.54	32	32	64	154	1411	217294
4(blk_c)	9.53	8	56	64	270	3895	1051650
blk_f, blk_e	7.33	52	12	64	259	2033	526547
blk_d	6.01	38	18	56	259	1579	408961
.	Total tester cycles		2646118

Next, test the eight instances of blk_b together. The retargeted patterns are applied at the chip level. The 32 input channel pins broadcast to the eight blk_b instances. In addition, each instance uses four output channels for a total of 64 pins. The tool proceeds to test the four blk_c instances with eight input channels broadcast and 14 output channels pins per instance of blk_c. The blk_f and blk_e instances are tested together because they have similar complexity and channel requirements. The blk_d instances, which require the most channels, are tested by themselves.

Use the following guidelines when developing your test schedule:

1. Group identical core instances and execute them at the same time.
2. Test cores of similar complexity and channel requirements.
3. Test the cores that require the most channels by themselves.

Specific Tasks That May Require Planning

Every design has its unique set of tasks that you need to perform during testing. Some tasks may involve planning while others may only require turning on or off features at the command line.

The following list provides examples of tasks that you may need to plan for ahead of time. This list is not exhaustive.

- Dual-mode configuration for Tessent TestKompress
- Low-power mode and threshold setup

- Multi-load ATPG patterns via the memories
- Test point provisioning for extra scan chains
- Support for multiple power islands and voltage islands
- Functional timing exceptions to be read in with SDC
- Low pin count test (LPCT) controllers
- Test setup sequence to enter test mode

Sample DFT Planning Steps

There are some considerations when planning your DFT implementation.

Note



This example may not be applicable to all designs.

1. Tally how many chip pins are available for scan channels and for ATE channels that are required for test. Use the smaller value when planning for the number of chip pins available for test.
2. Use a dedicated test clock apart from existing functional clocks. This clock can use the TCK signal as the clock source.
3. Assess the functional clock architecture for the chip and plan your OCC configuration. Consider how many OCCs you need, their OCC types (child, parent, standard), and where they need to be inserted for wrapped cores and the chip.
4. Based on the design size and pattern count, determine the number of channels required for each core.
5. Determine test scheduling for cores that you can run in parallel as described in “[Test Scheduling](#)” on page 103.
6. Determine the external mode configuration as described in “[DFT Implementation Strategy](#)” on page 107.
7. Confirm the top-level TAM that is required.

DFT Implementation Strategy

For most designs, a few decisions can make a big difference to the overall DFT implementation architecture.

Multiple Cores Testing in Internal and External Modes

You can test multiple cores in both internal and external modes. You must consider several factors when testing in these modes.

Internal Mode

When you have multiple wrapped cores, fix the scan chain length at a constant value so that at the next hierarchical level when you combine cores to be tested in parallel, the shift length is identical.

If the OCC bits are roughly 25% of the longest chain, stitch OCC bits into a few dedicated compressed chains rather than distributing them across all of the chains. If the OCC bits add up to greater than 75% of the longest chain, then stitch the OCC bits as an uncompressed chain. If there are any remaining OCC bits, connect them into a compressed chain.

If there are embedded boundary scan cells present, ensure they are stitched with the rest of the boundary scan cells at the chip level.

External Mode

The number of external mode chains from across multiple wrapped cores dictates how the chains should be handled, for example:

- Connect the external mode chains to one EDT controller.
- Connect the external mode chains to multiple EDT controllers.
- Connect the external mode chains directly to primary pins without any EDT compression logic. However, this can cause routing congestion.
- Connect the external mode chains when there is EDT compression logic. There may be EDT compression logic built inside the wrapped cores for the purpose of external mode, which leads to having at least two pins (one input and one output) for the compression logic to be used for external mode. This configuration eliminates routing congestion, but each EDT logic block is required to have two pins in the external mode of each core that is connected to the chip level.

If external mode chains from various cores are to be connected to one or more EDT controllers at chip level, then you must ensure that you balance the external mode chains so they are the same length. You can do this by using the chain length across multiple cores.

If there are multiple levels of hierarchy and the OCC needs to be active, you may need to include the OCC bits in the external mode. That is, if you want the OCC to be used in both internal and external modes, then you need to plan for the OCC bits to be included in the external chain also. Typically, OCCs are only included in internal mode.

During ATPG, you can include the boundary scan chain at chip level when running the external mode of the wrapped cores.

Dedicated Test Clock

For hierarchical test, configure the test clock so that it does not share the functional clock port. The test clock needs an extra pulse to initialize the EDT hardware, which disturbs the scan cells and can result in D1 violations that cannot be fixed.

From the test clock, you can use DFT signals to generate the shift capture clock required for OCCs and the EDT clock required for EDT hardware.

Optionally, you can use TCK as your test clock. In this case, the TAP needs to run at the TCK rate, and the shift for ATPG is limited to the TCK frequency. Thus, you need to ensure that you are using an optimal TCK rate.

Using a dedicated test clock aids with timing closure and is beneficial for both internal test and external test because during shift when the scan enable signal is 1, the same clock path is chosen irrespective of whether the core is placed in internal or external mode of operation. For more information, refer to “Shift-Only Mode” under [Clock Control Operation Modes](#) in the *Tessent Scan and ATPG User’s Manual*.

Automated Features Within the Tessent Shell Flow

Tessent Shell provides automated features to help you implement your DFT strategy.

- **TSDB** — The Tessent Shell Data Base is a common database used by all the Tessent products, which means that test information generated by one tool is recognized and usable by downstream tools. For details, refer to “[TSDB Data Flow for the Tessent Shell Flow](#).”
- **IJTAG automation** — As described in “[What Is Tessent Shell?](#)” on page 23, IJTAG provides many benefits, including DFT setup reuse from blocks and sub-blocks to higher levels in the hierarchy. This is essential for using the bottom-up DFT insertion flow as described in “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141.
- **Retargetable ATPG patterns** — Using OCCs inside hierarchical cores makes the ATPG patterns self-contained, which enables them to be retargeted as described in “[On-Chip Clock Controller](#)” on page 99.

Note

 Using the following Tessent automated features depends on your design implementation and how you are performing the two-pass RTL and scan DFT insertion flow for hierarchical designs as described in “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141.

- **Reset control** — Tessent automatically fixes asynchronous resets with pre-DFT DRCs that are enabled when you set the `-logic_test` option of `set_dft_specification_requirements` to “on”. Tessent deactivates the reset during shift

and tests the reset during capture. Optionally, provision is provided to override the reset during capture by deactivating it. This auto-fix is beneficial when the functional reset is generated internally.

- **Boundary scan chain included for ATPG** — At the chip level, you can segment the boundary scan chain into smaller chains to be connected as compressed scan chains and controlled during ATPG. Optionally, you can configure these boundary scan chains to participate during capture or use them during shift.
- **Use memory to target shadow logic faults during logic test** — During logic test insertion, inserted memories get bypassed. The bypass can be built within the memories themselves or built within the MemoryBIST infrastructure. You can use IJTAG-controlled internal Test Data Registers (TDRs) to enable bypass or to use the memories during logic test. By using the memories during logic test, the shadow logic around the memories can be tested. This requires an ATPG model for the memories. For details, refer to the [Comprehensive RAM Primitive](#) information in the *Tessent Cell Library Manual*.
- **Unused scan chains** — During scan insertion and stitching if there are any over-specified scan chains, Tessent Scan automatically adds the unused scan chains with pipeline registers. This is beneficial when not all the scan chains of the inserted EDT compression logic are utilized.

Chapter 5

Tessent Shell Workflows

Tessent Shell workflows are grouped into two main subflows: prelayout and postlayout. The prelayout DFT insertion workflow describes the process for using Tessent Shell for flat and hierarchical designs. The post-insertion validation workflow describes the flow for netlists that have completed placement and routing.

Tessent Shell Flow for Flat Designs	112
Tessent Shell Flow for Hierarchical Designs	141
Tessent Shell Post-Layout Validation Flow	215
Test Bench Generation and Simulation in RTL Mode	224
Hybrid TK/LBIST Flow for Flat Designs	232
Running Multi-Load ATPG on Wrapped Core Memories with Built-In Self Repair .	254
Built-in Self Repair in Hierarchical Tessent MemoryBIST Flow	261

Tessent Shell Flow for Flat Designs

In the RTL and scan DFT insertion flow for flat designs, you perform DFT insertion for the entire chip-level design.

The flat DFT implementation aligns with the physical implementation of the design. The flow consists of the following steps:

1. Performing DFT hardware insertion
2. Synthesis
3. Scan insertion (optional)
4. Gate-level ATPG

Understanding the RTL and scan DFT prelayout insertion flow for flat designs helps you manipulate hierarchical designs or implement a variation of the basic flow.

Refer to the following test case for a detailed usage example of the flow described in this section:

```
tessent_example_flat_flow_<software_version>.tgz
```

You can access this test case by navigating to the following directory:

```
<software_release_tree>/share/UsageExamples/
```


Overview of the RTL and Scan DFT Insertion Flow	113
First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan	116
Second DFT Insertion Pass: EDT, Hybrid TK/LBIST, and OCC	120
Loading the Design	121
Specifying and Verifying the DFT Requirements	123
Creating the DFT Specification	126
Generating the EDT, Hybrid TK/LBIST, and OCC Hardware	130
Extracting the ICL Module Description	130
Generating ICL Patterns and Running Simulation	131
Performing Synthesis	132
Performing Scan Chain Insertion (Flat Design)	133
Performing ATPG Pattern Generation	135
Simulating LBIST Faults	137
Considerations for Using Gate-Level Verilog Netlists	139

Overview of the RTL and Scan DFT Insertion Flow

Whether you are using a flat design or a hierarchical design, the RTL and scan DFT insertion flow requires you to use a two-pass insertion process to insert the DFT hardware.

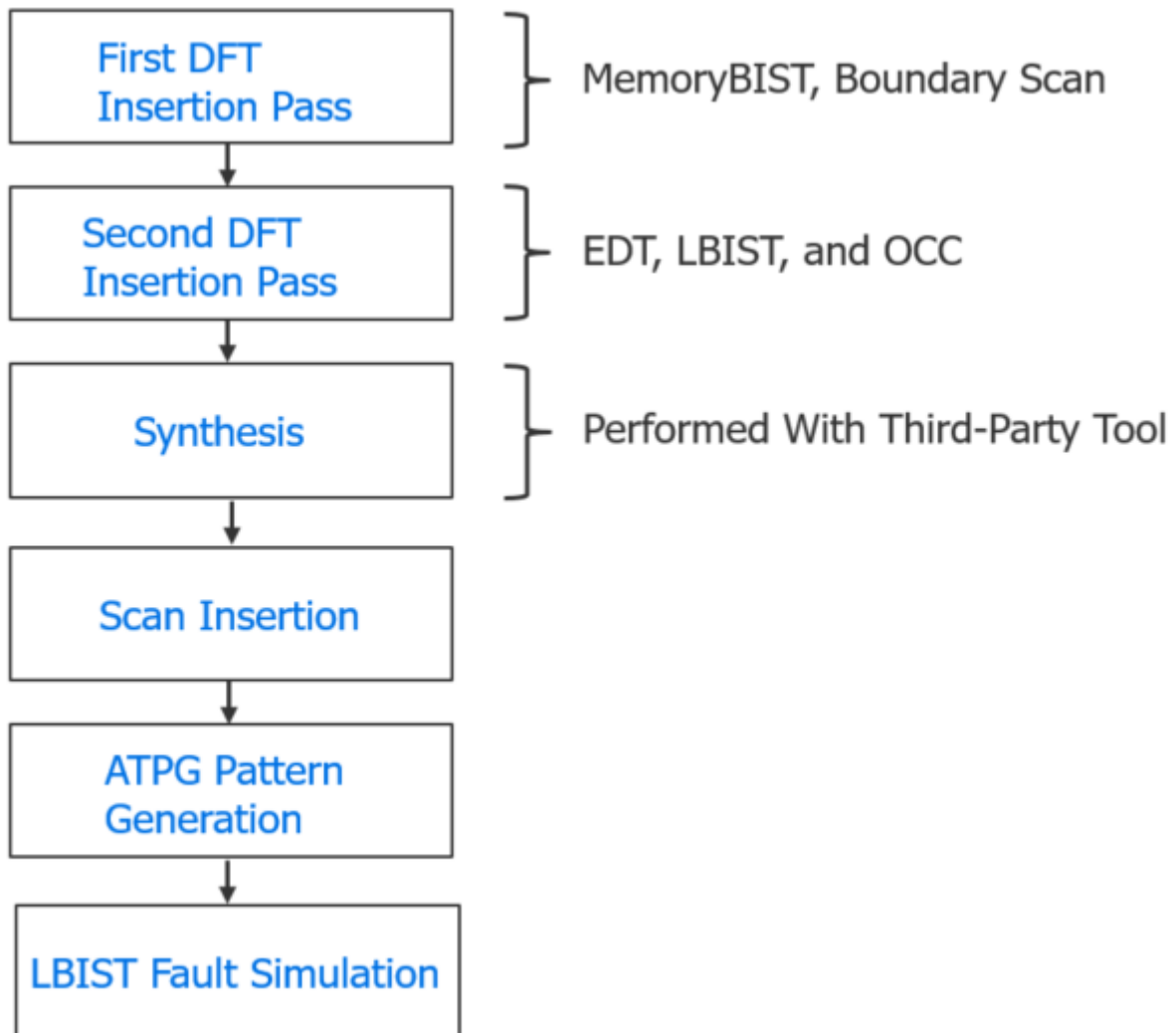
As shown in the following figure, insert MemoryBIST and boundary scan prior to embedded deterministic test (EDT) or hybrid TK/LBIST (LBIST), and the on-chip clock controller (OCC). In this section, EDT refers to embedded deterministic test IP only; and LBIST refers to hybrid TK/LBIST IP, which also includes EDT.

Note

 EDT is a subset of LBIST. You can choose EDT and no LBIST, but you cannot choose LBIST without it also including EDT.

The DFT logic you insert during the first DFT insertion pass gets tested by EDT or LBIST. When inserting DFT into an RTL design, you only need to run synthesis once after you have performed the two DFT insertion passes.

Figure 5-1. Two-Pass Insertion Flow for RTL, Flat Designs



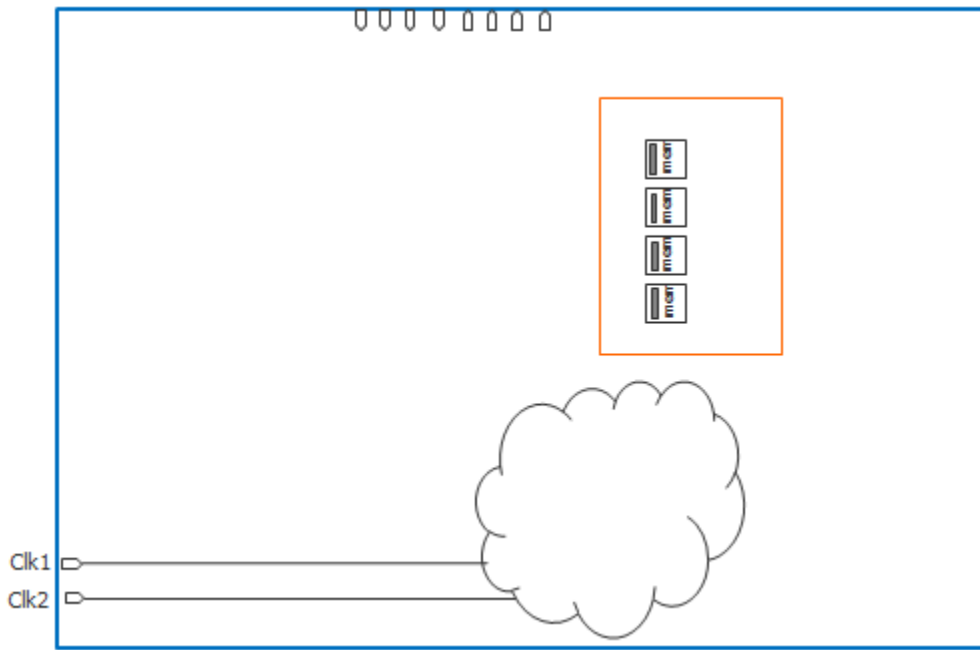
During the first pass, Tessent inserts the IJTAG network and any IJTAG instruments that have ICL descriptions. Refer to the command for details about this process.

During the second pass, the tool checks MemoryBIST's logic for rule compliance with the rest of the functional logic to prevent implications for the DFT signals and IJTAG network connections that could result in coverage loss and pattern count increase.

Optionally, you can perform scan insertion at the same time as synthesis. This does not affect how you perform DFT insertion, but you do lose some of the automation that Tessent Shell provides during ATPG pattern generation.

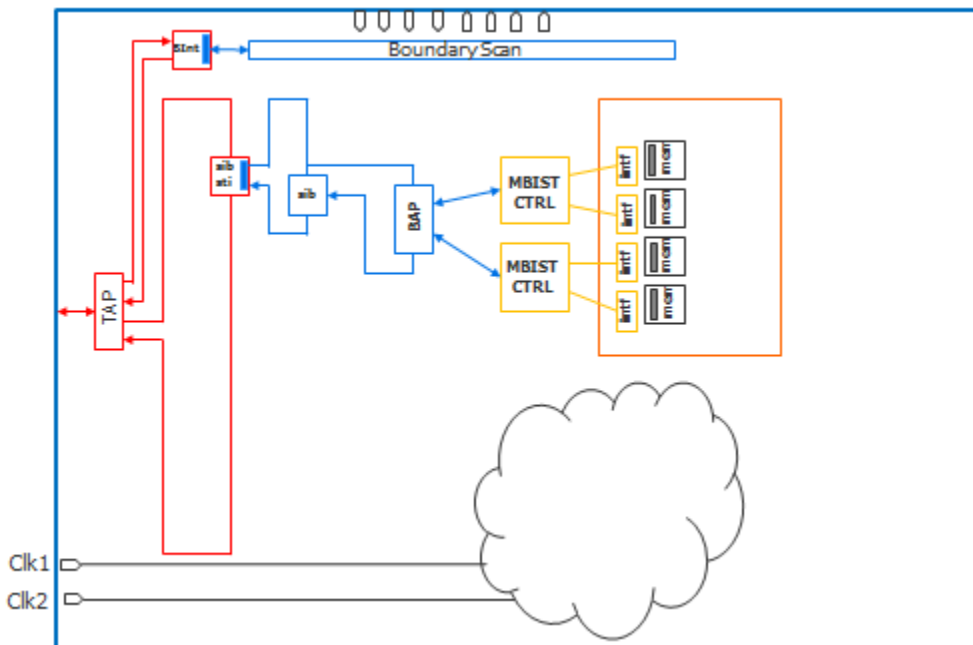
The following figures show an example of the progression of DFT hardware inserted into a DFT-ready design.

Figure 5-2. DFT-Ready Design



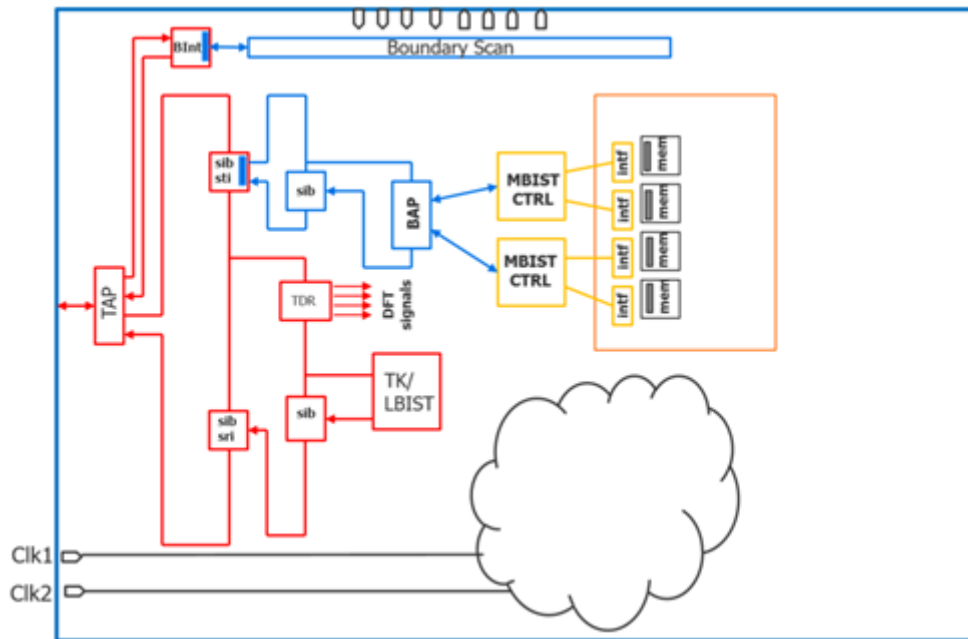
In the first DFT insertion pass, Tessent inserts the MemoryBIST and boundary scan hardware.

Figure 5-3. After the First DFT Insertion Pass



In the second DFT insertion pass, Tessent inserts the EDT or LBIST, and OCC hardware. You clock the MemoryBIST logic (shown in yellow in Figure 5-4) using the same functional clock that feeds the memories. The IJTAG network (blue) is scan tested using the IJTAG clock, which is the TCK clock. The TAP network (red) is not scan tested or is made non-scan during ATPG.

Figure 5-4. After the Second DFT Insertion Pass




First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan

Insert MemoryBIST and boundary scan prior to EDT and OCC so that you can accurately estimate the number of scannable sequential elements (often referred to as “flops”) to be tested. The number of flops determines the size of the EDT controller that Tessent generates in the second DFT insertion pass.

The flow for inserting MemoryBIST and boundary scan together is the same as inserting them separately. For details about this insertion pass flow, refer to:

- “[Getting Started](#)” in the *Tessent MemoryBIST User’s Manual*
- “[Getting Started With Tessent BoundaryScan](#)” in the *Tessent BoundaryScan User’s Manual*

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-1](#) on page 118.

Prerequisites

- To insert boundary scan, you must have an RTL design with instantiated I/O pads if you are using a chip-level design.


- For RTL netlists, you must have a Tessent cell library or the pad library for the I/O pads. For more information, refer to the [Tessent Cell Library Manual](#).

Procedure

1. Load the RTL design data (see lines 1-7).
2. For the first insertion pass, set the `set_context -design_id` switch. By convention, the identifier used for this is typically “rtl1”.

The `-design_id` switch stores all the data associated with a particular DFT insertion pass in the TSDB. For the first pass, rtl1 contains the data for the MemoryBIST and boundary scan hardware, and for the IJTAG network.

Note

 rtl1 is the recommended naming convention for the design ID for the first insertion pass, but you can specify any name. Refer to “[Loading the Design](#)” for more information about setting the design ID.

3. Set the `set_dft_specification_requirements` command to “on” for both `-memory_test` and `-boundary_scan`. (See line 11.)

This tells Tessent Shell that you are generating the MemoryBIST and boundary scan hardware at the same time.

4. Set the design level (`set_design_level chip`, see line 12).

Refer to “[Design Levels](#)” for more information.

5. Identify test pins and apply options to special pins. (See lines 14-26.)
6. Apply the `check_design_rules` command to instruct the tool to leave setup mode and enter analysis mode.


If there are issues with the design, the tool remains in setup mode. (See line 31.)

7. Create the DFT specification. (See lines 33-43.)

- a. To configure the functional pins so that they are shared as EDT channel input and output pins, add the required logic using the AuxiliaryInput ports and AuxiliaryOutput ports wrappers, respectively.

Functional pins can be shared as EDT channel pins. You must insert auxiliary input and output logic at the same time as you insert boundary scan to avoid cascading two multiplexers along the functional output paths. For additional information, refer to the [AuxiliaryInputOutputPorts](#) wrapper in the *Tessent Shell Reference Manual*.

Note

 You can also share `test_clock`, `scan_en`, and `edt_update` pins with functional pins. The functional coverage is maintained when you use boundary scan during scan test.

8. Create the DFT hardware, JTAG network connectivity, and input test patterns. (See lines 46-53.)
9. Run simulations to verify the design. (See lines 55-62.)

Results

For MemoryBIST, Tessent inserts the MemoryBIST controllers, interfaces, [BIST Access Port \(BAP\)](#), and [segment insertion bits \(SIBs\)](#). This hardware is later scan-tested using EDT, which you insert during the second insertion pass. In addition, Tessent automatically connects pre-existing scan testable instruments and scan resource instruments to the [Scan Tested Instrument \(STI\)](#) and [Scan Resource Instrument \(SRI\)](#) sides of the JTAG network, respectively.

For boundary scan, you can segment the boundary scan chain into smaller chains that are used during logic testing with Tessent TestKompress. To segment the boundary scan chain into smaller chains to be connected to the EDT, specify [max_segment_length_for_logicest](#) within the [BoundaryScan](#) wrapper or alternatively specify the following command prior to running `create_dft_specification`:

```
set_defaults_value DftSpecification/BoundaryScan/max_segment_length_for_logicest
```

Examples

The following dofile example shows a typical command flow.

The highlighted command lines illustrate additional considerations for inserting the Tessent Shell MemoryBIST and Tessent Shell Boundary Scan instruments in the first insertion pass of a two-pass DFT insertion process. The functional pins are equipped with logic so that they can be shared as EDT channel input and output pins.

Example 5-1. Dofile Example for MemoryBIST and BoundaryScan

```
1 # Load the design
2 set_context dft -rtl -design_id rtl1
3 set_tsdb_output_directory ../tsdb_rtl
4 read_verilog ../RTL/cpu_top.v
5 read_cell_library ../library/tessent/adk.tcelllib
6 set_design_sources -format tcd_memory -Y ../library/memory \
7   -extension memlib
8 set_current_design cpu_top
9
10 # Specify and verify the DFT requirements
11 set_dft_specification_requirements -memory_test on -boundary_scan on
12 set_design_level chip
13
14 # Identify TAP pins
15 set_attribute_value tck_p -name function -value tck
16 set_attribute_value tdi_p -name function -value tdi
17 set_attribute_value tms_p -name function -value tms
18 set_attribute_value trst_p -name function -value trst
19 set_attribute_value tdo_p -name function -value tdo
20 set_boundary_scan_port_options ramclk_p -cell_options clock
21 set_boundary_scan_port_options reset_p -cell_options sample
22
```

```
23 # Some pins cannot add any boundary scan cells
24 set_boundary_scan_port_options test_clock_p -cell_options dont_touch
25 set_boundary_scan_port_options edt_update -cell_options dont_touch
26 set_boundary_scan_port_options scan_en_p -cell_options dont_touch
27
28 # Specify the clocks feeding the memories
29 add_clocks 0 ramclk_p -Period 5ns
30
31 check_design_rules
32
33 # Create DFT specification
34 set_spec [create_dft_specification]
35 report_config_data $spec
36 set_config_value $spec/BoundaryScan/max_segment_length_for_logictest 150
37 read_config_data -in ${spec}/BoundaryScan -from_string {
38     AuxiliaryInputOutputPorts {
39         auxiliary_input_ports : portain_p[0], portain_p[1], portain_p[2] ;
40         auxiliary_output_ports : portbout_p[0], portbout_p[1],
41         portbout_p[2];
42     }
43 }
44 report_config_data $spec
45
46 # Create the hardware for the DFT inserted with the DFT specification
47 process_dft_specification
48
49 # Extract the IJTAG network connectivity and create ICL for the design
50 extract_icl
51
52 # Create patterns specification for validating the inserted hardware
53 create_patterns_specification
54
55 # Process the patterns specification and create simulation test benches
56 process_patterns_specification
57
58 # Run and check test bench simulations
59 set_simulation_library_sources -v ../library/verilog/adk.v \
60     -v ../library/pad_cells.v -v ../library/memory/ram.v
61 run_testbench_simulations
62 check_testbench_simulations -report_status
```

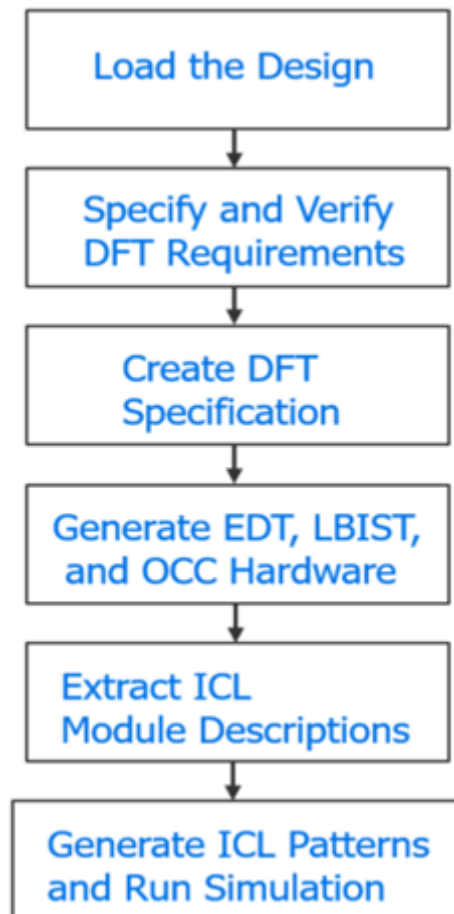
Second DFT Insertion Pass: EDT, Hybrid TK/LBIST, and OCC

In the second DFT insertion pass, insert either the EDT or hybrid TK/LBIST logic test hardware, and OCC. This insertion pass includes requirements and considerations that differ from the first DFT insertion pass, including inserting the DFT signals.


For information about OCC, refer to “[Tessent On-Chip Clock Controller](#)” in the *Tessent Scan and ATPG User’s Manual*.

Before running this flow for chip-level designs, source nodes must be present in the RTL design so that you can define dynamic DFT signals as described in “[Specifying and Verifying the DFT Requirements](#).” Dynamic DFT signals are signals such as scan enable, edt_clock, and edt_update that need to change during specific tests.

Figure 5-5. Flow for the Second DFT Insertion Pass



Note

 For the second DFT insertion pass, use the process described in “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” on page 116 to generate ATPG patterns and perform simulation.

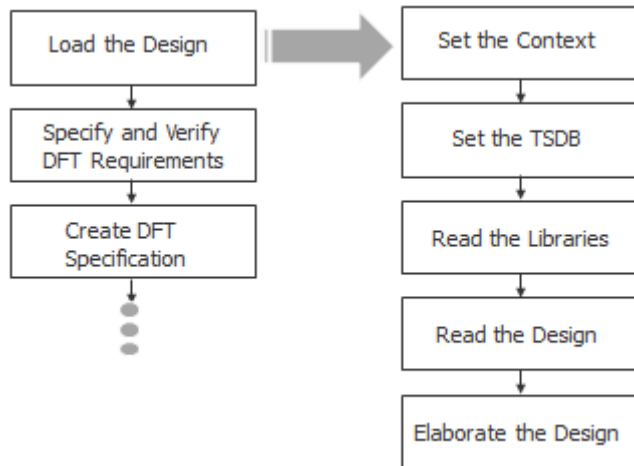
Loading the Design **121**
Specifying and Verifying the DFT Requirements...... **123**
Creating the DFT Specification **126**
Generating the EDT, Hybrid TK/LBIST, and OCC Hardware **130**
Extracting the ICL Module Description **130**
Generating ICL Patterns and Running Simulation **131**

Loading the Design

When loading the design for the second DFT insertion pass, you must ensure that you specify a new design ID name and, optimally, use the same TSDB directory that you used in the first insertion pass. In addition, you must read in the design from the first insertion pass and elaborate the design.

The design loading process for the second pass is similar to the process you used in the first insertion pass, with a few exceptions.

Figure 5-6. Flow for Loading the Design



Prerequisites

- For chip-level designs, source nodes must be present in the RTL design so that you can define dynamic DFT signals as described in “[Specifying and Verifying the DFT Requirements.](#)” Dynamic DFT signals are signals such as scan enable, edt_clock, edt_update, and so on, that need to change during specific tests.


Procedure

1. Apply the `set_context` command with the ID “rtl2” for the second pass. For example:

```
set_context dft -rtl design_id rtl2
```

In the first DFT insertion pass, you set the design ID to “rtl1” with the `command`.

Note

 This manual uses recommended naming conventions for the design IDs for flat designs, which are “rtl1” for the first DFT insertion pass, “rtl2” for the second DFT insertion pass, and “gate” for the scan chain insertion pass. Refer to “[Considerations for Using Gate-Level Verilog Netlists](#)” for the naming conventions when using gate-level Verilog netlists.

For a specified design, the design ID stores all the data associated with a DFT insertion pass into the TSDB. For the first pass, “rtl1” contains the data for the MemoryBIST and boundary scan hardware. Setting the `design_id` to `rtl2` at the beginning of the second DFT insertion pass identifies that “rtl2” stores the EDT or LBIST, and OCC hardware data generated during the second pass.

The `rtl2` design data is cumulative. That is, it contains the necessary `rtl1` data in addition to the new data generated for EDT or LBIST, and OCC. The “rtl2” designation indicates that the second insertion pass is performed on the resulting edits of the first pass RTL data. In subsequent insertion passes, you can use either design ID to load the design and its supporting files.

Tessent generates IJTAG nodes during both insertion passes and their module names are differentiated using the design ID you specified in each pass.

2. Specify the `set_tsdb_output_directory` command with the same directory location for both the first and second DFT insertion passes.

```
set_tsdb_output_directory ../tsdb_rtl
```

If you forget to specify the command for the second DFT insertion pass, Tessent creates a default `tsdb_outdir` directory in the current working directory. If you use a different output TSDB directory for the two insertion passes, ensure that you open the TSDB used by the first insertion pass by specifying the `open_tsdb` command.

For more information about the TSDB, refer to “[Tessent Shell Data Base \(TSDB\)](#)” in the *Tessent Shell Reference Manual*. For information about the TSDB data flow, refer to “[TSDB Data Flow for the Tessent Shell Flow](#).”

3. Specify the `read_cell_library` command to read in the library file for the standard cells and the pad I/O macros.

The following command reads the Tessent cell library file for the standard cells and pad I/O macros:

```
read_cell_library ../library/adk_complete.tcelllib
```

If the pad I/O library and standard cell library are separate, use the following commands to read in the atpg.lib files and the library for the pad I/O description:

```
read_cell_library ../library/atpg.lib
read_cell_library ../library/pad.library
```

- Specify the [read_design](#) command to load the design:

```
read_design cpu_top -design_id rtl1
```

The design was created in the first DFT insertion pass when you used the `read_verilog` or `read_vhdl` commands. The `read_design` command also loads supporting files such as the TCD, ICL, and PDL, if present in the TSDB.

The design is read in using the design ID from the first DFT insertion. In this case, the design ID is “rtl1”

To load the design correctly for the second insertion pass, the `read_design` command refers to the design source dictionary that was created during the first DFT insertion pass and stored in the [dft_inserted_designs](#) directory.

- Specify the [set_current_design](#) command to elaborate the design.

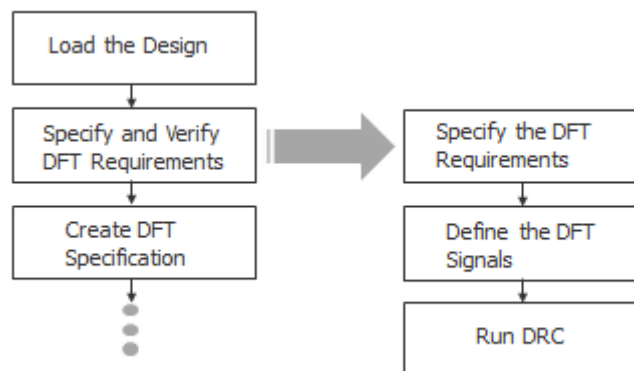
```
set_current_design cpu_top
```

If any module descriptions are missing, design elaboration identifies them. You can fix elaboration errors by adding the missing modules or by specifying the “[add_black_boxes -module](#)” command.

Specifying and Verifying the DFT Requirements

After loading your design data, define the DFT requirements for the EDT or LBIST, and OCC hardware you want to insert. The requirements definition includes adding DFT signals followed by performing design rule checking.

Figure 5-7. Flow for Specifying and Verifying the DFT Requirements



Procedure

1. Specify the [set_dft_specification_requirements](#) command to run pre-DFT design rule checking as follows:

```
set_dft_specification_requirements -logic_test on
```

Because you already specified that you were working at the chip level in the first DFT insertion pass, you do not need to specify this information for the second insertion pass.

2. Specify the [add_dft_signals](#) command to define the DFT signals. For example:

```
add_dft_signals ltest_en  
...
```

See “[DFT Signals](#)” on page 124 for a more complete example.

DFT signals are used to determine the various modes of operation, control values, and signal behaviors that are necessary for each test mode. The DFT signals capability in Tessent Shell automates the following tasks:

- Adding DFT signals
- Configuring DFT signals for various modes of operation
- Transferring information about DFT signals from one step to the next

Based on the specified mode of operation, Tessent creates the necessary setup procedures to control DFT signals through an IJTAG network.

3. Specify the [check_design_rules](#) command to run pre-DFT design rule checking.

Tessent Shell generates DFT_C violations when error conditions are detected. For details, refer to “[Pre-DFT Clock Rules \(DFT_C Rules\)](#)” in the *Tessent Shell Reference Manual*.

Results

When DRC passes, Tessent Shell shifts from setup mode to analysis mode. Pre-DFT DRC verifies that clocks are defined for all of the scannable sequential elements to be scan-tested and identifies the async sets and resets so that they can be turned off during shift operations. In addition, if you have specified the `add_dft_clock_enable` command, Tessent checks clock-gating logic and module-type clocks.

Examples

DFT Signals

You can add static DFT signals and dynamic DFT signals. Static DFT signals include global DFT control, logic test control, and scan mode signals. As described in the [add_dft_signals](#) command description, these DFT signals are typically controlled by a Test Data Register that is part of the IJTAG network.

Most dynamic DFT signals originate from primary input ports. For chip-level designs, these primary input ports must already be present in the RTL and be pre-connected to a pad buffer cell. The three dynamic DFT signals that originate from primary inputs are `test_clock`, `scan_en`, and `edt_update`. To share their input ports with the functional mode, ensure that you added auxiliary input logic for them during boundary scan insertion. Tessent cannot create the nodes as ports.

The following example shows the required DFT signals for the second insertion pass when you are inserting EDT or LBIST, and OCC.

```
// Create the static DFT signals
add_dft_signals ltest_en

// Generate dynamic DFT signals from source nodes
add_dft_signals scan_en edt_update test_clock \
    -source_node { porta_in1 portb_in1 porta_in2 }

// Create shift_capture_clock and edt_clock as gated versions of
// test_clock
add_dft_signals shift_capture_clock edt_clock -create_from_other_signals

// Used by top-level EDT
add_dft_signals edt_mode

// Required for boundary scan to be used with logic test without
// contacting inputs
add_dft_signals int_ltest_en output_pad_disable

// Required for scan-tested instruments (STI network) such as MemoryBIST
// and boundary scan
add_dft_signals tck_occ_en

// To bypass memories or to run multi-load ATPG for memories
add_dft_signals memory_bypass_en
```

Note



Tessent Shell automatically recognizes scan-tested instruments and stitches them into scan chains.

Refer to the *Tessent Shell Reference Manual* for information about the following commands:

- **add_dft_signals** — Insert DFT signals.
- **register_static_dft_signal_names** — Register your DFT signal, if, for example, you need to augment the default DFT signals for specific usage requirements.
- **report_dft_signals** — View a list of the DFT signals added with the `add_dft_signals` command.
- **delete_dft_signals** — Delete any previously specified DFT signals.

- **add_dft_modal_connections** — Though not normally required for flat designs, if you have multiple EDT configurations that you want to multiplex into a common set of top-level I/Os, use this command to implement the multiplexing. In addition, if the EDT channel pins are shared with functional pins, they need to include auxiliary input/output muxing logic.

Related Topics

[add_dft_signals](#)

[register_static_dft_signal_names](#)

[report_dft_signal_names](#)

[report_dft_signals](#)

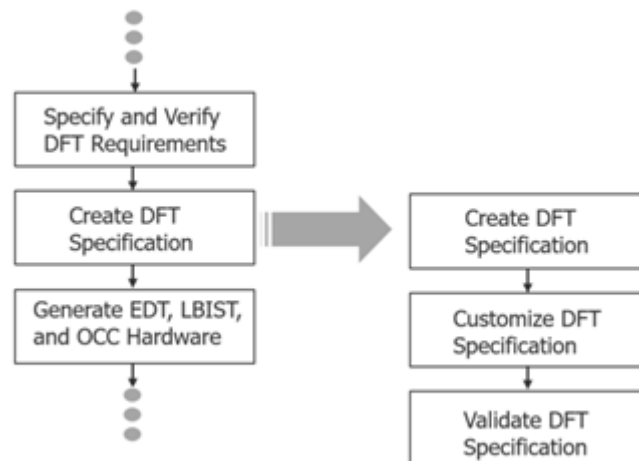
[delete_dft_signals](#)

[add_dft_modal_connections](#)

Creating the DFT Specification

After defining the DFT requirements for the DFT signals in setup mode, you are ready to create the DFT specification for EDT or LBIST, and OCC in analysis mode. The DFT specification defines how the tool inserts hardware into the design.

Figure 5-8. Flow for Creating the DFT Specification



Prerequisites

- For EDT or LBIST, and OCC, you must first generate a skeleton DFT specification that contains three empty SRI SIBs that specify the EDT, LBIST, and OCC sections of the IJTAG network. Creating the DFT specification for EDT or LBIST, and OCC differs from the process you use for MemoryBIST and boundary scan in the first DFT insertion pass.

Procedure


1. Specify the `create_dft_specification` command as follows:

```
create_dft_specification -sri_sib_list {occ edt lbist}
```


2. Apply commands to customize the DFT specification using one of the following interfaces:

To customize the DFT specification on the command line, type the EDT or LBIST, and OCC data as an argument to the `read_config_data -from_string` command. Modify the DFT specification with introspected data using the `add_config_element` and `set_config_value` commands. Tessent automatically saves modifications to the `dofile/scripts` directory for use in future sessions. See “[DFT Customization Example](#)” on page 127 for an example.

Note

 To input the EDT or LBIST, and OCC wrapper details for the DFT specification, you can either use the Tessent GUI, known as Tessent Visualizer (see “[Customizing the DFT Specification for EDT](#)”), or the Tessent Shell command line.

Note

 Tessent automatically connects the divided boundary scan segment (if present from the first DFT insertion pass) to the EDT hardware that Tessent inserts in the second insertion pass. Refer to [connect_bscan_segments_to_lsb_chains](#) for details.

3. Specify the following command to ensure that no errors exist in the DFT specification:

```
process_dft_specification -validate_only
```


Complete this step before generating the EDT, LBIST, and OCC hardware.

Examples

DFT Customization Example

The following example modifies a DFT specification to add LBIST (including EDT) and OCC and saves the changes.

Note

 If you are using Tessent OCC, Tessent Scan automatically identifies and stitches the sub-chains in the OCC into scan chains.

```

read_config_data -in $spec -from_string {
  OCC {
    ijtag_host_interface : Sib(occ);
  }
}
# The following illustrates a generic way to populate the OCC. The clock
# list is design specific and needs to be updated for the design to the
# OCC, scan_enable and shift_capture_clock are connected automatically
# Modify the following for your specific design requirements
set id_clk_list [list \
  clk1 clk1_p \
  clk2 clk2_p \
  clk3 clk3_p \
  clk4 clk4_p \
  ramclk_p ramclk_p \
]

foreach {id clk} $id_clk_list {
  set occ [add_config_element OCC/Controller($id) -in $spec]
  set_config_value clock_intercept_node -in $occ $clk
}
report_config_data $spec

# To the EDT controller, the edt_clock and edt_update are connected
# automatically. The EDT controller is built-in with bypass
# Modify the following for your specific design requirements
read_config_data -in $spec -from_string {
  EDT {
    ijtag_host_interface : Sib(edt);
    Controller (c1) {
      longest_chain_range : 65, 100;
      scan_chain_count : 85;
      input_channel_count : 3;
      output_channel_count : 3;
      Connections +{
        EdtChannelsIn(1) {
        }
        EdtChannelsIn(2) {
        }
        EdtChannelsIn(3) {
        }
        EdtChannelsOut(1) {
        }
        EdtChannelsOut(2) {
        }
        EdtChannelsOut(3) {
        }
      }
    }
  }
}

```



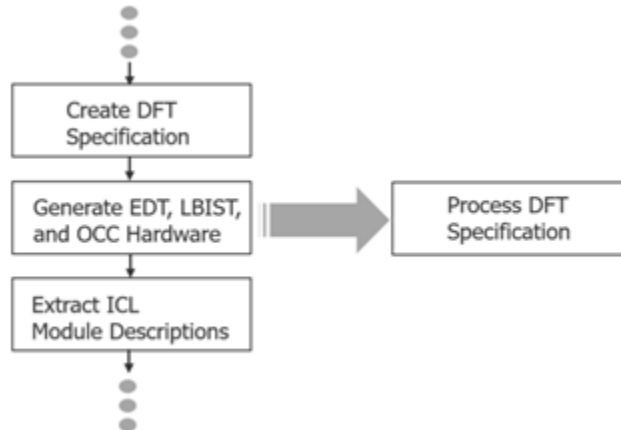
```
# Connecting the EDT channel in to the primary input and connecting
# EDT channel out to the primary output
# You had inserted the auxiliary logic for these ports during the first
DFT insertion pass
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsIn(1) \
[get_single_name [get_auxiliary_pins portain_p[0] -direction input]]
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsIn(2) \
[get_single_name [get_auxiliary_pins portain_p[1] -direction input]]
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsIn(3) \
[get_single_name [get_auxiliary_pins portain_p[2] -direction input]]
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsOut(1) \
[get_single_name [get_auxiliary_pins portbout_p[0] -direction output] ]
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsOut(2) \
[get_single_name [get_auxiliary_pins portbout_p[1] -direction output] ]
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsOut(3) \
[get_single_name [get_auxiliary_pins portbout_p[2] -direction output] ]
report_config_data $spec
```

```
LogicBist {
  ihtag_host_interface : Sib(lbist) ;
  Controller(C0) {
    ShiftCycles {
      max          : 100 ;
    }
    CaptureCycles {
      max          : 10 ;
    }
    PatternCount {
      max          : 16384;
    }
    Connections {
      shift_clock_src : clk1;
    }
  }
  NcpIndexDecoder {
    Ncp(no_pulse) {
    }
    Ncp(pulse_clk1) {
      cycle(0) : $occ;
    }
    Ncp(pulse_clk2) {
      cycle(0) : $occ;
      cycle(1) : $occ;
    }
  }
}
```

Generating the EDT, Hybrid TK/LBIST, and OCC Hardware

After creating the DFT specification, generate the EDT or hybrid TK/LBIST, and OCC hardware. The `process_dft_specification` command generates and inserts the DFT hardware, as defined by the DFT specification, into your design.

Figure 5-9. Flow for Inserting the Second-Pass Hardware



Procedure

1. Specify the `process_dft_specification` command to insert EDT or LBIST, and OCC in the second pass.

```
process_dft_specification
```

2. (Optional) If you want to generate the hardware, but not insert it into the design, specify the following command:

```
process_dft_specification -no_insertion
```

You can then insert the hardware into the design manually.

Extracting the ICL Module Description

After inserting the EDT or LBIST, and OCC hardware, verify the connectivity of the ICL modules that were inserted with the `process_dft_specification` command. This is a preparatory step for ICL pattern generation.

Figure 5-10. Flow for Extracting ICL



Procedure

1. Specify the [extract_icl](#) command to find all modules (both Tessent instruments and non-Tessent instruments) and their associated ICL descriptions, and to run DRC to verify their connectivity.

The top-level ICL description corresponds to the design name you specified with the `set_current_design` command during the first insertion pass (which is also the same design name you specified when you elaborated the design at the beginning of the second insertion pass).

2. Specify the [analyze_drc_violation](#) command to debug the DRC violations.

Tessent generates I-rule errors when it detects ICL extraction errors and opens Tessent Visualizer to a schematic view of the error. For more information about the DRC I-rule errors, refer to “[ICL Extraction Rules \(I Rules\)](#)” in the *Tessent Shell Reference Manual*. For details about using Tessent Visualizer, refer to “[Tessent Visualizer](#)” on page 625.

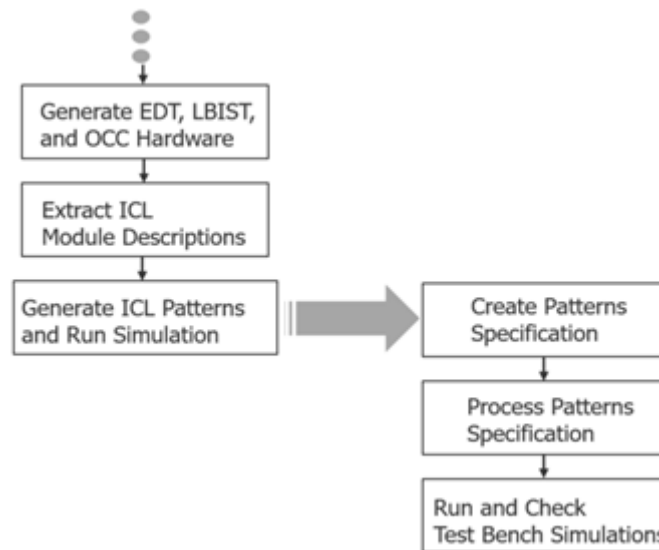
The `extract_icl` command also creates a Synopsys Design Compiler file that you can use for synthesis. Refer to “[Timing Constraints \(SDC\)](#)” for more information.

Generating ICL Patterns and Running Simulation

Generate and process the ICL verification patterns for EDT or LBIST, and OCC, then verify the test bench simulations.

The following figure shows the tasks required to complete the second insertion pass and move on to synthesis.

Figure 5-11. Flow for Generating and Simulating ICL Patterns



Procedure

1. Generate the test patterns for the design:

```
create_patterns_specification
```

2. Validate and process the content of the test patterns:

```
process_patterns_specification
```

3. Run and check test bench simulations. For example:

```
# Run and check testbench simulations
set_simulation_library_sources -v ../library/verilog/adk.v \
    -v ../library/pad_cells.v -v ../library/memory/ram.v

run_testbench_simulations
check_testbench_simulations -report_status
```

Performing Synthesis

Synthesize the original RTL and the DFT-inserted RTL for MemoryBIST, boundary scan, EDT or LBIST, and OCC. For RTL designs, perform synthesis once after performing the first and second DFT insertion passes.

Prerequisites


- For information regarding synthesis with third-party tools and Tessent DFT methodologies, refer to “[Synthesis Guidelines for RTL Designs with Tessent Inserted DFT](#)” on page 779.
- Tessent Shell supports synthesis using Synopsys Design Compiler. Refer to “[Timing Constraints \(SDC\)](#)” for more information.

Procedure

Specify the `write_design_import_script` command to create a `dc_shell` design load script that you can use to load the RTL design as it exists after the two DFT insertion passes. For example:

```
write_design_import_script for_synthesis.tcl -replace
```

Note

 If you are not using Synopsys Design Compiler as your third-party synthesis tool, you can still use the command to create a `dc_shell` script and adjust it to match your synthesis tool's command set.

```
# You can generate the synthesis script only without running
# synthesis using the following command
# run_synthesis -startup_file ./synopsys_dc.setup
# To generate a script for third-party synthesis
# run_synthesis -generate_script_only
```

Performing Scan Chain Insertion (Flat Design)

During scan chain insertion, Tessent inserts and stitches scan chains on the gate-level netlist that was synthesized after DFT insertion. After you run scan chain insertion, proceed to ATPG pattern generation.

During scan insertion using Tessent Scan, the non-scan instances such as EDT are automatically understood. In addition, the built-in OCC sub-chains are understood and stitched into scan chains. Tessent uses DFT signals such as the scan enable that you specified in the previous insertion passes, therefore you do not need to specify the DFT signals again.

For information about scan chain insertion using Tessent Shell, refer to the “[Internal Scan and Test Circuitry Insertion](#)” in the *Tessent Scan and ATPG User's Manual*.

Procedure

1. Specify the following command to set the DFT context:

```
set_context dft -scan -design_id gate
```

When setting the context, ensure that you specify the design ID with a unique name. The recommended name is `gate` for a flat design. For a gate-level netlist the recommended name is `gate3`.

2. Load the synthesized netlist. For example:

```
../Synthesis/cpu_top_synthesized.vg
```

This netlist contains the gates for the original RTL design and the DFT-inserted hardware.

3. Specify the same output directory you used in the first and second DFT passes:

```
../tsdb_rtl
```

4. Load the rtl2 design data for the DFT hardware that you inserted. For example:

```
cpu_top -design_id rtl2 -no_hdl
```

The `-no_hdl` switch specifies to read in all of the DFT data files—such as ICL, PDL, and TCD—except for the design files. (You are using the synthesized design from this point forward.)

After design elaboration and design rule checking, Tessent Shell transitions from Setup mode to Analysis mode.

5. Specify the `add_scan_mode` `edt_mode` command to connect the scan chains to the EDT signals and EDT hardware that you inserted during the second insertion pass.

The use of preregistered DFT signal `edt_mode` as the scan mode using the `add_scan_mode` command infers the `-enable_dft_signal` also as the `edt_mode` DFT signal.

Results

The scan stitched and inserted netlist is located in the TSDB under the design ID “gate” for RTL designs or “gate3” for gate-level netlists. Refer to “[Considerations for Using Gate-Level Verilog Netlists](#)” for details.

Examples

The following dofile shows a command flow for scan insertion and stitching:

```
set_context dft -scan -design_id gate
read_cell_library ../library/tessent/adk.tcelllib
read_verilog ../Synthesis/cpu_top_synthesized.vg
set_tsdb_output_directory ../tsdb_rtl
read_design cpu_top -design_id rtl2 -no_hdl
set_current_design cpu_top

check_design_rules

set edt_instance [get_name_list [get_instance -of_module \
                               [get_name [get_icl_module -filter \
                                             tessent_instrument_type==mentor::edt]]]]
add_scan_mode edt_mode -edt_instance $edt_instance
analyze_scan_chains
report_scan_chains
insert_test_logic
exit
```

Related Topics

[read_verilog](#)

[set_tsdb_output_directory](#)

[read_design](#)

Performing ATPG Pattern Generation

After inserting scan chains, proceed to ATPG pattern generation. You can create patterns for various fault model types, including stuck-at, transition, path delay, and IDDQ.

Procedure

1. Do one of the following:
 - If you used Tessent Scan to insert the scan chains, run the “[import_scan_mode edt_mode](#)” command for ATPG pattern generation.
 - If you did not use Tessent Scan for scan insertion, use the TCD IP mapping flow as described in “[Running ATPG Patterns without Tessent Scan](#)” in the *Tessent Scan and ATPG User’s Manual*.

When you run the `import_scan_mode` command, Tessent Shell passes through the scan-insert design data for the EDT or LBIST, and OCC logic. This data includes the scan structures (scan chains and scan channels) that are stored in the TSDB under the “gate” design ID. The gate design ID was created during scan insertion when you specified the `insert_test_logic` command.

In addition, Tessent automatically creates and simulates the `test_setup` procedure cycles that are required to initialize the EDT or LBIST, and OCC static signals. You only need to specify non-default parameter values, if, for example, you run EDT with `bypass` on or set `int_ltest_en` to 1 to use the boundary scan as the source of the core values and isolate the ATPG test from the top-level IOs.

You can create ATPG patterns for any mode that you need, such as stuck-at and transition. These patterns are stored in the TSDB database under the `logic_test_cores` directory. The `import_scan_mode` command uses the same scan configuration—that is, the DFT signals—that were used for the `add_scan_mode` command during scan insertion.

2. Specify the `set_current_mode` command to indicate the type of pattern you are generating (see “[Stuck-at ATPG patterns](#)” on page 136 and “[Transition at-speed ATPG patterns](#)” on page 136).

In addition, the name you give to the generated ATPG pattern sets must differ from the mode name you specify for `import_scan_mode` (that is, “`edt_mode`”).

3. Specify the `write_patterns` command to write out the Verilog test benches and STIL patterns.
4. Specify the `write_tsdb_data` command to save the flat model, fault list, PatDB, and TCD files into the TSDB.

For details about ATPG pattern generation, refer to “[Running ATPG Patterns](#)” in the *Tessent Scan and ATPG User’s Manual*.

Examples

Stuck-at ATPG patterns

The following example generates stuck-at ATPG patterns as indicated by the `set_current_mode edt_stuck` command. It shows that you have a choice between using the boundary scan chains for capture during ATPG or using the primary pins of the chips (using the pads).

```
set_context patterns -scan
read_cell_library ../library/tessent/adk.tcelllib
read_cell_library ../library/memory/memory.lib
open_tsdb ../tsdb_rtl
read_design cpu_top -design_id gate
set_current_design cpu_top
set_current_mode edt_stuck
import_scan_mode edt_mode

# To apply the patterns through the boundary scan chains and not through
# the pads use:
# set_static_dft_signal_values int_ltest_en 1
# set_static_dft_signal_value output_pad_disable 1
# To allow the shift_capture_clock during capture phase on Scan Tested
# Instruments:
set_system_mode analysis
create_patterns
write_tsdb_data -replace
write_patterns patterns/cpu_top_stuck_parallel.v -verilog -parallel \
    -replace -scan -parameter_list {SIM_KEEP_PATH 1}
write_patterns patterns/cpu_top_stuck_serial.v -verilog -serial -replace \
    -parameter_list {SIM_KEEP_PATH 1}
exit
```

Transition at-speed ATPG patterns

The following example generates transition at-speed patterns. The `import_scan_mode` command uses the `-fast_capture_mode` switch to indicate that the OCC supplies the fast clock during capture. The `set_current_mode` command indicates `edt_transition`, and the `set_fault_type` command indicates transition faults.


```
set_context patterns -scan
read_cell_library ../library/tessent/adk.tcelllib
read_cell_library ../library/memory/memory.lib
open_tsdb ../tsdb_rtl
read_design cpu_top -design_id gate
set_current_design cpu_top
set_current_mode edt_transition
import_scan_mode edt_mode -fast_capture_mode on -verbose

set_static_dft_signal_values int_ltest_en 1
set_static_dft_signal_value output_pad_disable 1
set_system_mode analysis
set_fault_type transition
set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]
create_patterns
write_tsdb_data -replace
write_patterns patterns/cpu_top_transition_parallel.v -verilog \
               -parallel -replace -scan -parameter_list {SIM_KEEP_PATH 1}
write_patterns patterns/cpu_top_transition_serial.v -verilog -serial \
               -replace -parameter_list {SIM_KEEP_PATH 1}
exit
```

Simulating LBIST Faults

If your design contains logic BIST, you must run LBIST fault simulation after scan insertion.

Prerequisites

- Scan insertion has been completed.

Procedure

1. Set the context:

```
set_context pattern -scan -design_id rtl2
```

2. Load the cell and memory libraries:

```
set_tsdb_output_directory tsdb_outdir
read_cell_library ../prereq/techlib_adk.tnt/tessent/adk.tcelllib
read_cell_library design/mem/mems.atpglib
```

3. Load the scan-inserted gate-level design:

```
read_design top -design_id rtl2
```

4. Elaborate the design:

```
set_current_design top
```

5. Import the scan insertion data:

```
import_scan_mode int_mode
```

6. Define the LBIST core instance:

```
add_core_instances -module top_dft2_tessent_lbist
```

7. Define DFT signals that differ from their default values:

```
set_static_dft_signal_values          tck_occ_en 1
set_static_dft_signal_values          int_ltest_en 1
set_static_dft_signal_values          ltest_en 1
set_static_dft_signal_values control_test_point_en 1
set_static_dft_signal_values observe_test_point_en 1

set_core_instance_parameters -instrument_type occ -parameter_values
{static_clock_control external}
```

8. Point to the dofile containing the proc definitions:

```
dofile tsdb_outdir/instruments/
top_dft2_lbist_ncp_index_decoder.instrument/
top_dft2_tessent_lbist_ncp_index_decoder.dofile

set_static_dft_signal_values x_bounding_en 1
```

9. Add required pin constraints and output masks:

```
add_input_constraints [get_ports {[ABC].*} -regexp] -CX
add_output_masks [get_ports Y*]

report_static_dft_signal_settings
```

10. Run rule checks:

```
set_system_mode analysis
report_scan_cells> scan_cells_fsim.rpt
report_clocks
report_pin_constraints
```

11. Read the test proc file containing NCP definitions:

```
read_procfiler tsdb_outdir/instruments/  
top_dft2_lbist_ncp_index_decoder.instrument/  
top_dft2_tessent_lbist_ncp_index_decoder.testproc
```

12. Run pattern fault simulation:

```
add_faults -all  
set_random_patterns 4096  
simulate_patterns -source bist -store all
```

13. Report the test coverage and other test data:

```
report_statistics
```

14. Write the patterns using the [write_patterns](#) or [write_tsdb_data](#) command:

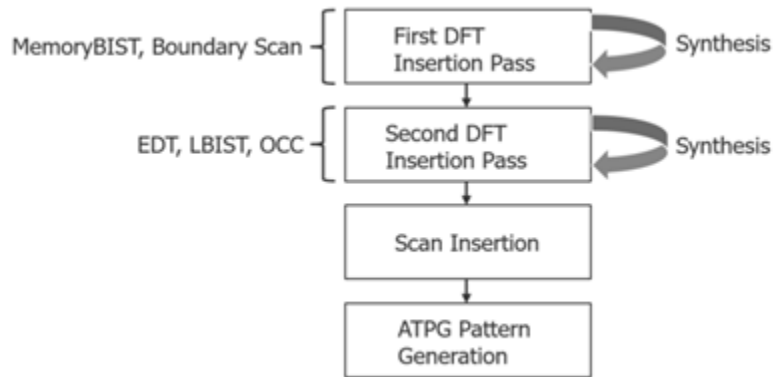
```
#write_patterns elt1_lbist_patt_parallel_ncp.v -verilog -parallel -  
replace -mode_internal -begin 0 -end 260  
write_tsdb_data -rep  
  
report_lbist_configuration -hardware_default_compatibility  
  
exit
```

Considerations for Using Gate-Level Verilog Netlists

In some cases, you may only have a gate-level Verilog netlist, not the RTL. You can use Tessent Shell to perform the DFT insertion flow when you have a gate-level netlist. The flow is similar to the RTL and scan DFT insertion flow. The main difference is that you must perform synthesis after each DFT insertion pass.

[Figure 5-12](#) shows the gate-level DFT insertion sequence in which you insert MemoryBIST and boundary scan in the first insertion pass and then immediately run synthesis on these instruments before inserting the logic test instruments for the second DFT insertion pass (EDT, LBIST, OCC, and so on). After inserting the logic test instruments, run synthesis a second time before proceeding to scan chain insertion.

Figure 5-12. Two-Pass Insertion Flow for RTL, Gate-Level Designs



The following differences apply when using a gate-level Verilog netlist rather than RTL. Other than these differences, plus performing synthesis after each DFT insertion pass, you can follow the flow as described starting with “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan.](#)”


- **Prerequisites** — You must have the Tessent cell library or the ATPG library for the standard cells, in addition to the Tessent cell library for the I/O pad cells.
- **Design Loading** — During the first and second DFT insertion passes, ensure the following:
 - Specify the [set_context](#) command with the `-no_rtl` option rather than the `-rtl` option.
 - When setting the context, use the recommended naming conventions for the design IDs, which are “gate1” for the boundary scan/MemoryBIST insertion pass and “gate2” for the EDT/OCC insertion pass. If you are following this convention, you would then use design ID “gate3” for scan insertion.
 - Use the [read_cell_library](#) command to read in the library files for both the standard cells and pad I/O macros that are instantiated in the design.

Tessent Shell Flow for Hierarchical Designs

Tessent Shell uses the same “divide and conquer” methodology for hierarchical DFT by enabling you to perform RTL and scan DFT insertion at the sub-physical block level rather than only at the chip level. Implementing DFT hierarchically in a bottom-up process, starting with the lowest level blocks, helps divide the task and make it manageable.

Hierarchical design methodologies provide efficiencies for large designs. Rather than design at the chip level, chip designers break designs down into RTL blocks that enable design teams to work on different functional blocks in parallel. This method streamlines the process and enables RTL modifications, engineering change orders (ECOs), and other changes to be localized to particular blocks.

Tip

 Before performing DFT on a hierarchical design, familiarize yourself with “[DFT Architecture Guidelines for Hierarchical Designs](#)” on page 95.

This section builds on what you learned in “[Tessent Shell Flow for Flat Designs](#)” to describe the pre-layout RTL and scan DFT insertion process for hierarchical designs.

Refer to the following test case for a detailed usage example of the flow described in this section:

```
tessent_example_hierarchical_flow_<software_version>.tgz
```

You can access this test case by navigating to the following directory:

```
<software_release_tree>/share/UsageExamples/
```

Hierarchical DFT Terminology	142
How the DFT Insertion Flow Applies to Hierarchical Designs	144
RTL and Scan DFT Insertion Flow for Physical Blocks	146
RTL and Scan DFT Insertion Flow for the Top Chip	165
RTL and Scan DFT Insertion Flow for Sub-Blocks	180
RTL and Scan DFT Insertion Flow for Instrument Blocks	184
RTL and DFT Insertion Flow With Third-Party Scan	189

Hierarchical DFT Terminology

Tessent Shell uses a particular set of terms related to working with blocks in a hierarchical design. You must understand these terms to perform the RTL and scan DFT insertion process on hierarchical designs.

Refer to “[set_design_level](#)” in the *Tessent Shell Reference Manual* for more information.

- **Physical Block** — Physical blocks are logical entities that remain intact through tapeout. They are synthesis and layout regions. Below the top level of a chip, these are blocks that you can reuse, or instantiate, within a chip or across multiple chips. You perform synthesis on these blocks independent of the rest of the chip design.

When performing DFT insertion on physical blocks, Tessent preserves the ports at the root of the physical block. Instances of the physical block that exist below the current physical block may not be preserved in the final layout when ungrouping is used.

In Tessent Shell, the hierarchical DFT insertion flow distinguishes between three types of physical blocks: wrapped cores, unwrapped cores, and chip.

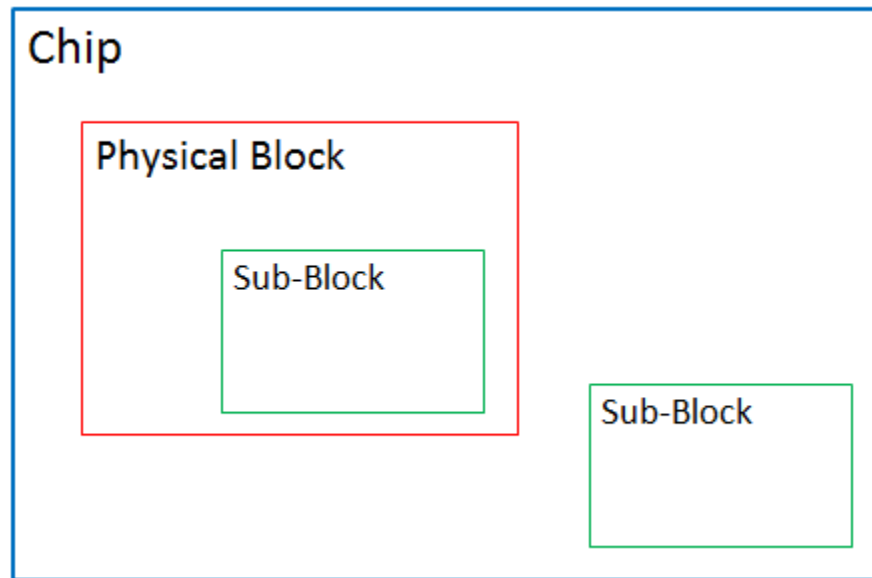
- **Wrapped core.** Wrapped cores contain wrapper cells that are used to isolate the internal logic of the core. Wrapper cells are inserted when you perform scan chain insertion. Wrapped cores are required to make the cores reusable through ATPG pattern retargeting. Wrapped cores can contain sub-blocks. (See the following description.)
- **Unwrapped core.** Unwrapped cores do not contain wrapper cells but can contain sub-blocks.

For additional information, refer to “[Unwrapped Cores Versus Wrapped Cores](#)” in the *Tessent Scan and ATPG User’s Manual*.

- **Chip.** The chip is the top-level physical block—that is, the entire design—in which you typically find the pad I/O macros and clock controllers. A chip may include another physical block or sub-block. Physical blocks can be wrapped cores or unwrapped cores. Unlike the other types of physical blocks, chips are layout regions.
- **Sub-Block** — Sub-blocks are designs that exist within parent blocks and are synthesized with their parent blocks, which could be wrapped cores, unwrapped cores, or the top level of the chip. Sub-blocks merge into their parent physical blocks during synthesis of the parent block. Refer to [set_design_level](#) for details.

Sub-blocks are not layout physical regions. After layout is performed on the post-layout netlist, the sub-block module boundary may or may not be preserved. Sometimes the same sub-block is instantiated at both the physical block level and the chip level, as shown in the following figure.

Figure 5-13. Hierarchical Design Levels



You can insert DFT hardware such as MemoryBIST, EDT, and OCC into sub-blocks, but you perform synthesis and scan insertion at the sub-block's parent physical block level (where the sub-block is instantiated). To learn about how to use sub-blocks within the Tessent Shell flow, refer to [“RTL and Scan DFT Insertion Flow for Sub-Blocks.”](#)

- **Instrument Block** — The design is a special empty module into which the DFT elements are inserted. The special module is then manually instantiated into its parent block and its pins are manually connected inside the parent block.

The synthesis, scan chain insertion, and pattern generation steps are done the usual way, as described in [“Instrument Block DFT Insertion Flow for the Next Parent Level”](#) on page 187.

- **ATPG (or scan) pattern retargeting** — The process by which Tessent Shell preserves the ATPG patterns associated with wrapped cores for purposes of reuse when testing the logic at the parent instantiation level. This means you do not have to regenerate the patterns when you process the top level of the chip. Instead, you retarget the wrapped core ATPG patterns to the top level. Every instantiation of a wrapped core includes its associated ATPG patterns.

For details, refer to [“Scan Pattern Retargeting”](#) in the *Tessent Scan and ATPG User's Manual*.

For purposes of ATPG pattern retargeting and graybox modeling (see the following description), Tessent Shell differentiates between a wrapped core's internal circuitry and its external circuitry.

- **Internal mode.** Internal mode is the view into the wrapped core from the wrapper cells. That is, the logic completely internal to the core. Tessent Shell retargets

internal mode ATPG patterns during ATPG pattern generation for the chip-level design.

- External mode. External mode indicates the view out of the wrapped core from the wrapper cells. That is, the logic that connects the wrapped core to external logic. Tessent Shell uses the external mode to build graybox models, which are used by the internal modes of their parent physical blocks.
- **Graybox** — Graybox models are wrapped core models that preserve only the core's external mode logic along with the portion of the IJTAG network needed for test setup of the logic test modes. The purpose of graybox models in the bottom-up hierarchical DFT process is to retain the minimum logic required to generate ATPG patterns for the internal modes of the parent physical blocks. For details, refer to “[Graybox Overview](#)” in the *Tessent Scan and ATPG User's Manual* and “[Graybox Model](#)” on page 100.

How the DFT Insertion Flow Applies to Hierarchical Designs

Performing RTL and scan DFT insertion on hierarchical designs assumes basic knowledge of the RTL and scan insertion flow for flat designs. The process for hierarchical designs builds on the process you would use for a flat design.

You perform a two-pass pre-scan insertion process for flat designs. For hierarchical designs, the same flow applies except that you perform the insertion process in a bottom-up manner for each core and sub-block in the design. After you have inserted DFT into all the lower-level physical blocks, you perform the same insertion into the parent blocks until you have reached the top level of the chip.

The following considerations apply when performing DFT insertion on a hierarchical design as opposed to a flat design:

- When performing hierarchical DFT, you must specify the hierarchical design level at which you are performing the DFT insertion process. For flat designs, the [set_design_level](#) command is always set to chip. For hierarchical designs, you can also specify `physical_block` or `sub_block`.
- Inserting boundary scan during the first DFT insertion pass typically applies to the chip design level. For hierarchical designs, this means that for cores and sub-blocks you insert only MemoryBIST during the first DFT insertion pass unless you have cores with embedded pad I/O macros. If you have cores with embedded pad I/O macros, then you need to insert boundary scan into the pad IOs using the [embedded boundary scan](#) flow as described in the *Tessent Boundary Scan User's Manual*.
- Within the hierarchical flow, each physical block and sub-block has a unique design name and should have its own TSDB.

Tip

i To facilitate data management, save each design (whether core, sub-block, or chip) in its own TSDB directory. This is the recommended practice when using Tessent Shell for DFT insertion. Using different directories ensures that you can run all sibling physical and sub-blocks in parallel without causing read-write errors into the TSDB directories between the parallel runs. Only when all the child physical and sub-blocks of a given block are completed can you then implement the DFT into the given block. Refer to “[TSDB Data Flow for the Tessent Shell Flow](#)” for more information.

- You can perform ATPG pattern retargeting of core-level patterns when you process the parent physical block of the wrapper cores, as shown in section “[Top-Level ATPG Pattern Generation Example](#).”

RTL and Scan DFT Insertion Flow for Physical Blocks

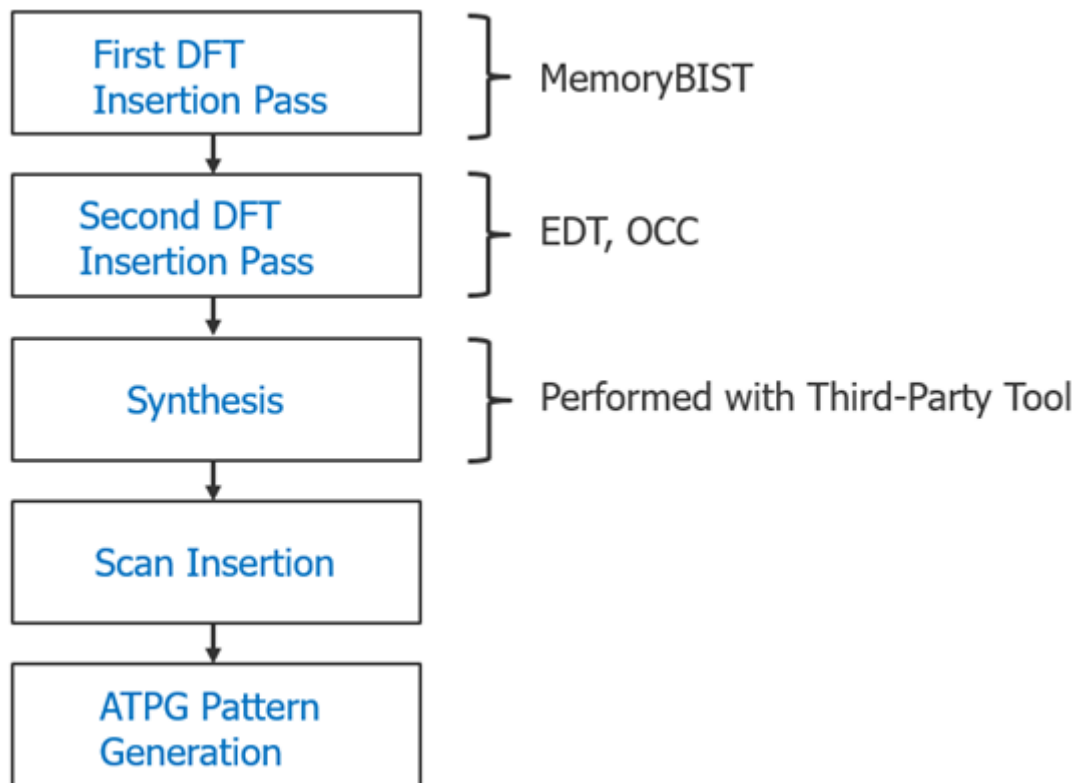
For each wrapped core, perform a two-pass pre-scan DFT insertion process as you would for a flat design except that in the first DFT insertion pass, do not insert boundary scan unless you have embedded pad IO macros present inside the core. The flow details for working with wrapped cores differ from those when working with flat designs.

Refer to “[Tessent Shell Flow for Flat Designs](#)” for overall flow details.

Note

This discussion assumes that your design consists of wrapped cores as your lower level physical blocks, and that the wrapped cores do not contain embedded pad IOs, so boundary scan is not required.

Figure 5-14. Two-Pass Insertion Flow, Physical Blocks



First DFT Insertion Pass: Performing Block-Level MemoryBIST.....	147
Second DFT Insertion Pass: Inserting Block-Level EDT and OCC.....	149
Specifying and Verifying the DFT Requirements: DFT Signals for Wrapped Cores..	152
Performing Scan Chain Insertion: Wrapped Core	154
Verifying the ICL Model.....	157


Performing ATPG Pattern Generation: Wrapped Core 158
Running Recommended Validation Step for Pre-Layout Design Sign Off 163

First DFT Insertion Pass: Performing Block-Level MemoryBIST

Because you are not inserting boundary scan at the same time as MemoryBIST (as in the flat flow), you can follow the DFT insertion process for Tessent MemoryBIST.

Refer to “[Getting Started](#)” in the *Tessent MemoryBIST User’s Manual*.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-2](#) on page 148.


Procedure

1. Load the RTL design data. (See lines 1-22.) The following steps are important for the first DFT insertion pass for wrapped cores:

- a. Set the `set_context -design_id` switch to “rtl1.”

All the data associated with MemoryBIST insertion for the wrapped core is stored under the ID “rtl1” in the wrapped core’s TSDB.

Note

 “rtl1” is the recommended naming convention for the design ID for the first insertion pass, but you can specify any name. Refer to “[Loading the Design](#)” for more information about setting the design ID.

- b. Specify the `read_verilog` command with a list of the RTL filenames and locations for Tessent to read and compile for the design. For example:

```
../rtl/omsp_timerA_defines.v  
../rtl/omsp_timerA_undefines.v  
../rtl/omsp_timerA.v  
../rtl/omsp_wakeup_cell.v  
../rtl/omsp_watchdog.v  
../rtl/openMSP430_defines.v  
../rtl/openMSP430_undefines.v  
../rtl/openMSP430.v  
../rtl/processor_core.v
```

- c. Set the design level to “physical_block” so that the layout of this core is maintained as an independent logical entity through tapeout.
2. Create the DFT specification. (See lines 24-30.)
 3. Generate the MemoryBIST hardware and extract the ICL. (See lines 32-36.)

4. Create the input test patterns and simulation test benches. (See lines 38-41.)
5. Run simulations to verify the design. (See lines 43-47.)

Examples

The following dofile example shows a typical command flow as detailed in the procedure listed in the preceding.

Example 5-2. Dofile Example for MemoryBIST in Physical Blocks

```
1 # Set context to dft and indicate DFT insertion into an rtl-level design
2
3 set_context dft -rtl -design_id rtl1
4
5 # Set the TSDB directory location to be used
6
7 set_tsdb_output_directory ../tsdb_core
8
9 # Read in the memory library model
10 set_design_sources -format tcd_memory -y ../../../../library/memory \
11     -extension tcd_memory
12
13 # Read in memory Verilog model
14 set_design_sources -format verilog -v {../../../../library/memory/*.v }
15
16 # Read in the design
17 read_verilog -f rtl_file_list
18
19 set_current_design processor_core
20
21 # Set the design level as a physical_block
22 set_design_level physical_block
23
24 # Specify to insert memory test
25 set_dft_specification_requirements -memory_test on
26 add_clocks 0 dco_clk -period 3
27 check_design_rules
28 report_memory_instances
29 set_spec [create_dft_specification]
30 report_config_data $spec
31
32 # Generate the memoryBIST hardware
33 process_dft_specification
34
35 # Create ICL for this design
36 extract_icl
37
38 # Generate testbenches
39 create_patterns_specification
40 process_patterns_specification
41 set_simulation_library_sources -v ../../../../library/verilog/adk.v
42
43 # Run simulations
44 run_testbench_simulations
45
46 # If simulation fails use the following command
```

```
47 //check_testbench_simulations
48 exit
```

Second DFT Insertion Pass: Inserting Block-Level EDT and OCC

In the second DFT insertion pass for wrapped cores, insert the EDT and OCC hardware. Unlike flat designs, which use standard OCCs, for wrapped cores you have a choice between inserting OCCs of type child or type standard.


Refer to “[On-Chip Clock Controller Design Description](#)” in the *Tessent Scan and ATPG User’s Manual* for details.

If your physical design implementation does not support using a clock MUX in the clock path, then you can use an OCC of type child. With the child type OCC, only clock chopping and clock gating functions occur inside the wrapped core, and these functions are sufficient for ATPG pattern retargeting (later in the flow) as long as at the chip-level you have included a parent OCC or other hardware that can perform clock selection.

Clock selection is used to select between shift and capture clocks when generating ATPG patterns for the internal mode of the core. Typically, scan chain shifting occurs at a significantly slower speed than the capture clock, hence the need for the clock.

Most of the steps for the second DFT insertion pass for wrapped cores are the same as those you would perform for a flat design. Refer to the applicable sections.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-3](#).

Procedure

1. Load the design (lines 1-10). See “[Loading the Design](#)” on page 121.
2. Specify and verify the DFT requirements (lines 12-32). See “[Specifying and Verifying the DFT Requirements](#)” on page 123

Note

 Wrapped cores have their own DFT requirements for the clock signals.

3. Create the DFT specification (lines 34-38). See “[Creating the DFT Specification](#)” on page 126.
4. Generate the EDT and OCC hardware (lines 40-78). See “[Generating the EDT, Hybrid TK/LBIST, and OCC Hardware](#)” on page 130.
5. Extract the ICL module description (lines 80-84). See “[Extracting the ICL Module Description](#)” on page 130.

6. Generate ICL patterns and run the simulation (lines 86-95). See “[Generating ICL Patterns and Running Simulation](#)” on page 131.

Examples

The following dofile example shows that you set the design ID to “rtl2” for the second DFT insertion pass, set the internal mode and external mode for the wrapped core, and have chosen to specify an OCC of type child.

Example 5-3. Dofile for Second DFT Pass

```
1 # Set context to dft and specify design_id as rtl2
2 set_context dft -rtl -design_id rtl2
3
4 # Specify where the tsdb_outdir is to be located, default is at the
5 # current working directory
6 set_tsdb_output_directory ../tsdb_core
7
8 # Use read_design with the design ID from the first DFT insertion pass
9 read_design processor_core -design_id rtl1
10 set_current_design processor_core
11
12 # No need to reset the design level, which remains "physical_block" from
13 # the first pass
14 # Add DFT signals for the wrapped core
15
16 # The following commands required for logic test
17 add_dft_signals ltest_en -create_with_tdr
18 add_dft_signals scan_en edt_update test_clock -source_node \
19 { scan_enable edt_update test_clock_u }
20 add_dft_signals edt_clock shift_capture_clock -create_from_other_signals
21
22 # Required for memories to be tested with multi-load ATPG patterns
23 add_dft_signals memory_bypass_en -create_with_tdr
24
25 # Required for STI network to be tested during logic test
26 add_dft_signals tck_occ_en -create_with_tdr
27
28 # Required for hierarchical DFT and used during scan insertion
29 # Specifying both the internal mode and the external mode
30 add_dft_signals int_ltest_en ext_ltest_en int_mode ext_mode
31
32 report_dft_signals
33
34 # Run pre-DFT DRC
35 set_dft_specification_requirements -logic_test on
36
37 check_design_rules
38 set_spec [create_dft_specification -sri_sib_list {edt occ} ]
39
40 # Use report_config_syntax DftSpecification/edt|occ to see full syntax
41 report_config_data $spec
42 read_config_data -in $spec -from_string {
43     occ {
```

```

44     ijtag_host_interface : Sib(occ);
45   }
46 }
47 # The following is a generic way to populate the OCC
48 # The clock list is design specific and needs to be updated for the design
49 # To the OCC - scan_enable and shift_capture_clock gets connected
50 # automatically
51 # Modify the following to your design requirements
52 set id_clk_list [list \
53   dco_clk dco_clk\
54 ]
55
56 foreach {id clk} $id_clk_list {
57   set occ [add_config_element OCC/Controller($id) -in $spec]
58   set_config_value clock_intercept_node -in $occ $clk
59 }
60 report_config_data $spec
61
62 ## To EDT controller, the edt_clock and edt_update get auto connected
63 ## The EDT controller is built-in with bypass
64 ## Modify the following to your design requirements

1 read_config_data -in $spec -from_string {
2   EDT {
3     ijtag_host_interface : Sib(edt);
4     Controller (c1) {
5       longest_chain_range : 200, 300;
6       scan_chain_count : 60;
7       input_channel_count : 3;
8       output_channel_count : 2;
9     }
10  }
11 }
12 report_config_data $spec
13 //display_spec
14 # Generate the hardware
15 process_dft_specification
16
17 # Extract the IJTAG network and create ICL file for core level

```

Comment:

```

18 extract_icl
19
20 # Write updated RTL into this new file to elaborate and synthesize later
21 write_design_import_script for_dc_synthesis.tcl -replace
22
23 # Generate testbench
24 create_patterns_specification
25 process_patterns_specification
26
27 set_simulation_library_sources -v ../../../../library/verilog/adk.v
28
29 # Run Verilog simulation
30 run_testbench_simulations
31 # If simulation fails, use the command below to see which pattern failed
32 //check_testbench_simulations

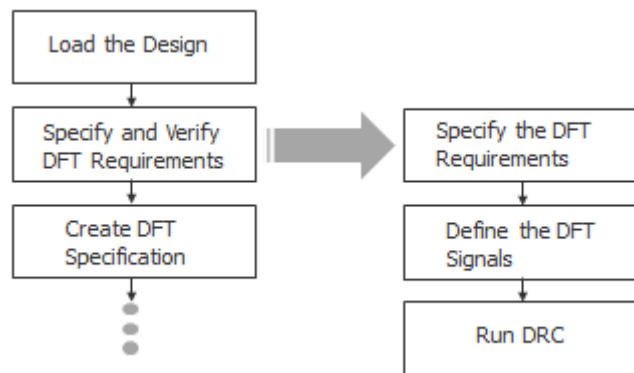
```

33
34 exit

Specifying and Verifying the DFT Requirements: DFT Signals for Wrapped Cores

After loading the design data from the TSDB, define the DFT requirements for the EDT and OCC hardware. This includes adding DFT signals and performing DRC.

Figure 5-15. Flow for Specifying and Verifying the DFT Requirements for Wrapped Cores



Procedure

1. Specify the following command to set DFT requirements:

```
set_dft_specification_requirements -logic_test on
```

Note

 The design level as specified by `set_design_level` remains “physical_block” from the first DFT insertion pass, so you do not need to specify this command again.

2. Use the following procedure to define the DFT signals for wrapped cores in the second DFT insertion pass:

- a. Specify a global DFT signal to enter logic test mode. For example:

```
add_dft_signals ltest_en -create_with_tdr
```

- b. Define the DFT signals. For example:

```
add_dft_signals scan_en edt_update test_clock -source_node \  
{ scan_enable edt_update test_clock_u }
```

```
add_dft_signals edt_clock shift_capture_clock \  
-create_from_other_signals
```

If these ports do not exist, the tool creates new ports.

- c. Specify the following command to test with multi-load ATPG patterns in MemoryBIST:

```
add_dft_signals memory_bypass_en -create_with_tdr
```

- d. Specify the following command to test an STI network during logic test.

```
add_dft_signals tck_occ_en -create_with_tdr
```

- e. Specify both the internal mode and the external mode for hierarchical DFT. This is required for scan insertion.

```
add_dft_signals int_ltest_en ext_ltest_en int_mode ext_mode
```

This command specifies that wrapped cores have both internal modes and external modes, and that you must specify both. Differentiating between internal mode and external mode enables Tessent to stitch the scan chains into internal chains and external chains as described in [“Performing Scan Chain Insertion: Wrapped Core.”](#)

The internal and external modes are also required for proper ATPG pattern retargeting and graybox modeling later in the insertion flow. Refer to [“Hierarchical DFT Terminology”](#) for more information.

3. After defining the DFT signals, run DRC as in the flat design flow.

If the design includes clock gating that is implemented in RTL and not with an integrated clock gating cell, you must specify their `func_en` and `test_en` ports using the [`add_dft_clock_enables`](#) command. Tessent checks for proper clock and asynchronous set and reset controllability.

Results

Tessent Shell generates DFT_C errors for DRCs that are run. For details, refer to “Pre-DFT Clock Rules (DFT_C Rules)” in the *Tessent Shell Reference Manual*.

Examples

To define the DFT signals, the following example shows the required DFT signals for wrapped cores in the second DFT insertion pass:

```
# Required global DFT signal to enter logic test mode
add_dft_signals ltest_en -create_with_tdr

# If these ports do not exist, the tool creates new ports
add_dft_signals scan_en edt_update test_clock -source_node \
{ scan_enable edt_update test_clock_u }
add_dft_signals edt_clock shift_capture_clock -create_from_other_signals

# Required for MemoryBIST to be tested with multi-load ATPG patterns
add_dft_signals memory_bypass_en -create_with_tdr

# Required for STI network to be tested during logic test
add_dft_signals tck_occ_en -create_with_tdr

# Required for hierarchical DFT and used during scan insertion
# Specifying both the internal mode and the external mode
add_dft_signals int_ltest_en ext_ltest_en int_mode ext_mode
```

Performing Scan Chain Insertion: Wrapped Core


Scan chain insertion for wrapped cores includes a task for performing wrapper analysis, which prepares the functional scannable sequential elements (or “flops”) for reuse as wrapper cells. Specifically, these cells are called shared wrapper cells. Using shared wrapper cells reduces the number of flops required for isolating the internal test modes from the primary input ports of the wrapper core. It also reduces the number of flops required for isolating the external test modes from the internal logic. Tessent automatically generates shared wrapper cells.

The [analyze_wrapper_cells](#) command performs wrapper analysis. In addition to preparing the shared wrapper cells, the command infers dedicated wrapper cells for ports having a large fanout or fanin. You can configure the size of the fanin and fanout using the `set_wrapper_analysis_options -input_fanout_flop_threshold` and `-output_fanin_flop_threshold` options. By default, Tessent infers dedicated wrapper cells on input ports fanning out to 32 or more scannable flip-flops, and on output ports in the fanout of 32 or more scannable flip-flops.

Both shared wrapper cells and dedicated wrapper cells can coexist in the same wrapper chains, which helps Tessent maintain wrapper chains of similar length. Scan insertion uses the DFT signals `int_ltest_en` and `ext_ltest_en` along with the scan enable signal to control the wrapper cell.

For information about scan chain insertion using Tessent Shell, refer to the “[Internal Scan and Test Circuitry Insertion](#)” in the *Tessent Scan and ATPG User’s Manual*.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-4](#) on page 156.

Prerequisites

- Prior to scan chain insertion, perform synthesis as described in section “[Performing Synthesis.](#)”

Procedure

1. Specify the following command to set the DFT context:

```
set_context dft -scan -design_id gate
```

2. Load the synthesized netlist. For example:

```
read_verilog ../3.synthesis/processor_core_synthesized.vg
```

3. Specify the same output directory you used in the first and second DFT passes:

```
set_tsdb_output_directory ../tsdb_core
```

4. Load the rtl2 design data for the DFT hardware you previously inserted. For example:

```
read_design processor_core -design_id rtl2 -no_hdl
```

5. Run design rule checking (DRC) using the following command:

```
check_design_rules
```

6. Set up the wrapper cells as follows (see lines 25-35):

- a. Specify the options for analyzing the wrapper cells.
- b. Add dedicated wrappers, as necessary.
- c. Analyze the wrapper cells with the [analyze_wrapper_cells](#) command.
- d. Ensure that you exclude the EDT channel IO ports from wrapper analysis.

For details, refer to “[Scan Insertion for Wrapped Core](#)” in the *Tessent Scan and ATPG User’s Manual*.

7. Specify the [add_scan_mode](#) command to connect the scan chains to the EDT signals and EDT hardware that you inserted during the second insertion pass.

Scan insertion for wrapper cells requires using [multi-mode scan insertion](#) as described in the *Tessent Scan and ATPG User’s Manual*. Do the following (see lines 41-46):

- a. Create one scan mode for the entire population of scan cells.

The entire population of scan cells are stitched into the first scan mode using the `int_mode` command to generate a scan mode consisting of all the scan cells stitched together.

- b. Create a second scan mode only for the shared and dedicated wrapper cells.

In the second DFT insertion pass, you had generated a DFT signal named “`int_mode`” with the `add_dft_signal` command. This signal enables this scan mode. You do not need to specify the `add_scan_mode -enable_dft_signal` switch when the mode name matches the name of a DFT signal of type scan mode.

The `add_scan_mode ext_mode` command stitches the shared and dedicated wrapper cells together, similar to the DFT signal you generated named `ext_mode`. `ext_mode` enables the scan mode for shared and dedicated wrapper cells.

Examples

The following dofile shows a command flow for scan insertion. The highlighted statements illustrate additional considerations for performing scan insertion for wrapper cores. For a general overview, refer to “[Performing Scan Chain Insertion \(Flat Design\)](#)” for a flat design.

Example 5-4. Scan Chain Insertion in Hierarchical Flow

```
1 set_context dft -scan -design_id gate
2
3 # You must respecify the TSDB
4 set_tsdb_output_directory ../tsdb_core
5
6 # Read Tessent library
7 read_cell_library ../../../../library/tessent/adk.tcelllib
8
9 # Read synthesized netlist
10 read_verilog ../3.synthesis/processor_core_synthesized.vg
11
12 # Use read_design to read in the DFT signals and other data from
13 # the second DFT insertion pass
14 read_design processor_core -design_id rtl2 -no_hdl
15 set_current_design processor_core
16
17 # If there are no ATPG models available for memories, use blackboxes
18 add_black_boxes -modules { \
19     SYNC_1RW_8Kx16 \
20 }
21
22 check_design_rules
23
24 report_clocks
25 # Exclude the EDT channel in and out ports from wrapper chain analysis
26 # The ijtag_* edt_update ports are automatically excluded
27 set_wrapper_analysis_options \
28     -exclude_ports [ get_ports {*_edt_channels_*} ]
29
30 # To force insertion of dedicated wrapper cell use the following command
31 # set_dedicated_wrapper_cell_options on -ports {.... }
```

```
32
33 # Perform wrapper cell analysis
34 analyze_wrapper_cells
35 report_wrapper_cells -Verbose
36
37 # Find edt_instance
38 set edt_instance [get_instances -of_icl_instances \
39 [get_icl_instances -filter tessent_instrument_type==mentor::edt]]
40
41 # Specify different modes (internal and external) of the chains that need
42 # to be stitched
43 # The type internal/external and enable_dft_signal are inferred from the
44 # registered DFT signals(int_mode and ext_mode)
45 add_scan_mode int_mode -edt_instances $edt_instance
46 add_scan_mode ext_mode -chain_count 2
47
48 # Before scan insertion you can analyze the different scan modes and scan
49 # chains
50 analyze_scan_chains
51 report_scan_chains
52 //delete_scan_modes -all
53 # Insert scan chains and write the scan inserted design into the TSDB
54 insert_test_logic
55 report_scan_chains
56 report_scan_cells > scan_cells.list
57 exit
```

Verifying the ICL Model

The ICL-based verification step verifies the ICL network before the pattern generation step.

During this verification, the tool checks the ICL network against semantic rules to verify the connectivity of the network. The tool verifies that it can access each IJTAG test data register through iWrite and iRead. The checks includes MemoryBIST logic if it is present in the design.

During scan chain stitching with Tessent Scan, the tool automatically updates the ICL model when complex ports generate loops, which causes ports and instances names to change through synthesis. When using a third-party scan insertion tool, additional ICL module matching rules may be required to complete the ICL-based verification step.

The following procedure and example dofile show how to verify the ICL model for SSN.

Procedure

1. Set the context to patterns to create ICL-based patterns.
2. Set the location of the *tsdb_outdir* directory and load the cell libraries.
3. Read the third-party scan-inserted netlist.
4. Load the collateral from the last DFT insertion step before scan insertion without reading the netlist.

5. Create and write the ICL-based pattern sets. This includes ICLNetwork verify patterns and MemoryBIST patterns, if memories are present.
6. Define the path to the Verilog simulation libraries, simulate the patterns, and check the simulation results.

Examples

This example dofile shows how to verify the ICL model for SSN.

```
# Set context to patterns
set_context patterns

# Open the previous TSDB directories if not in the current
# working directory
set_tsdb_output_directory ../tsdb_outdir

# Read Tessent Library
read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v -interface_only

# Read the design files from the scan insertion pass
read_design processor_core -design_identifier gate -verbose
set_current_design processor_core

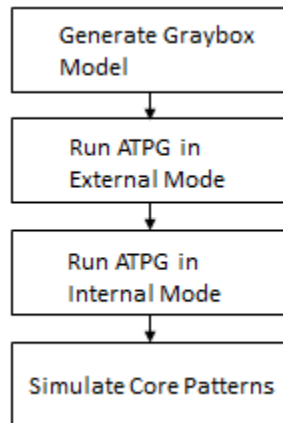
# Generate patterns to verify the inserted DFT logic
set_spec [create_patterns_specification]
report_config_data $spec
process_patterns_specification

# Point to the libraries and run the simulation
set_simulation_library_sources -v ../../../../library/standard_cells/ \
    verilog/adk.v -v ../../../../library/memories/*.v
run_testbench_simulations
check_testbench_simulations -report_status
exit
```

Performing ATPG Pattern Generation: Wrapped Core

Generating ATPG patterns for a wrapped core includes a step for generating a graybox model. In addition, you must perform ATPG twice, once for the core's external mode and once for the core's internal mode.

Refer to [“Performing ATPG Pattern Generation”](#) for flat designs for a general description.

Figure 5-16. ATPG Pattern Generation Flow for Wrapped Cores

As described in “[Hierarchical DFT Terminology](#),” the graybox model excludes the internal mode logic of the wrapped core, preserving only the external mode logic that needs to be tested at the parent level. The JTAG infrastructure is preserved in the graybox model also. Specifically, Tessent preserves the external logic that is present between the primary input and the input wrapper cells, plus any logic between the output wrapper cells and primary output. The parent could be another wrapper core or the top level of the chip.

You can use external mode patterns, if generated, for calculating fault coverage for the entire core (both internal and external mode). Use the internal mode ATPG patterns for ATPG pattern retargeting when performing the RTL and scan DFT insertion process on the top level of the chip.

Procedure

1. Generate graybox models (see [Example 5-5](#) on page 161 for a command flow example):
 - a. Load in the design using the same design ID as you used for scan insertion to write the graybox to the TSDB.

(Recommended) Use “gate” as the design ID if you inserted the MemoryBIST and EDT logic at the RTL level and “gate3” if the logic was also inserted into the gate level.
 - b. Specify the [analyze_graybox](#) command to generate graybox models.

When working at the parent level, using the same design ID for the graybox model as you used for scan insertion enables Tessent to access the full design view or the graybox model with the same design ID.
2. Run ATPG on the *external* mode of the wrapped core. This ATPG pattern is only used to calculate the entire core's fault coverage and cannot be reused from the chip-level. To generate ATPG patterns for external mode, do the following:
 - a. Read in the graybox model of the design with the [read_design](#) command.

Use the [set_current_mode](#) command to specify a unique ATPG mode name that represents the purpose of the pattern. The mode type is external.

- b. Use the [import_scan_mode](#) command to retrieve the core's external mode data. Tessent uses the graybox model of the core. Using the `import_scan_mode` command assumes that you used Tessent Scan to perform scan chain stitching.
- c. Run design rule checking (DRC) using the following command:

```
check_design_rules
```

- d. Generate the ATPG patterns using the following command:

```
create_patterns
```

- e. Use the [write_tsdb_data](#) command to store the TCD, flat model, fault list and PatDB files in the TSDB.
- f. Use the [write_patterns](#) command to write out the test bench required to simulate the generated ATPG patterns.

See [Example 5-6](#) on page 161.

3. Run ATPG on the *internal* mode of the wrapped core. This results in the ATPG pattern that you retarget at the top level of the chip. To generate ATPG patterns for internal mode, do the following:

- a. Load in the graybox views for the wrapper cores that contain child wrapper cores.
- b. If you used Tessent Scan for scan insertion, specify `import_scan_mode` to import the internal mode.
- c. Specify a unique ATPG mode name with the `set_current_mode` command.

The current mode type is internal. Typical mode names are *scan_mode_name_sa* or *scan_mode_name_tdf*.

- d. Run design rule checking (DRC) using the following command:

```
check_design_rules
```

- e. Generate the ATPG patterns using the following command:

```
create_patterns
```

- f. Store the TCD, flat model, fault list and PatDB files in the TSDB using the `write_tsdb_data` command.
- g. Use the [read_faults](#) command to merge the fault list from running external mode to find the total overall fault coverage of the wrapped core.

See [Example 5-7](#) on page 162.

4. Run Verilog simulation of the core-level ATPG patterns.

Performing this task ensures that the patterns function as needed when they are retargeted at the parent level. For parallel load patterns as specified by the `-parallel` command, simulate all the patterns. For serial load patterns, a handful of patterns are sufficient; the run time for simulating gate-level serial load patterns is significant.

Examples

Generate the Graybox Model

The following example creates a graybox model for a scan-inserted `processor_core` design and saves the data under the “gate” design ID.

Example 5-5. Graybox Example

```
set_context patterns -scan -design_id gate
set_tsdb_output_directory ../tsdb_core
read_cell_library ../../../../library/tessent/adk.tcelllib

## Reads in the scan inserted netlist/design
read_design processor_core -design_id gate -verbose
set_current_design processor_core
add_black_boxes -modules { \
                    SYNC_1RW_8Kx16 \
                    }

report_dft_signals

import_scan_mode ext_mode
check_design_rules
analyze_graybox
write_design -tsdb -graybox -verbose

exit
```

Run ATPG on the Core's External Mode

The following example shows the command flow for running ATPG on the external mode of a core.

Example 5-6. ATPG External Mode

```
set_context patterns -scan
set_tsdb_output_directory ../tsdb_core
read_cell_library ../../../../library/tessent/adk.tcelllib

# Read in the graybox model using the -view switch
read_design processor_core -design_id gate -view graybox -verbose
set_current_design processor_core

# Specify a different name that what was used during scan insertion with
# the add_scan_mode command
set_current_mode edt_multi_SAF -type external
report_dft_signals
```

```
# Extract the external mode specified during scan insertion
import_scan_mode ext_mode
report_core_instances
report_static_dft_signal_settings

# Run DRC, Tessent Shell prompts changes to Analysis
check_design_rules
report_clocks
set_xclock_handling x

# Generate ATPG patterns
create_patterns

# Store TCD, flat_model, fault list and PatDB files in the TSDB
write_tsdb_data -replace
write_patterns patterns/processor_core_ext_stuck_parallel.v -verilog \
  -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
set_pattern_filtering -sample_per_type 2
write_patterns patterns/processor_core_ext_stuck_serial.v \
  -verilog -serial -replace -parameter_list {SIM_KEEP_PATH 1}
exit
```

Run ATPG on the Core's Internal Mode

The following example shows the command flow for running ATPG on the internal mode of a core.

Example 5-7. ATPG Internal Mode

```
set_context patterns -scan
set_tsdb_output_directory ../tsdb_core
read_cell_library ../../../../library/tessent/adk.tcelllib
# Read in the full scan-inserted netlist
read_design_processor_core -design_id gate
set_current_design processor_core

# Extract the internal mode specified during scan insertion
import_scan_mode int_mode

# Use add_scan_mode to specify a different name than what was used during
# scan insertion
# Specify import_scan_mode before set_current_mode because
# import_scan_mode overrides the test mode type specified by
# set_current_mode
set_current_mode int_mode_sa -type internal

report_dft_signals
report_core_instances
report_static_dft_signal_settings

# Run DRC
check_design_rules
report_clocks
report_core_instances
add_fault -all
report_statistics -detail
```

```
# Generate ATPG patterns
create_patterns
report_statistics -detail

# Store TCD, flat_model, fault list and PatDB files in the TSDB
write_tsdb_data -replace
write_patterns patterns/processor_core_stuck_parallel.v -verilog \
  -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
set_pattern_filtering -sample_per_type 2
write_patterns patterns/processor_core_stuck_serial.v \
  -verilog -serial -replace -parameter_list {SIM_KEEP_PATH 1}
exit

# To view the coverage of the faults testable by internal mode, you must
# eliminate the undetected faults, which would be detected in
# external mode. Do this by merging in the fault list generated from
# performing ATPG on the graybox in external mode
read_faults -mode ext_multi_SAF -fault_type stuck -merge

# Final coverage of the core that includes internal and external modes
report_statistics -detail
exit
```

Running Recommended Validation Step for Pre-Layout Design Sign Off

During the first pass of the hierarchical DFT insertion flow for a wrapped core, you insert and simulate the MemoryBIST hardware, mostly at the RTL level. To ensure that the MemoryBIST simulation passes before the core is signed off as the pre-layout netlist, rerun MemoryBIST simulation at gate level on the core's scan-inserted netlist.

Procedure

1. Read in the scan-inserted netlist from the TSDB

Using the recommended naming conventions for the RTL and scan DFT insertion flow, the design ID would be “gate” if you inserted the MemoryBIST and EDT logic at the RTL level and “gate3” if the logic was also inserted into the gate level.

2. Elaborate the design and run DRC.
3. Create the input test patterns and simulation files.
4. Simulate the test bench.

Tessent Shell stores the generated test bench in the TSDB under the Patterns directory using the design ID “gate”.

Examples

The following example shows how to validate the MemoryBIST patterns for a scan-inserted netlist.

```
set_context_patterns -ijtag -design_id gate
set_tsdb_output_directory ../tsdb_core

##Reading the tessent cell library
read_cell_library ../../../../library/tessent/adk.tcelllib

## Reading the scan inserted netlist
read_design_processor_core -design_id gate -verbose -view interface
set_current_design processor_core
add_black_boxes -modules { \
                    SYNC_1RW_8Kx16 \
                    }
check_design_rules

create_patterns_specification
process_patterns_specification
set_simulation_library_sources \
    -v ../../../../library/memory/SYNC_1RW_8Kx16.v \
    -v ../../../../library/verilog/adk.v
run_testbench_simulation
exit
```

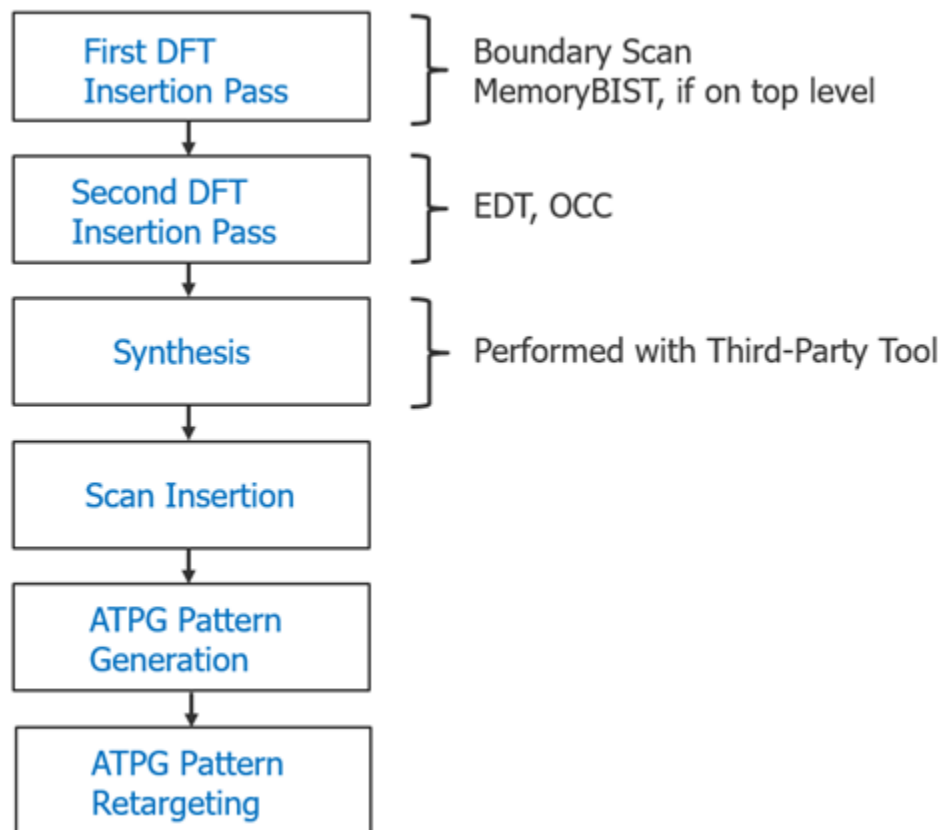
RTL and Scan DFT Insertion Flow for the Top Chip

After performing the RTL and scan DFT insertion flow for each wrapped core in your design, you can perform the DFT insertion process for the top-level chip design.

Before implementing DFT at the top level of the chip, plan how you should test the wrapped cores at the chip level. The planning for logic testing the wrapped cores is not automated, so you must decide how you to allocate the resources and organize the test schedule (especially for ATPG pattern retargeting) and specify your intent by using the [add_dft_modal_connections](#) command.

The RTL and scan DFT insertion flow for the top level of a chip follows the same basic process you used for the cores, with the addition of a step for retargeting the ATPG patterns you generated for the wrapped cores.

Figure 5-17. Two-Pass Insertion Flow for RTL, Top Level



In addition, processing at the chip level differs from wrapped cores in that you must insert a Test Access Mechanism (TAM). A TAM is a mechanism that you use to carry the scan data in and out of the chip for each group of wrapped cores you intend to run in parallel. The TAM schedules the tests for the wrapped cores at the top level, and enables access to the chip level so that you can run these tests.

During insertion and pattern generation, you open the TSDBs that store the wrapped core design data and read in the designs for their graybox models. The DFT insertion flow for the top level of the chip requires differentiating between three design views of the wrapped cores.

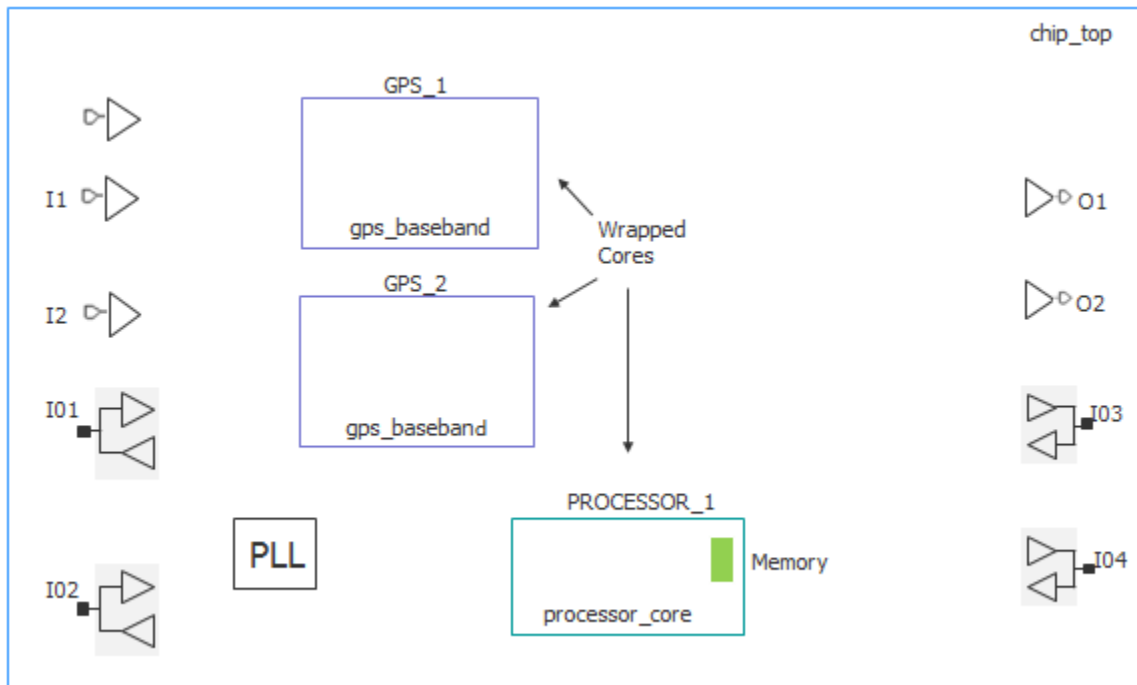
- **Full netlist view** — All the logic for the core. This is the default view when you do not explicitly specify a design view with the [read_design](#) command.
- **Graybox model** — External mode logic for the core as described in “[Hierarchical DFT Terminology](#),” including its IJTAG interface. You use this view so that Tessent Shell can connect the wrapped cores at the top level for logic testing of the chip.
- **Interface view** — The core’s ports only. Tessent auto-loads the interface view of any sub-physical block for which you have not used [read_design](#) to load its view.

Top-Level DFT Insertion Example	166
First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan.	168
Second DFT Insertion Pass: Inserting Top-Level EDT and OCC	171
Top-Level Scan Chain Insertion Example.	175
Top-Level ATPG Pattern Generation Example	175
Performing Top-Level ATPG Pattern Retargeting	177

Top-Level DFT Insertion Example

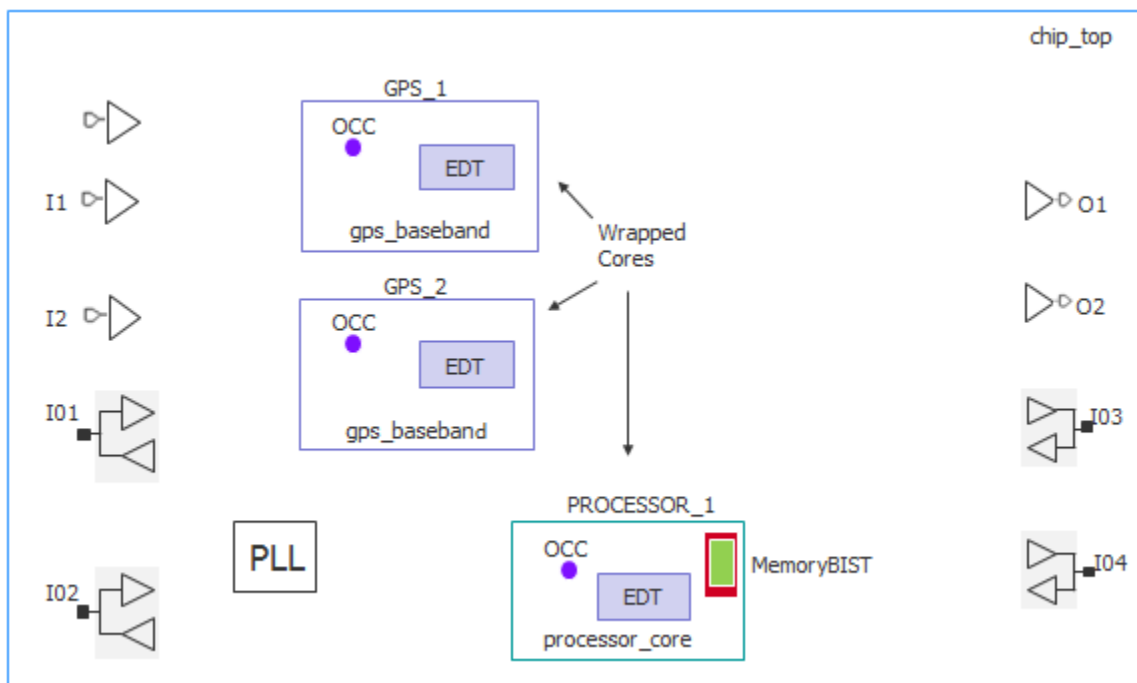
The RTL and scan DFT insertion flow for the top level of a chip can be illustrated using two wrapped cores, `processor_core` and `gps_baseband`. The processor core has memory instantiated within it, and the GPS baseband is a logic-only core that is instantiated twice.

Figure 5-18. Top-Level Example, Before DFT Insertion



After performing the RTL and scan DFT insertion process for the wrapped cores, each of the cores has an EDT controller and a child OCC. The processor core has the memories with MemoryBIST already inserted.

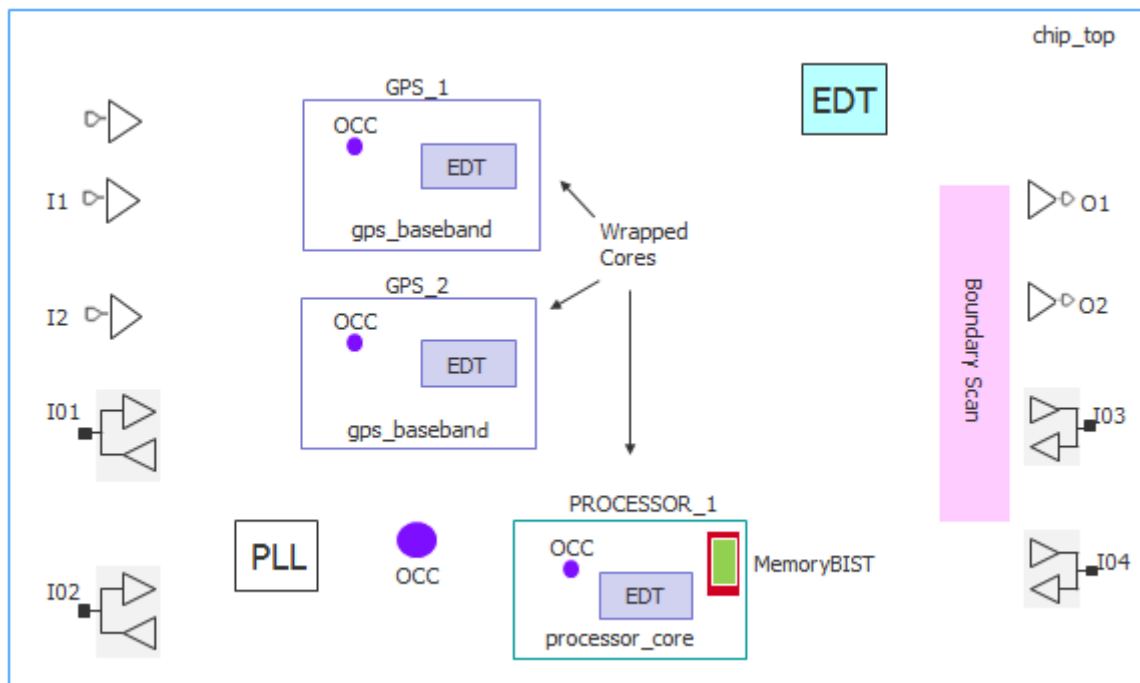
Figure 5-19. Top-Level Example, After DFT Insertion for Wrapped Cores



After inserting DFT at the top level, the design includes a TAP controller along with boundary scan, a top-level EDT controller, a parent OCC, and a TAM for purposes of retargeting the wrapped cores (not shown in Figure 5-20 on page 168).

In this top-level example test case, the hierarchical core starts with RTL insertion, and in addition, at the top level you want to perform DFT at RTL as well. However, there is no RTL logic at the top level that needs to be synthesized. The PLL and pad IOs are Verilog macros. The only RTL that needs to be synthesized is the Tessent-generated RTL. Hence, this test case uses design IDs “gate1” and “gate2” for the first two DFT insertion passes, respectively.

Figure 5-20. Top-Level Example, After DFT Insertion at the Top Level



First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan

For the top level of a chip, insert boundary scan plus MemoryBIST for any memories that are present at the top level. In addition, build the IJTAG network for the wrapped cores at the top level and insert a TAP controller or reuse an existing TAP controller.

As with flat designs, the flow for inserting MemoryBIST and boundary scan together is the same as inserting them separately. For details about this insertion pass flow, refer to:

- “Getting Started” in the *Tessent MemoryBIST User’s Manual*, or
- “Getting Started With Tessent BoundaryScan” in the *Tessent BoundaryScan User’s Manual*

For information about TAP controller reuse, refer to “[create_dft_specification](#)” in the *Tessent Shell Reference Manual*.


Prerequisites

- Refer to “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” (for flat designs) for general information and prerequisites. For example, as with flat designs, you can segment the boundary scan chain into smaller chains that are used during logic testing with Tessent TestKompress by using the [max_segment_length_for_logictest](#) command.

Procedure

1. Load the design.

Note

-  Use the [open_tsdb](#) command to open the TSDBs for all the lower-level cores. Opening their TSDBs makes their design data available.

There is no need to specify the `read_design` command because, by default, Tessent reads in the interface views of the wrapped cores, which is all that is required for DRC for the first DFT insertion pass.

2. Set the design level to “chip” for the top level of the chip.
When working with the wrapped cores, you had set the design level to “physical_block.”
3. Create the DFT specification.
4. Specify enough auxiliary input and output ports for the largest retargeting wrapper core group.
5. Generate the hardware and extract the ICL.
6. Create the input test patterns and simulation test benches.
7. Run simulations to verify the design.

Examples

The following dofile example shows a typical command flow for inserting MemoryBIST and boundary scan. The flow is the same as you would use for a flat design with the exception of the commands highlighted in bold. The chip-level design data exists in its own TSDB. This is recommended for the data flow as described in “[TSDB Data Flow for the Tessent Shell Flow](#).”

```
set_context dft -no_rtl -design_id gate1
set_tsdb_output_directory ../tsdb_outdir

# Open the TSDB of all the wrapped cores
open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir

# Read the Tessent cell library
read_cell_library ../../library/tessent/adk.tcelllib

# Read in PLL as a blackbox so that it is excluded in design list that you
write
# out for synthesis
read_verilog ../rtl/noncore_blocks/pll.v -blackbox

# Point to Verilog libraries for the pad IO macros
set_design_sources -format verilog \
    -v ../rtl/noncore_blocks/pad8_io_macro.v \
    -v ../rtl/noncore_blocks/iopad_sel.v \
    ../rtl/noncore_blocks/iopad.v
read_verilog ../rtl/chip_top.v
set_current_design chip_top
set_design_level chip

# Specify and verify the DFT requirements
set_dft_specification_requirements -boundary_scan on -memory_bist on

# Identify the TAP pins
set_attribute_value TCK -name function -value tck
set_attribute_value TDI -name function -value tdi
set_attribute_value TMS -name function -value tms
set_attribute_value TRST -name function -value trst
set_attribute_value TDO -name function -value tdo

# Some pins cannot add any boundary scan cells
set_boundary_scan_port_options TEST_CLOCK_top -cell_options dont_touch
set_boundary_scan_port_options EDT_UPDATE_top -cell_options dont_touch
set_boundary_scan_port_options SCAN_ENABLE -cell_options dont_touch
set_boundary_scan_port_options RESET_N -cell_options dont_touch
set_boundary_scan_port_options INCLK -cell_options dont_touch

# Add DFT signals
add_dft_signals scan_en -source_nodes SCAN_ENABLE
report_dft_signals
add_clocks PLL_1/pll_clock_0 -reference REF_CLK -FREQ_Multiplier 16
add_clock REF_CLK -period 48
check_design_rules

# Create a dft specification
set_spec [create_dft_specification]
report_config_data $spec
# Segment the boundary scan to be used during logic test
set_config_value $spec/BoundaryScan/max_segment_length_for_logic_test 80

# Equip these ports with auxiliary input/output muxes to be used by EDT
# channel pins
read_config_data -in ${spec}/BoundaryScan -from_string {
    AuxiliaryInputOutputPorts {
```

```
    auxiliary_input_ports : GPIO1_0, GPIO1_1, GPIO1_2, GPIO1_3;
    auxiliary_output_ports : GPIO2_0, GPIO2_1, GPIO2_2, GPIO2_3, GPIO1_0,
GPIO1_1 ;
}
}


report_config_data $spec
process_dft_specification
extract_icl
run_synthesis
create_patterns_specification
process_patterns_specification
set_simulation_library_sources \
-v ../../library/verilog/*.v \
-v ../rtl/noncore_blocks/pll.v \
-v ../rtl/noncore_blocks/iopad.v \
-v ../../library/memory/SYNC_1RW_8Kx16.v

run_testbench_simulations
exit
```

Second DFT Insertion Pass: Inserting Top-Level EDT and OCC

For the top level of a chip, additional considerations for the second DFT insertion pass for EDT and OCC include grayboxes, DFT signals for purposes of ATPG retargeting, and TAMs. EDT, or Embedded Deterministic Test, reduces scan data volume and test time.

Note

 This procedure follows a similar flow as flat designs as documented in “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” on page 116.

Procedure

1. Specify the [open_tsdb](#) command to open the TSDBs for the wrapped cores, as in the first DFT insertion pass for the chip.
2. Specify the [read_design](#) command to read in the graybox models of the wrapped cores.

This enables Tessent to perform DRC on the wrapper chains in addition to the rest of the top-level logic to be tested.

3. Define a retargeting mode for each group of wrapped cores whose ATPG patterns you want to retarget to run in parallel.

For ATPG pattern retargeting purposes, Tessent requires you to include retargeting mode DFT signals in addition to the DFT signals defined in section “[DFT Signals](#)”. The `retargeting<X>_mode` signals along with the TAM specified with the `add_dft_modal_connections` command ensure that ATPG pattern retargeting occurs correctly. Optionally, you can register your own DFT signals to be used for retargeting purposes.

[Example 5-8](#) on page 172 demonstrates retargeting of two wrapped cores: processor_core and gps_baseband.

4. Apply the `add_dft_modal_connections` command to specify the TAM.

During the first insertion pass, you identified the functional pins to be shared with EDT channel pins and added the required auxiliary input and auxiliary output logic. Now you use the TAM to sensitize paths to and from the top-level EDT using the `set_config_value` command.

In [Example 5-8](#) on page 172, the functional input pin GPI01_0 is shared as an EDT channel input pin. Similarly, the functional output pin GPI02_0 is shared as an EDT channel output pin.

5. Combine with the top-level EDT mode signal you defined by connecting the EDT channel IOs of the wrapped cores to top-level pins via the TAM. Use the `add_dft_modal_connections` command.
6. Apply the `set_defaults_value` command to specify that Tessent Shell simulate the instruments within the wrapped cores in addition to the top-level instruments.

Examples

The following dofile example shows that the DFT insertion flow for inserting EDT and OCC into the top level of a chip follows the same basic process as for flat designs as described in “[Second DFT Insertion Pass: EDT, Hybrid TK/LBIST, and OCC](#),” with the exception of the highlighted commands.

Example 5-8. Top-Level Second DFT Pass

```
set_context dft -no_rtl -design_id gate2
set_tsdb_output_directory ../tsdb_outdir
open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir

read_cell_library ../../library/tessent/adk.tcelllib
read_verilog ../rtl/noncore_blocks/p11.v -blackbox
set_design_sources -format verilog -v ../rtl/noncore_blocks/iopad_sel.v \
-v ../rtl/noncore_blocks/iopad.v
read_design chip_top -design_id rtl1 -verbose
read_design processor_core -design_id gate -view graybox -verbose
read_design gps_baseband -design_id gate -view graybox -verbose
set_current_design chip_top
```

```

# Add DFT Signals
# When using boundary scan in logic test without contacting inputs
add_dft_signals int_ltest_en output_pad_disable -create_with_tdr

# Needed for Scan Tested Instruments such as MemoryBIST and boundary scan
add_dft_signals tck_occ_en -create_with_tdr

# When you are using logic test
add_dft_signals ltest_en -create_with_tdr

# Used by top-level EDT
add_dft_signals edt_mode -create_with_tdr

# These are used for the top-level EDT
add_dft_signals test_clock edt_update \
    -source_nodes {TEST_CLOCK_top EDT_UPDATE_top}
add_dft_signals shift_capture_clock edt_clock -create_from_other_signals

# Retargeting mode signals used for ATPG pattern retargeting
add_dft_signals retargeting1_mode retargeting2_mode retargeting3_mode \
retargeting4_mode

# Add TAM connections
# For chip-level EDT mode, the asterisk(*) means do not make connection to
the data inputs
add_dft_modal_connections -ports GPIO1_0 -input_data_destination_nodes * \
    -enable_dft_signal edt_mode
add_dft_modal_connections -ports GPIO2_0 -output_data_source_nodes * \
    -enable_dft_signal edt_mode

# Connect wrapped core EDT channel I/Os to top level for retargeting1_mode
signal
add_dft_modal_connections -ports GPIO1_2 -input_data_destination_nodes
PROCESSOR_1/processor_core_rtl2_controller_c1_edt_channels_in[0] \
    -enable_dft_signal retargeting1_mode
add_dft_modal_connections -ports GPIO2_2 -output_data_source_nodes
PROCESSOR_1/processor_core_rtl2_controller_c1_edt_channels_out[0] \
    -enable_dft_signal retargeting1_mode

# Connect wrapped core EDT channel I/Os to top level for retargeting2_mode
signal
add_dft_modal_connections -ports GPIO1_2 -input_data_destination_nodes
GPS_1/gps_baseband_rtl1_controller_c1_edt_channels_in[0] \
    -enable_dft_signal retargeting2_mode
add_dft_modal_connections -ports GPIO1_2 -input_data_destination_nodes
GPS_2/gps_baseband_rtl1_controller_c1_edt_channels_in[0] \
    -enable_dft_signal retargeting2_mode
...
add_dft_modal_connections -ports GPIO1_0 -output_data_source_nodes GPS_1/
gps_baseband_rtl1_controller_c1_edt_channels_out[0] \
    -enable_dft_signal retargeting2_mode -pipeline_stages 1
add_dft_modal_connections -ports GPIO1_1 -output_data_source_nodes GPS_1/
gps_baseband_rtl1_controller_c1_edt_channels_out[1] \
    -enable_dft_signal retargeting2_mode -pipeline_stages 1
...

```

```
report_dft_modal_connections
set_dft_specification_requirements -logic_test On
add_clocks INCLK -period 10ns
check_design_rules
report_dft_control_points

# Create DFT specification
set_spec [create_dft_specification -sri_sib_list {occ edt} ]
report_config_data $spec
read_config_data -in $spec -from_string {
  OCC {
    ijtag_host_interface : Sib(occ);
  }
}
set_id_clk_list [list \
  pll_clock_0 PLL_1/pll_clock_0 \
  INCLK INCLK \
]
foreach {id clk} $id_clk_list {
  set_occ [add_config_element OCC/Controller($id) -in $spec]
  set_config_value clock_intercept_node -in $occ $clk
}
report_config_data $spec
read_config_data -in $spec -from_string {
  EDT {
    ijtag_host_interface : Sib(edt);
    ...
  }
}

set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsIn(1) \
  [get_single_name [get_auxiliary_pins GPIO1_0 -direction input]]
set_config_value port_pin_name \
  -in $spec/EDT/Controller(c1)/Connections/EdtChannelsOut(1)
  [get_single_name [get_auxiliary_pins GPIO2_0 -direction output] ]
report_config_data $spec
process_dft_specification
extract_icl


# By setting this value, all the lower level instruments in the wrapped
# cores are simulated
set_defaults_value /PatternsSpecification/SignoffOptions/
simulate_instruments_in_lower_physical_instances on
create_patterns_specification
process_patterns_specification
set_simulation_library_sources -v ../../library/verilog/*.v \
  ...

run_testbench_simulations
exit
```

Top-Level Scan Chain Insertion Example

For the top level of the chip, additional considerations for scan chain insertion include opening the TSDBs for the wrapped cores and reading in the wrapped core graybox models as you did for the second DFT insertion pass.

Note

 Prior to scan chain insertion, perform synthesis as described in section “[Performing Synthesis](#).”

Logic that is present at the top-level needs to be scan stitched along with the wrapper chains (external mode) of the cores.

Refer to “[Performing Scan Chain Insertion \(Flat Design\)](#)” (for flat designs) for more information about scan chain insertion. The following dofile example shows that Tessent Shell needs to access the graybox models for each wrapped core.

Example 5-9. Top-Level Scan Chain Insertion

```
set_context dft -scan -design_id gate
set_tsdb_output_directory ../tsdb_outdir
open_tsdb ../../wrapped_cores/processor_core/tsdb_core
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_core

read_cell_library ../../library/tessent/adk.tcelllib
read_verilog ../rtl/noncore_blocks/p11.v -blackbox
set_design_sources -format verilog -v ../rtl/noncore_blocks/iopad_sel.v \
    -v ../rtl/noncore_blocks/iopad.v

read_verilog ../Synthesis/chip_top_synthesized.vg
read_design chip_top -design_id rtl2 -no_hdl -verbose
read_design processor_core -design_id gate -view graybox -verbose
read_design gps_baseband -design_id gate -view graybox -verbose
set_current_design chip_top

check_design_rules
report_clocks

set edt_instance [get_name_list [get_instance -of_module \
    [get_name [get_icl_module -of_instances chip_top* \
    -filter tessent_instrument_type==mentor::edt]]] ]
add_scan_mode edt_mode -edt_instance $edt_instance
analyze_scan_chains
report_scan_chains
insert_test_logic
exit
```

Top-Level ATPG Pattern Generation Example

At the top level, Tessent Shell uses the scan-inserted design data for the chip. In the recommended flow this equates to design ID “gate.” In addition, Tessent uses the graybox

models for the wrapped cores so that you can use the external mode of the wrapper chains to run ATPG.

Refer to “[Performing ATPG Pattern Generation](#)” (for flat designs) for information about generating the ATPG patterns. The flow for the top level of a chip builds onto the process by adding in the grayboxes and blackboxes. As with flat designs, you have a choice between using the boundary scan chains for capture or using the primary pins of the chips (using the pads).

The following dofile example generates ATPG patterns at the top level using the stuck-at fault model. The dofile shows that you can read in the stuck-at fault models for the wrapped cores to calculate the total fault coverage for the chip.

Example 5-10. Top-Level ATPG Pattern Generation

```
set_context pattern -scan
set_tsdb_output_directory ../tsdb_top
open_tsdb ../../wrapped_cores/processor_core/tsdb_core
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_core
read_cell_library ../../library/tessent/adk.tcelllib

# Read the Verilog
read_verilog ../rtl/noncore_blocks/pll.v -blackbox
set_design_sources -format verilog -v ../rtl/noncore_blocks/iopad.v \
    -v ../rtl/noncore_blocks/iopad_sel.v

read_design chip_top -design_id gate
read_design processor_core -design_id gate -view graybox -verbose
read_design gps_baseband -design_id gate -view graybox -verbose
set_current_design chip_top
set_current_mode edt_top_stuck

import_scan_mode edt_mode
report_core_instances

set_static_dft_signal_values tck_occ_en 1
report_static_dft_signal_settings

set_system_mode analysis
add_fault -all
report_statistics -detail
create_patterns
report_statistics -detail
write_tsdb_data -replace

write_patterns patterns/chip_top_edt_parallel.v -verilog -parallel \
    -replace -scan -parameter_list {SIM_KEEP_PATH 1}
set_pattern_filtering -sample_per_type 2
write_patterns patterns/chip_top_edt_serial.v -verilog -serial -replace \
    -parameter_list {SIM_KEEP_PATH 1}
write_patterns patterns/chip_top_edt_stuck.stil -stil -replace
```



```
# Total fault coverage for stuck-at patterns at the chip-level including
# the cores
read_faults -module processor_core -mode edt_int_stuck \
  -fault_type stuck -merge -graybox
read_faults -module gps_baseband -module edt_int_stuck \
  -fault_type stuck -merge -graybox
report_statistics -detail
exit
```

Performing Top-Level ATPG Pattern Retargeting

For each wrapped core, retarget the internal ATPG patterns for all the modes you have specified and all the fault types.

Note

 This procedure is similar to “[Performing ATPG Pattern Generation: Wrapped Core](#)” on page 158.

Procedure

1. Specify the `set_context` patterns -retargeting command to retarget the ATPG patterns generated for the wrapped cores.
2. Use the same TSDB for ATPG retargeting as you used for ATPG pattern generation.
3. Set the current mode to a unique mode name that, ideally, indicates the core name, the fault type, and the retargeting mode DFT signal you had previously defined.
4. Specify which wrapped core ATPG patterns you are retargeting by enabling the correct retargeting mode DFT signal. Set the `set_static_dft_signal_values` command to the retargeting mode for this wrapped core.
5. Apply the `add_core_instances` command to specify the wrapped core whose internal ATPG patterns you want to retarget.
6. Apply the `read_patterns` command to read in the stuck-at ATPG patterns that you want to retarget.

Examples

The following dofile example shows how you would retarget the stuck-at ATPG patterns for the wrapped core, `processor_core`.

Example 5-11. Retarget Stuck-At ATPG Patterns for the Top-Level

```
set_context pattern -scan_retargeting
set_tsdb_output_directory ../tsdb_top
open_tsdb ../../wrapped_cores/processor_core/tsdb_core
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_core
```

```
read_cell_library ../../library/tessent/adk.tcelllib
read_verilog ../rtl/noncore_blocks/pll.v -blackbox
set_design_sources -format verilog -v ../rtl/noncore_blocks/iopad.v \
  -v ../rtl/noncore_blocks/iopad_sel.v
read_design chip_top -design_id gate
read_design processor_core -design_id gate -view graybox -verbose
read_design gps_baseband -design_id gate -view graybox -verbose

set_current_design chip_top

# Retarget processor_core stuck-at patterns
set_current_mode retarget1_processor_stuck
set_static_dft_signal_values retargeting1_mode 1
add_core_instances -instances {PROCESSOR_1} -core processor_core \
  -mode edt_int_stuck

report_core_descriptions
report_clocks

set_system_mode analysis
write_tsdb_data -replace

# Read the patterns to be retargeted
read_patterns -module processor_core -fault_type stuck -mode edt_int_stuck
set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]

write_patterns patterns/processor_core_edt_stuck_retargeted.v -verilog \
  -parallel -replace -begin 0 -end 7 -scan \
  -parameter_list {SIM_KEEP_PATH 1}
write_patterns patterns/processor_core_edt_stuck_retargeted_serial.v \
  -verilog -serial -replace -Begin 0 -End 2 \
  -parameter_list {SIM_KEEP_PATH 1}

# Write out the STIL file to use on the tester
write_patterns patterns/processor_core_edt_stuck_retargeted.stil \
  -stil -replace
exit
```

The following dofile snippet shows how you would retarget at-speed transition ATPG patterns for the wrapped core, `gps_baseband`. Commands that are not shown are the same as the commands for processor core in [Example 5-11](#) on page 177.

Example 5-12. Retarget At-Speed Transition ATPG Patterns for the Top-Level

```
# Set the context and open TSDBs ...

# Read in cell libraries, PLL, macro IO pads, designs ...

# Set the current design ...
```

```
# Retarget gps_baseband transition patterns
set_current_mode retarget2_gps_transition
set_static_dft_signal_values retargeting2_mode 1
add_core_instances -instances {GPS_1 GPS_2} -core gps_baseband \
  -mode edt_int_transition
report_core_descriptions
import_clocks -verbose
report_clocks

set_system_mode analysis
write_tsdb_data -replace
# Read the patterns to be retargeted
read_patterns -module gps_baseband -mode edt_int_transition \
  -fault_type transition
set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]

write_patterns patterns/gps_edt_transition_retargeted.v -verilog \
  -parallel -replace -begin 0 -end 7 -scan \
  -parameter_list {SIM_KEEP_PATH 1}
write_patterns patterns/gps_edt_transition_retargeted_serial.v \
  -verilog -serial -replace -Begin 0 -End 2 \
  -parameter_list {SIM_KEEP_PATH 1}

# Write out the STIL file to use on the tester
write_patterns patterns/gps_edt_transition_retargeted.stil -stil -replace

exit
```

RTL and Scan DFT Insertion Flow for Sub-Blocks

When performing the DFT insertion flow with sub-blocks, you insert MemoryBIST and pre-DFT DRCs at the sub-block level and then move up to the sub-block's next parent physical block level (where the sub-block is instantiated) to perform synthesis and scan insertion.

Refer to “[Hierarchical DFT Terminology](#)” for more information about sub-blocks.

You may want to use the sub-block flow for any of the following reasons.

- **Multiple instantiations** — You only need to perform the DFT insertion flow once for a sub-block. Thereafter, every instantiation of the sub-block includes the inserted DFT hardware.
- **Small size** — Most sub-blocks are not big enough to be considered their own physical regions.
- **Readiness** — Sometimes the sub-block RTL is complete before the RTL for the physical layout region, thus you can begin DFT insertion on the sub-block as soon as RTL is ready.

DFT Insertion Flow for the Sub-Block 180

DFT Insertion Flow for the Next Parent Level..... 181


DFT Insertion Flow for the Sub-Block

The DFT insertion flow for sub-blocks is similar to the insertion flow for physical blocks with a few exceptions.

- You do not perform synthesis or scan insertion at the sub-block level because the sub-block netlist may not exist after synthesis.
- During the second DFT insertion pass, you only insert pre-DFT DRCs.

Typically, you do not insert EDT controllers at the sub-block level because the logic inside of the sub-block is too small, and the sub-block module may not exist after synthesis. You can insert an EDT controller at the next parent level that you dedicate to testing the logic inside of a sub-block. In most cases, this EDT controller is active at the same time as other EDT controllers that are present at the next parent level.

Note

 The sub-block flow for inserting MemoryBIST and pre-DFT DRCs follows the same steps as described in “[RTL and Scan DFT Insertion Flow for Physical Blocks](#),” with the exception of generating the EDT and OCC hardware. There are slight variations, which are described in the following procedure.

Procedure

1. [First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#). Ensure that you set the design level to “sub_block” rather than “physical_block”.

```
set_design_level sub_block
```

2. Second DFT insertion pass: insert pre-DFT DRCs. Follow the steps for the “[Second DFT Insertion Pass: Inserting Block-Level EDT and OCC](#)” procedure, excluding generating the EDT and OCC hardware.

Because you specified that you were working at the sub_block level in the first DFT insertion pass, you do not need to re-specify this information for the second insertion pass.

After specifying and verifying the DFT requirements, Tessent Shell performs the following tasks automatically:

- At the sub-block level, any static DFT signals you have added are implemented as IJTAG ports rather than inserted via TDRs. Tessent Shell automatically connects the IJTAG ports to the TDR at the next parent level.
- At the next parent level, adds DFT signals such as ltest_en and async_set_reset_static_disable. Tessent Shell infers the add_dft_control_point command on the sub-block pins if you have DFT DRCs that need to be fixed.
- At the next parent level, adds the tck_occ_en DFT signal if there is a STI-SIB network inside the sub-block.

The following command performs pre-DFT DRC:

```
set_dft_specification_requirements -logic_test on
```

During ICL extraction, Tessent Shell generates the Synopsys Design Constraints (SDC) for the sub-block.

Results

Proceed to performing the DFT insertion flow on the next parent level where this sub-block is instantiated.


DFT Insertion Flow for the Next Parent Level

The next parent level in the bottom-up flow where a sub-block is instantiated may either be a physical layout block that is a wrapped core, or the physical layout region that is the chip (the top level). By moving the focus of DFT insertion from child block to immediate parent block, you maintain the correctness and consistency of the test hardware. The following procedure describes the flow when you have a sub-block instantiated at the chip level.

Prerequisites

- You have performed the DFT insertion flow as described in “[DFT Insertion Flow for the Sub-Block](#)” for the sub-blocks instantiated at this parent level so that you have the design after a clean pre-DFT DRC run.

Note

 This flow occurs after the DFT insertion process described in “[RTL and Scan DFT Insertion Flow for the Top Chip](#)”. When you have a sub-block inserted inside a wrapped core, you complete the procedures described in “[RTL and Scan DFT Insertion Flow for Physical Blocks](#)” on page 146 after you load the sub-block design.

Procedure

1. [First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan](#). When you load the design, open the sub-block’s TSDB and load the design of the sub-block that passed pre-DFT DRCs.
2. [Second DFT Insertion Pass: Inserting Top-Level EDT and OCC](#). When you load the design, open the sub-block’s TSDB, load the full design for the sub-block, and load the design for the top-level after the first DFT insertion pass.
3. [Synthesis](#). Synthesis of the chip-level RTL and the sub-block’s post-DFT inserted RTL occur at the same time. Tessent Shell merges the sub-block into the parent-level logic.
Refer to “[Timing Constraints \(SDC\)](#)” to learn how the synthesis constraints from the sub-block are merged at the next parent level.
4. [Scan Chain Insertion](#). Tessent Shell performs scan chain insertion on the sub-block and parent-level logic at the same time.
5. [ATPG Pattern Generation](#).

Examples

Design Loading for the First DFT Insertion Pass

The following example shows opening the TSDB for the sub-block and using `read_design` to read in the sub-block (`processor_core`) design.

```
# Set the context to insert DFT into RTL-level design
set_context dft -rtl -design_id rtl1
# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_top

# Open the TSDBs for all the instantiated sub-blocks
open_tsdb ../../sub_block/tsdb_outdir
# Read the tessent cell library
read_cell_library ../../library/tessent/adk.tcelllib
# Read the design
read_verilog ../../rtl/chip_top.v
```

```
# Explicitly load the sub-block netlist
read_design_processor_core -design_id rtl2 -verbose
set_current_design chip_top
set_design_level chip
```

```
# Follow the rest of the flow for the first DFT insertion pass for a chip.
```

Design Loading for the Second DFT Insertion Pass

```
# Set the context to insert DFT. Define a new design ID
set_context dft -rtl -design_id rtl2
```

```
# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_top
```

```
# Open the TSDBs for all the instantiated sub-blocks
open_tsdb ../../sub_block/tsdb_outdir
```

```
# Read the tessent cell library
read_cell_library ../../library/tessent/adk.tcelllib
```

```
# Read the verilog
read_verilog ../../rtl/noncore_blocks/p11.v -blackboxset_design_sources \
             -format verilog -v ../../rtl/noncore_blocks/iopad_sel.v \
             -v ../../rtl/noncore_blocks/iopad.v
```

```
read_design chip_top -design_id rtl1 -verbose
# Explicitly load the sub-block netlist
read_design_processor_core -design_id rtl2 -verbose
```

```
set_current_design chip_top
```

```
# Follow the rest of the flow for the second DFT insertion pass for a chip
```

RTL and Scan DFT Insertion Flow for Instrument Blocks

When performing the DFT insertion flow with instrument blocks, create a special empty module to insert DFT elements into, then move up to the instrument block's next parent physical block level to perform synthesis and scan insertion.

You may want to use the instrument block flow if you want to avoid having Tessent Shell modify your golden RTL.

DFT Insertion Flow for the Instrument Block	184
Instrument Block DFT Insertion Flow for the Next Parent Level	187

DFT Insertion Flow for the Instrument Block

Create a special empty module into which the DFT elements are inserted.

Procedure

1. Start with an empty DFT module that contains the port definitions for IJTAG and an input and output port for each functional clock. Optionally, create output ports for the `all_test` and `async_set_reset_dynamic_disable` DFT signals, which you can use to control clock multiplexers and turn off your asynchronous set and reset signals during scan shifting.

The following RTL shows an example module definition required for the instrument block flow:

```

module elt1_dft_box (
  input clk,
  output clk_occ,
  input ijtag_tck,
  input ijtag_reset,
  input ijtag_ce,
  input ijtag_se,
  input ijtag_ue,
  input ijtag_sel,
  input ijtag_si,
  output ijtag_so,
  input [1:0] edt_channels_in,
  output edt_channels_out,
  input scan_en,
  input test_clock,
  input edt_update,
  output async_set_reset_dynamic_disable
);

// Drive the DFT signal to its off value

assign async_set_reset_dynamic_disable = 1'b0;

// Add feedthroughs for ijtag chain
assign ijtag_so = ijtag_si;
// Add feedthrough for func clock that will be equipped with OCC
assign clk_occ = clk;
endmodule

```

2. Perform the DFT insertion flow within the DFT module as normal, but you must add DFT control points to output ports corresponding to DFT signals.

```
read_verilog design/rtl/elt1_dft_box.v
set_current_design elt1_dft_box
set_design_level instrument_block

set_dft_specification_requirements -logic_test on
add_dft_signals ltest_en int_mode int_ltest_en ext_mode ext_ltest_en
add_dft_signals scan_en test_clock edt_update \
  -source_node {scan_en test_clock edt_update}
add_dft_signals edt_clock shift_capture_clock \
  -create_from_other_signals
add_dft_signals x_bounding_en mcp_bounding_en \
  observe_test_point_en control_test_point_en -create_with_tdr
add_dft_signals async_set_reset_static_disable
add_dft_signals async_set_reset_dynamic_disable \
  -create_from_other_signals
add_dft_signals capture_per_cycle_static_en -create_with_tdr

#Add DFT control points
add_dft_control_points async_set_reset_dynamic_disable \
  -type dynamic_dft_control \
  -dft_signal_source_name async_set_reset_dynamic_disable

add_clocks clk -period [expr {1000.0/300.0}]

check_design_rules

set_spec [create_dft_specification -sri_sib_list {edt lbist occ}]
set_config_value use_rtl_cells off -in_wrapper $spec

read_config_data -in_wrapper $spec -from_string {
  Edt { // {{{
    ijtag_host_interface : Sib(edt);
    Controller(c1) {
      ...
      Connections {
        EdtChannelsIn(1:2) {
          port_pin_name : edt_channels_in;
        }
        EdtChannelsOut(1) {
          port_pin_name : edt_channels_out;
        }
      }
    }
  } // }}}
  LogicBist { // {{{
    ijtag_host_interface : Sib(lbist);
    Controller(lbist_1) {
      ...
      Connections {
        shift_clock_src : clk;
      }
    }
    NcpIndexDecoder {
      ...
    }
  } // }}}
  OCC { // {{{
    ijtag_host_interface : Sib(occ);
```

```
        static_clock_control : external;  
        Controller(clk_controller) {  
            clock_intercept_node : clk;  
        }  
    } // }}}  
}  
  
process_dft_specification  
set_system_mode setup  
extract_icl  
write_design_import_script -replace -use_relative_path_to [pwd]
```

Results

Proceed to performing the DFT insertion flow on the next parent level where this instrument block is instantiated.

Instrument Block DFT Insertion Flow for the Next Parent Level

The next parent level in the bottom-up flow where an instrument block is instantiated may either be a physical layout block that is a wrapped core, or the physical layout region that is the chip (the top level). By moving the focus of DFT insertion from child block to immediate parent block, you maintain the correctness and consistency of the test hardware. The following procedure describes the flow when you have an instrument block instantiated at the `physical_block` level.


Prerequisites

- You have performed the DFT insertion flow as described in [DFT Insertion Flow for the Instrument Block](#) for the instrument blocks instantiated at this parent level so that you have the design after a clean pre-DFT DRC run.

Procedure

1. Instantiate and connect the special empty DFT module in your parent module. Modify your RTL to use `clk_occ` wherever it used `clk` so that the clock of all your scannable elements is controlled by the OCC inserted into the block.

Tip

 Remember to manually connect in the parent module all the connections for the JTAG network and all dynamic DFT signals.

2. At the next level, read your design normally, and then load the DFT module using the `read_design` command. Specify your clocks and run the `check_design_rules` command to validate and store them. Then, call the `extract_icl` command.

At that point, it is as if you had inserted the DFT elements from this level using the normal flow. After ICL is extracted, you can go to the DFT context and verify that the

RTL has a controllable clock and reset signals. DRC violation errors indicate if the RTL is not clean, which you must fix manually.

```
read_design elt1_dft_box -design_id dft
read_verilog design/rtl/elt1.v
set_current_design elt1
set_design_level physical_block
add_clocks CLK1 -period [expr {1000.0/300.0}]
check_design_rules
extract_icl
report_dft_signals
write_design_import_script -replace -use_relative_path_to [pwd]

#Check that it passes the pre-DFT DRC rules
set_context dft
set_dft_specification_requirements -logic_test on
set_drc_handling dft_c9 -auto_fix off
check_design_rules

# Generate patterns
set_spec [create_pattern_specification]
process_pattern_specification

# Run Simulation
set_simulation_library_sources -v ../techlib_adk.tnt/verilog/adk.v
run_testbench_simulations
```

3. **Synthesis.** Synthesis of the chip-level RTL and the instrument_block's post-DFT inserted RTL occur at the same time. Tessent Shell merges the instrument_block into the parent-level logic.

Refer to “[Timing Constraints \(SDC\)](#)” to learn how the synthesis constraints from the instrument_block are merged at the next parent level.

4. **Scan Chain Insertion.** Tessent Shell performs scan chain insertion on the instrument_block and parent-level logic at the same time.
5. **ATPG Pattern Generation.**


RTL and DFT Insertion Flow With Third-Party Scan

The RTL and DFT insertion flow enables you to use a third-party scan insertion tool instead of Tessent Scan. Regardless of which scan insertion tool you use, you must follow a number of sequences in the DFT insertion flow that include creating the logic for internal and external mode, and the multiplexing mechanism that Tessent Scan normally inserts in order to support hierarchical ATPG.


The workflow uses the standard hierarchical DFT insertion flow, both for each wrapped core and, once you have inserted DFT in each wrapped core, for top chip.

See “[Hierarchical DFT Terminology](#)” on page 142.

Note

 If your design consists of wrapped cores as your lower level physical blocks, and the wrapped cores do not contain embedded pad IOs, refer to “[How to Use Boundary Scan in a Wrapped Core](#)” on page 502.

Caution

 Third-party tools may have different insertion capabilities compared to Tessent Scan. This may affect the quality of the results you achieve.

DFT Insertion Flow With Third-Party Scan Insertion	189
Wrapped Core DFT Insertion With Third-Party Scan	191
Top Chip DFT Insertion with Third-Party Scan	206


DFT Insertion Flow With Third-Party Scan Insertion

The DFT insertion flow with third-party scan insertion requires you to manually collect and provide certain data during hierarchical ATPG. Normally when using Tessent Scan for scan insertion and stitching, the information that is needed for performing hierarchical ATPG is transferred.

Prerequisites

Before starting this flow, you should identify the DFT functions you need to insert in each core and identify how many scan chains and wrapper cells are needed for each core.

Note

 Tessent Scan automatically concatenates scan chain and wrapper chains into mixed chains to achieve the number of scan channels on the EDT logic.

CTL (IEEE 1450.6) Generation for Tessent IP With Embedded Scan Segments

Certain Tessent Shell IP blocks include embedded scan segments that must be a part of scan chains. To simplify integration with third-party scan insertion tools, Tessent Shell generates the CTL (IEEE 1450.6) file whenever generating the `tcd_scan` file for test IP.

The tool generates a CTL file that includes information about embedded scan segments for the following applications:

- A CTL file to describe the embedded chain to help in connecting the programmable registers inside the following IP:
 - OCC
 - STI SIB or Sib(STI)
- A CTL file to describe the controller chain mode (CCM) scan segment, when the tool generates test IP with enabled `segment_per_instrument` property. This applies to the following IP:
 - LogicBIST controller
 - EDT controller (available only when used as part of hybrid TK/LBIST IP)
 - Single Chain Mode Logic controller (part of hybrid TK/LBIST IP)
 - InSystemTest (MissionMode) controller

For information about CCM, see “[Controller Chain Mode](#)” in the *Hybrid TK/LBIST User’s Manual*. For information about CTL files, see “[Default Generation of CTL Files](#)” in the *Tessent Shell Reference Manual*.

Wrapped Core DFT Insertion With Third-Party Scan

For each wrapped core, perform a two-pass pre-scan DFT insertion process as you would for a flat design except that in the first DFT insertion pass, do not insert boundary scan unless you have embedded pad IO macros present inside the core. The flow details are unique to working with wrapped cores and using a third-party scan insertion tool to insert the scan.

Refer to “[RTL and Scan DFT Insertion Flow for Physical Blocks](#)” on page 146 for overall details.

Note


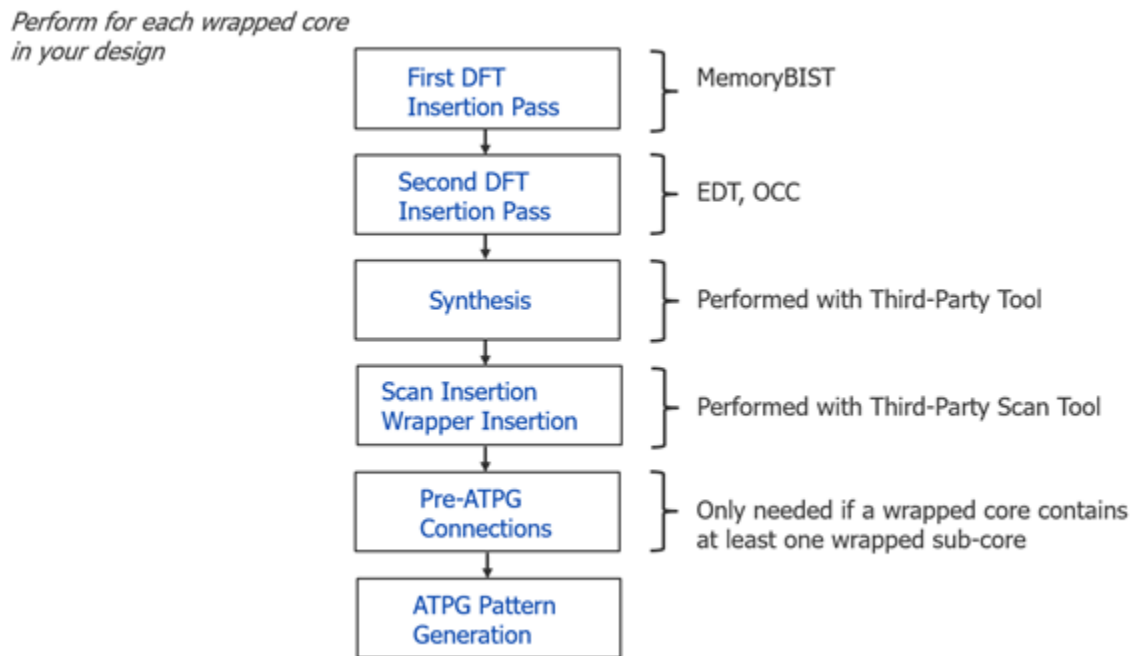
 This discussion assumes your design consists of wrapped cores as your lower level physical blocks, and the wrapped cores do not contain embedded pad IOs.

Figure 5-21. Two-Pass Insertion Flow for RTL, Wrapped Cores, and Third-Party Scan



Perform Two-Pass DFT Insertion and Synthesis for Wrapped Cores	192
Third-Party Scan Insertion for Wrapped Cores	193
Writing the Design Back to the TSDB	197
Make Pre-ATPG Connections with Third-Party Scan for Wrapped Cores	200
Performing Wrapped Core Graybox Generation and ATPG Pattern Generation....	202

Perform Two-Pass DFT Insertion and Synthesis for Wrapped Cores

This flow for wrapped cores is the same as the DFT insertion flow for physical blocks except you insert the scan with a third-party scan tool instead of Tessent Scan.

Procedure

1. [First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#). Ensure you set the design level to “physical_block”.

```
set_design_level physical_block
```

2. [Second DFT Insertion Pass: Inserting Block-Level EDT and OCC](#). Ensure you set the design level to “physical_block”.
 - a. During the second insertion pass and when specifying the DFT signals, follow the procedure defined in “[Specifying and Verifying the DFT Requirements: DFT Signals for Wrapped Cores](#)” on page 152.

The following are the internal and external modes that are required for mode switching:

```
add_dft_signals int_ltest_en ext_ltest_en int_mode ext_mode \  
                -create_with_tdr
```

```
add_dft_signals input_wrapper_scan_en output_wrapper_scan_en \  
                -create_from_other_signals
```

- b. [Creating the DFT Specification](#).

The test case implements compressed ATPG test using Tessent TestKompress. You define the EDT logic for Tessent TestKompress and the Tessent OCC in an EDT DftSpecification wrapper as follows:

```
read_config_data -in $spec -from_string "  
  OCC {  
    ijtag_host_interface : Sib(occ);  
    type : standard;  
  }  
  EDT {  
    ijtag_host_interface : Sib(edt);  
    Controller (c1) {  
      longest_chain_range : 50, 65;  
      scan_chain_count : 80;  
      input_channel_count : 2;  
      output_channel_count : 1;  
      Connections {  
        EdtChannelsIn(2:1) {  
          port_pin_name : edt_channel_in[1:0];  
        }  
        EdtChannelsOut(1:1) {  
          port_pin_name : edt_channel_out[0:0];  
        }  
      }  
    }  
  }  
}"
```

You define the scan_chain_count as follows:

```
scan_chain_count = number of internal only scan chains + number of wrapper  
chains
```

This number covers all chains that the EDT logic at the current level needs to connect to. In other words the EDT hardware need to be connected to both the internal only chains and wrapper chains.

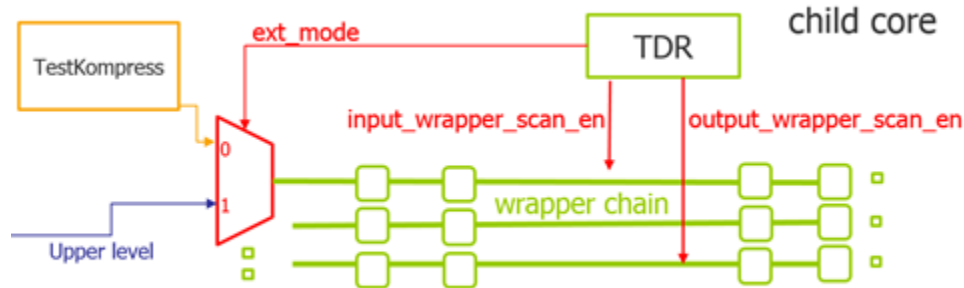
3. [Generating the EDT, Hybrid TK/LBIST, and OCC Hardware.](#)
4. [Extracting the ICL Module Description.](#)
5. [Generating ICL Patterns and Running Simulation.](#)
6. [Performing Synthesis.](#)

Third-Party Scan Insertion for Wrapped Cores

You can use a third-party scan insertion tool to insert scan and perform scan stitching for wrapped cores in your design. The actual process depends on the third-party scan insertion tool you use. You must wrap all the inputs and outputs of hierarchical physical blocks and perform scan insertion and stitching in your design. Your choice of wrapper cells depends on the capabilities of the third-party scan tool you are using.

For hierarchical ATPG, you must insert multiplexers before each wrapper chain to multiplex the primary input from the top level and the current level of the EDT scan input. Figure 5-22 shows an abstract level depiction of the multiplexing logic. The inclusion of these multiplexers is mandatory for hierarchical ATPG support using Tessent tools.

Figure 5-22. DFT Signals and Multiplexer Logic Support Hierarchical ATPG



To the child cores, the preregistered static DFT signals `ext_mode` and `int_mode` control these multiplexers as follows:

- **External Mode** — The `ext_mode` DFT signal is active, and the `int_mode` DFT signal is inactive. The top level drives input into the wrapper chains through the wrapper’s scan in on the core boundary.
- **Internal Mode** — The `ext_mode` DFT signal is inactive and the `int_mode` DFT signal is active. The EDT logic drives all the scan chains inside the hierarchical physical block (both internal only and wrapper chains).

When using this configuration, you must control the corresponding `TDR` bits for mode switching. See “[Performing Wrapped Core Graybox Generation and ATPG Pattern Generation](#)” on page 202 for a detailed explanation.

Usage Guidelines

Use the following guidelines and criterion for configuring your third-party scan insertion tool for wrapper and scan stitching:

- For the hierarchical physical block on which you intend to perform scan insertion, you have completed the “[Perform Two-Pass DFT Insertion and Synthesis for Wrapped Cores](#)” on page 192 flow.
- You have identified the scan chains of the current core and how many of the scan chains should be wrapper chains. The total number of scan chains is the same as the EDT `scan_chain_count`, as defined in the previous DFT insertion step. The rest of the scan chains can be specified and used as core chains.

- During wrapper analysis, you must exclude the following pins from any wrapper insertion:
 - IJTAG-related pins
 - `edt_channel_input` pins
 - `edt_channel_output` pins
 - `scan_enable` pins
 - Any clock pins
- The DFT signals for input and output wrapper scan enable that you have defined during [the second DFT insertion pass](#) should be used as scan enable signals for input and output wrapper chains, respectively, during wrapper analysis and insertion.
- To run ATPG after scan insertion, you must create and supply a test procedure file for the [Graybox generation step](#) to trace and control the wrapper chains. A test procedure file tells the tool how to clock the scan chains, and the tool automatically passes this to ATPG in the Tessent Shell flow. A graybox does not normally include an OCC and EDT logic for tracing and controlling wrapper chains; consequently, you must create the test procedure file manually when using the third-party scan insertion flow.

See “[Test Procedure File](#)” on page 533 for complete details on how you create and use a test procedure file. See also “[Example Test Procedure File](#)” on page 198.

- The Tessent Shell environment `get_dft_info_dictionary` command offers you a method to access the Tessent Database (TSDB) to use this information with your third-party scan insertion tool.

The command reads the scan information from the TSDB’s `dft_inserted_designs` directory, specifically in a Tcl dictionary file named:

`<design_name>.dft_info_dictionary`

The Tcl dictionary contains the information about the DFT inserted in the design that must be considered during scan insertion when using a third-party scan insertion tool. See “[Example dft_info_dictionary File](#)” on page 199.

All instances under "non_scannable_instance_list" should be regarded as non-scan. All modules under "modules_with_chains" should be considered as sub-chains and should not be modified during scan insertion.

- The Tessent Shell environment provides a mechanism for generating an example usage script you can customize to work with your third-party scan insertion tool. In the Tessent Shell tool, invoke the following commands in the `dft` or `pattern` contexts:

```
read_verilog design_netlist
source \
../tsdb_outdir/dft_inserted_designs/
design_name.last_DFT_insertion_design_id/
design_name.dft_info_dictionary
get_dft_info_dictionary -example_usage_script
```

After issuing these commands, the tool creates an example usage script. Use this script as a starting example to convert the dictionary into the specific commands used by your third-party scan insertion tool. In the file, the tool inserts pound signs (#) with comments that specify which actions your third-party tool must perform. For example:

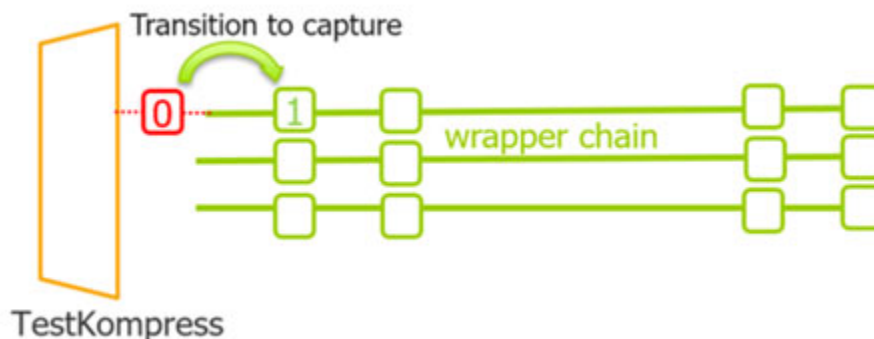
```
puts "Declare the Non-Scannable instances"

set non_scannable_instance_list [dict get $tessent_dft_info_dict
non_scannable_instance_list]
if {[llength $non_scannable_instance_list] > 0} {
  ### Command to declare the Non-Scannable instances
  add_nonscan_instances [get_instances
$non_scannable_instance_list]
}
...
```

After you add your third-party tool specific commands to the file, source the file to your scan insertion tool to finish the scan insertion and stitching.

- For the hierarchical transition fault model, you may need to insert one additional at-speed cell at the beginning of each wrapper chain to make the transition on the first cell of each wrapper chain. This way, you achieve the best coverage. The method you use depends on the third-party insertion tool. [Figure 5-23](#) shows the logic.

Figure 5-23. At-Speed Flows in the Wrapper Chain



- If required and after you have inserted the scan using your third-party tool, you can follow the procedure described in [“Make Pre-ATPG Connections with Third-Party Scan for Wrapped Cores”](#) on page 200 if required.

- If no pre-ATPG connections are required, then you can load the scan-inserted netlist into Tessent Shell and write the modified netlist and scan information back to the Tessent Shell Database (TSDB).

Writing the Design Back to the TSDB

After completing scan insertion by your third-party scan insertion tool, you must write the scan-inserted netlist back to the TSDB to associate the netlist with Tessent DFT instruments. You create this association by creating a softlink to the scan inserted netlist in TSDB, which also creates the Tessent Core Description files and the ICL information.

Prerequisites

- You have completed scan insertion with a third-party tool.

Use the following procedure to save a scan-inserted netlist to the TSDB. The line numbers are represented in the Example dofile under Examples:

Procedure

1. Set the context (see lines 1-3).
2. Set the TSDB location if not already set (see line 6).
3. Load the cell library (see line 9).
4. Load the scan-inserted netlist (see line 12). This netlist is modified by your third-party scan insertion tool and contains the scan cells.
5. Load the supporting DFT files (except the netlist) from the last DFT insertion pass and set the current design (see lines 15-17).
6. Set the design level. Make sure you specify “physical_block” (see line 21).
7. Write the design information to the TSDB and create the softlink (see line 22).

Examples

Example dofile

```
1 # Use the design_id as "scan." Use this in all the ATPG
2 # runs that this design is read in.
3 set_context patterns -ijtag -design_id <scan>
4
5 # Set the location of the TSDB. Default is the current working directory
6 set_tsdb_output_directory ../tsdb_outdir
7
8 # Read the Tessent Cell Library
9 read_cell_library ../../../../library/tessent/adk.tcelllib
10
11 # Read the scan inserted netlist and elaborate the design
12 read_verilog ../3.synthesize_rtl/<current_core>_scan.vg
13
14 # Read the -no_hdl from the last DFT insertion pass
```

```
15 read_design <current_core> -design_id <rtl2> -no_hdl -verbose
16 read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v
17 set_current_design <current_core>
18
19 # Specify the design level before writing out a softlink of the design in
20 # TSDB
21 set_design_level physical_block
22 write_design -tsdb -softlink_netlist -verbose
23 exit
```

Example Test Procedure File

```
set time scale 1.000000 ns ;
timeplate gen_tpl =
    force_pi 0 ;
    measure_po 10 ;
    pulse_clock 20 10 ;
    pulse clock2 20 10;
    period 40 ;
end;
procedure shift =
    scan_group grp1 ;
    timeplate gen_tpl ;
    // cycle 0 starts at time 0
    cycle =
        force_sci ;
        measure_sco ;
        pulse clock1;
        pulse clock2;
    end;
end;

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tpl ;
    // cycle 0 starts at time 0
    cycle =
        force clock1 0;
        force clock2 0;
        force scan_enable 1 ;
    end ;
    apply shift 76;
end;
```

Example dft_info_dictionary File

```

set tessent_dft_info_dict {
  version 1
  dft_signals {
    dft_signal_name {
      connection_node_name node_name
      connection_node_type port|pin
      forced_value_in_pre_scan_drc 0|1
    }
  }
  modules_with_chains {
    module_name {
      pre_scan_drc_on_boundary_only 0|1
      module_type normal|memory|occ
      is_hard_module 0|1
      internal_scan_only 0|1
      allow_scan_out_retiming 0|1
      instance_list {inst_name ...}
      scan_en_ports|ltest_en_ports|set_reset_ports {
        port_name { active_polarity 0|1
        }
      }
      clock_ports {
        port_name {
          off_state 0|1
        }
      }
      clock_out_ports_or_pins {
        port_or_pin_name {}
      }
      scan_chains {
        scan_chain_name {
          length auto|int
          scan_in_port port_name
          scan_in_clock_name port_name
          scan_in_clock_inversion 0|1
          scan_out_port port_name
          scan_out_clock_name port_name
          scan_out_clock_inversion 0|1
        }
      }
    }
  }
  non_scannable_instance_list {inst_name ...}
  edt_instances {
    instance_name {
      edt_module_name module_name
      scan_chains {
        chain_name {
          scan_in_port port_name
          scan_out_port port_name
        }
      }
    }
  }
}

```

Make Pre-ATPG Connections with Third-Party Scan for Wrapped Cores

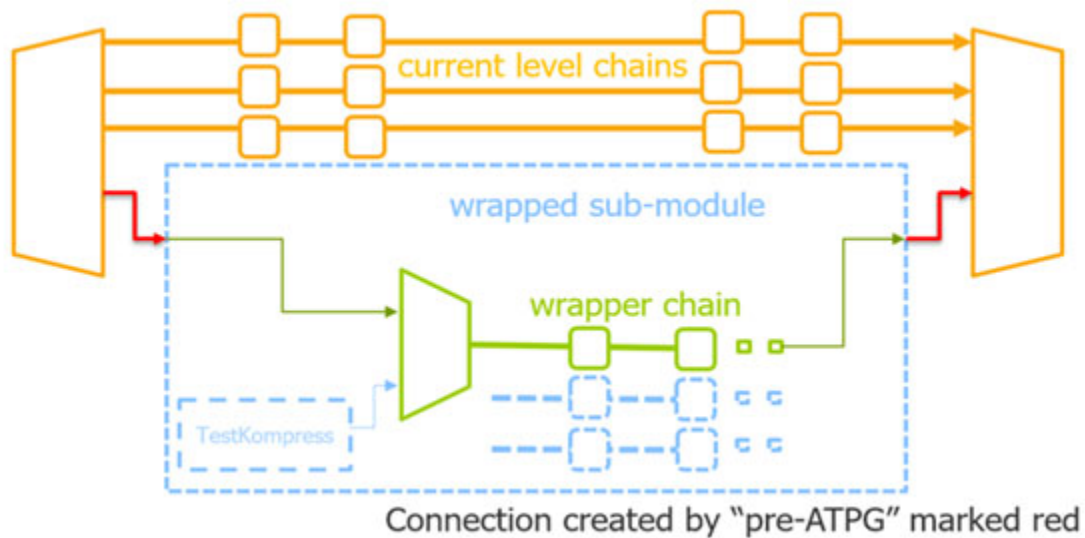
You perform this step in flow only if the wrapped core has at least one child wrapped core or there is one upper layer beyond the current module.

Otherwise, skip this step in the flow and proceed directly to “[Performing Wrapped Core Graybox Generation and ATPG Pattern Generation](#)” on page 202.

This step includes two parts:

1. It creates connections from the current level EDT logic to the wrapped child cores' wrapper chains for the paths depicted in red in [Figure 5-24](#).


Figure 5-24. Current Level Path to Child Core Wrapper Chains



The external mode logic and chains of child cores become part of the current test coverage.

2. It also creates logic from the current level wrapper chain to the upper-level logic, marked in green in [Figure 5-24](#). Logic added at this stage includes the muxes controlled by `dft_signals`, ports at the module boundary, and connections between those ports and the EDT logic. This enables the external mode testing of the wrapped child core from the upper level.

Note

 The line numbers used in this procedure refer to the command flow dofile in “[Example Dofile for Pre-ATPG Connections for Wrapped Cores](#)” on page 201.

Prerequisites

- You must have performed the two-pass DFT insertion as described in “[Perform Two-Pass DFT Insertion and Synthesis for Wrapped Cores](#)” on page 192.
- You must have inserted scan using your third-party scan insertion tool per the guidelines provided in “[Third-Party Scan Insertion for Wrapped Cores](#)” on page 193.

Procedure

1. Load the design. (See lines 1-7.)
2. Change to insertion mode. (See line 9.)
3. Make the red connections shown in [Figure 5-24](#) on page 200. (See lines 12-26.)
4. Make the ports, logic, and connections shown in green in [Figure 5-24](#). (See lines 28-43.)
5. Save the design. (See line 45.)
6. Write the design back to the TSDB. (See lines 30-66.)

Examples

Example Dofile for Pre-ATPG Connections for Wrapped Cores

```
1 set_context dft -no_rtl
2
3 read_verilog ../3.synthesis/<current_design_name>_scan.vg
4
5 read_cell_library ../../../../library/tessent/adk.tcelllib
6
7 set_current_design <current_design_name>
8
9 set_system_mode insertion
10
11
12 # Make the connections marked in red in the preceding figure (see Step 1)
13
14 # empty pins of edt logic were grounded, need to delete before connection
15 delete_connection <current_edt_instance>/edt_scan_out[]
16
17
18 delete_connection <current_edt_instance>/edt_scan_in[]
19
20
21 # connect edt scan-in/scan-out to ext_wsi/ext_wso of every chain
22 create_connection /<child_core_instance>/ext_wsi[] \
23                 <current_edt_instance>/edt_scan_in[]
24
25 create_connection /<child_core_instance>/ext_wso[] \
26                 <current_edt_instance>/edt_scan_out[]
27
28 # Make the ports, logic, and connection marked in green in the preceding
29 # figure (see Step 2)
30 delete_connection <current_edt_instance>/edt_scan_in[]
31 # create ports for upper level to connect to
32 create_port ext_wsi[]
```

```
32 create_port ext_wso[] -direction output
33 # create mux logic and control logic
34 create_instance wrapper_mux0 -of_module mux21
35 create_instance ext_mode_inv -of_module inv01
36 create_instance mode_and -of_module and02
37 create_connection <current_tdr_instance>/ext_mode ext_mode_inv/A
38 create_connection <current_tdr_instance>/int_mode mode_and/A0
39 # connect mux between ports and wrapper chain beginning/ends
40 create_connection wrapper_mux0/A1 <edt_instance>/edt_scan_in[]
41 create_connection wrapper_mux0/A0 ext_wsi[]
42 create_connection wrapper_mux0/Y <wrapper chain beginning cell or buffer
before it>/A
43 create_connection ext_wso[] <wrapper chain ending cell or buffer after
it>/Q
44
45 write_design -output_file ../3.synthesis/<current_design_name>.vg
46
47 # Use the design_id as "scan." Use this in all the ATPG
48 # runs that this design is read in.
49
50 set_context patterns -ijtag -design_id <scan>
51
52 # Set the location of the TSDB. Default is the current working directory
53 set_tsdb_output_directory ../tsdb_outdir
54
55 # Read the Tessent Cell Library
56 read_cell_library ../../../../library/tessent/adk.tcelllib
57
58 # Read the scan inserted netlist and elaborate the design
59 read_verilog ../3.synthesize_rtl/<current_core>_scan.vg
60
61 # Read the -no_hdl from the last DFT insertion pass
62
63 read_design <current_core> -design_id <rtl2> -no_hdl -verbose
64
65 read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v
66
67 set_current_design <current_core>
68
69 # Specify the design level before writing out a softlink of the design in
70 # TSDB
71 set_design_level physical_block
72 write_design -tsdb -softlink_netlist -verbose
73 exit
```

Performing Wrapped Core Graybox Generation and ATPG Pattern Generation

This step in the flow creates the graybox model of the current wrapped core and generates ATPG patterns. You must perform this step for each wrapped core in your design.

Prerequisites

- You must have completed the “[Performing Wrapped Core Graybox Generation and ATPG Pattern Generation](#)” on page 202 step for each wrapped core.

- You must have inserted scan with your third-party scan insertion tool per the guidelines cited in “[Third-Party Scan Insertion for Wrapped Cores](#)” on page 193 for each wrapped core, and written the scan-inserted netlist back into the Tessent Shell Database as described in “[Writing the Design Back to the TSDB](#)” on page 197.
- If required, make ATPG connections using the process outlined in “[Make Pre-ATPG Connections with Third-Party Scan for Wrapped Cores](#)” on page 200 for each wrapped core.

Procedure

1. Generate the graybox model for the wrapped core and add a scan group of wrapper chains by importing the [test procedure file](#) you created during [Third-Party Scan Insertion for Wrapped Cores](#)—see “[Example Graybox Generation](#)” on page 204 and “[Performing ATPG Pattern Generation: Wrapped Core](#)” on page 158.

To generate external patterns to check fault coverage for the wrapped core, refer to “[Example External ATPG Pattern Generation](#)” on page 205. You do not retarget these patterns.

As long as the scan mode (scan chains and test logic) is stitched conforming to DRC, Tessent Shell generates internal ATPG patterns formed in the correct way.

2. Save the design and write the patterns to the TSDB using the [write_tsdb_data](#) command.

```
write_tsdb_data -replace
```

3. Repeat Steps 1 and 2 for each wrapped core to generate transition patterns.

All of the information is passed by the Tessent Shell through the TSDB.

Results

When you complete graybox and ATPG for each wrapped core, perform scan insertion for the top chip. See “[Top Chip DFT Insertion with Third-Party Scan](#)” on page 206 for complete flow details.

Examples

Example Graybox Generation

```
set_context pattern -scan -design_id <scan>
set_tsdb_output_directory ../tsdb_outdir
read_cell_library ../../../../library/tessent/adk.tcelllib
read_design <current_design_name> -design_id <scan>
set_current_design <current_design_name>
# set memory modules black box
add_black_box -module SYNC_1RW_32x16_RC_BISR
add_black_box -module SYNC_1RW_32x4
add_clock 0 clock1
# Graybox generation requires core configured to external mode by
# setting DFT signal values:
set_static_dft_signal_values int_mode 0
set_static_dft_signal_values ext_mode 1
set_static_dft_signal_values int_ltest_en 0
set_static_dft_signal_values ext_ltest_en 1
# exclude edt_channels for graybox
set_attribute_value [get_ports *edt_channel*] \
                    -name ignore_for_graybox -value true
# import wrapper chain information by importing testproc file of
# wrapper chains
add_scan_group grp1 ../4.scan_insertion/wrapper.testproc
# add every wrapper chain from input pin to output pin
add_scan_chains chain1 grp1 ext_wsi[0] ext_wso[0]
add_scan_chains chain2 grp1 ext_wsi[1] ext_wso[1]
add_scan_chains chain3 grp1 ext_wsi[2] ext_wso[2]
set_system_mode analysis
# external mode information saved in tsdb
write_design -graybox -tsdb -verbose
exit
```

Example External ATPG Pattern Generation

```

set_context pattern -scan -design_id <scan>
set_tsdb_output_directory ../tsdb_outdir
read_cell_library ../../../../library/tessent/adk.tcelllib
read_design <current_design_name>-design_id <scan> -view graybox
set_current_design <current_design_name>
add_clock 0 clock
# create mode information for external ATPG
set_current_mode ext_multi_stuck -type external
# configure core to external mode
set_static_dft_signal_values int_mode 0
set_static_dft_signal_values ext_mode 1
set_static_dft_signal_values int_ltest_en 0
set_static_dft_signal_values ext_ltest_en 1
# for transition fault ATPG, replace stuck with transition
set_fault_type stuck
# import wrapper chain information
add_scan_group grp1 ../4.scan_insertion/wrapper.testproc
add_scan_chains chain1 grp1 ext_wsi[0] ext_wso[0]
add_scan_chains chain2 grp1 ext_wsi[1] ext_wso[1]
add_scan_chains chain3 grp1 ext_wsi[2] ext_wso[2]
set_system_mode analysis
create_pattern
write_tsdb_data -replace
exit

```

Example Internal ATPG Pattern Generation

```

set_context patterns -scan -design_id <scan>
set_tsdb_output_directory ../tsdb_outdir
read_cell_library ../../../../library/tessent/adk.tcelllib
set_tsdb_output_directory ../tsdb_outdir
read_cell_library ../../../../library/tessent/adk.tcelllib
read_design <current_design_name> -design_id <scan>
add_black_Box -module SYNC_1RW_32x16_RC_BISR
add_black_Box -module SYNC_1RW_32x4
set_current_design <current_design_name>
# current_mode is used when add core instance at top level
set_current_mode edt_int_stuck -type internal
# import core instances of OCC, EDT and sib_sti if exist
add_core_instances -module <occ_instance_name> -parameter_values {}
add_core_instances -module <edt_instance_name>
add_core_instances -module <sib_sti_instance_name>
# for transition fault ATPG, replace stuck with transition
set_fault_type stuck
add_clock 0 clock -pulse_always
# configure core to internal mode
set_static_dft_signal_values int_mode 1
set_static_dft_signal_values ext_mode 0
set_static_dft_signal_values int_ltest_en 1
set_static_dft_signal_values ext_ltest_en 0
report_static_dft_signal_settings
set_system_mode analysis
create_pattern
write_tsdb_data -replace
exit

```

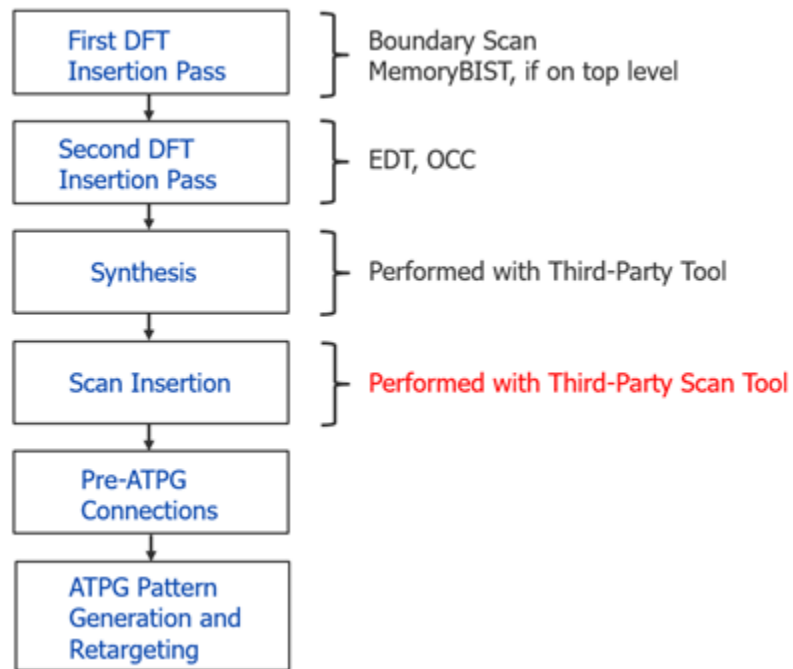
Top Chip DFT Insertion with Third-Party Scan

After performing the RTL and scan DFT insertion flow for each wrapped core in your design, you can perform the DFT insertion process for the top-level chip design using a third-party scan insertion tool to insert the scan.

See “[RTL and Scan DFT Insertion Flow for the Top Chip](#)” on page 165 for overall flow details, and a detailed description of the Test Access Mechanism (TAM).

Figure 5-25. Two-Pass Insertion Flow for RTL, Top Level, and Third-Party Scan

*Perform after wrapped core
 DFT insertion*



Perform Two-Pass Insertion and Synthesis for Top Chip	206
Third-Party Scan Insertion for Top Chip	207
Make Pre-ATPG Connections with Third-Party Scan for Top Chip	209
Perform Top Chip Graybox Generation, ATPG Pattern Generation, and Wrapped Core Pattern Retargeting	211

Perform Two-Pass Insertion and Synthesis for Top Chip

After performing the RTL and scan DFT insertion flow for each wrapped core in your design, you can perform the DFT insertion process for the top-level chip design.


Prerequisites

- Before you perform the top level steps, all the child cores must be ready with their retargetable patterns. A spec form for the top level logic is also recommended to define the number of scan chains and what DFT instruments need to be inserted.
- You must have completed the [Perform Two-Pass DFT Insertion and Synthesis for Wrapped Cores](#) step for each wrapped core.
- You must have completed the [Third-Party Scan Insertion for Wrapped Cores](#) step for each wrapped core.
- If required, you must have completed the [Make Pre-ATPG Connections with Third-Party Scan for Wrapped Cores](#) step for each wrapped core.
- You must have completed the [Performing Wrapped Core Graybox Generation and ATPG Pattern Generation](#) for each wrapped core.

Procedure

1. Set the design level to “chip” for the top level of the chip and insert DFT for MemoryBIST and Boundary Scan. Refer to “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” on page 116.
2. Insert top-level EDT and OCC. Refer to “[Second DFT Insertion Pass: EDT, Hybrid TK/LBIST, and OCC](#)” on page 120.

Note

 You must correctly define the scan_chain_count of the EDT, to include all scan chain counts of child cores in addition to the top-level scan chain count.

3. Perform synthesis. Refer to “[Performing Synthesis](#)” on page 132.

Results

You now have a design ready for top level scan insertion with your third-party scan insertion tool. Proceed to “[Third-Party Scan Insertion for Top Chip](#)” on page 207.

Third-Party Scan Insertion for Top Chip

You can use a third-party scan insertion tool to insert scan into the top chip of your design. Your process depends on the third-party scan insertion tool you use.

Usage Guidelines

- Top level logic does not require wrapper chains as all primary inputs and primary outputs are controllable.

- The Tessent Shell environment `get_dft_info_dictionary` command offers you a method to access the Tessent Database (TSDB) to use this information with your third-party scan insertion tool.

The command reads the scan information from the TSDB's *dft_inserted_designs* directory, specifically in a Tcl dictionary file named *<design_name>.dft_info_dictionary*.

This file can be sourced to any Tcl script engine. The file contains the information about the DFT inserted in the design that must be considered during scan insertion when using a third-party scan insertion tool and contains the following sections:

- `dft_signals` — Contains all of the DFT signals.
 - `modules_with_chains` — Contains all modules that already scanned, which means that they should be stitched into scan chains as sub-chains.
 - `non_scannable_instance_list` — Contains those instances set to non-scan during scan insertion.
 - `edt_instances` — Contains and describes the EDT modules that the scan changes connect to.
- The Tessent Shell environment provides a mechanism for generating an example usage script you can customize to work with your third-party scan insertion tool. In the Tessent Shell tool, you invoke the following commands:

```
read_verilog design_netlist
source \
../tsdb_outdir/dft_inserted_designs/
design_name.last_DFT_insertion_design_id/
design_name.dft_info_dictionary
get_dft_info_dictionary -example_usage_script
```

After issuing these commands, the tool creates an example usage script. Use this script as a starting example to convert the dictionary into the specific commands used by your

third-party scan insertion tool. In the file, the tool inserts pound signs (#) with comments that specify which actions your third-party tool must perform. For example:

```
puts "Processing DFT signals"
set dft_signals_dict [dict get $tessent_dft_info_dict dft_signals]
puts "Setting up Static Dft Signals"
foreach dft_signal [dict keys $dft_signals_dict] {
  if {[dict exists $dft_signals_dict \
    $dft_signal forced_value_in_pre_scan_drc]} {
    set connection_node_name \
      [dict get $dft_signals_dict $dft_signal connection_node_name]
    set connection_node_type \
      [dict get $dft_signals_dict $dft_signal connection_node_type]
    set forced_value_in_pre_scan_drc \
      [dict get $dft_signals_dict $dft_signal \
        forced_value_in_pre_scan_drc]
    puts "--- Static Dft Signal $dft_signal ---"
    if {$connection_node_type eq "pin"} {
      ### Command to cut and force created pseudo port
      add_primary_input [get_pins [list $connection_node_name]] \
        -internal
      add_input_constraints \
        [get_pins [list $connection_node_name]] \
        -c${forced_value_in_pre_scan_drc}
    } else {
      ### Command tp force port
      add_input_constraints \
        [get_ports [list $connection_node_name]] \
        -c${forced_value_in_pre_scan_drc}
    }
  }
}
...

```

- If required and after you have inserted the scan using your third-party tool, you can proceed to [“Make Pre-ATPG Connections with Third-Party Scan for Top Chip”](#) on page 209.
- When you have completed scan insertion, the next step is to [“Perform Top Chip Graybox Generation, ATPG Pattern Generation, and Wrapped Core Pattern Retargeting”](#) on page 211.

Make Pre-ATPG Connections with Third-Party Scan for Top Chip

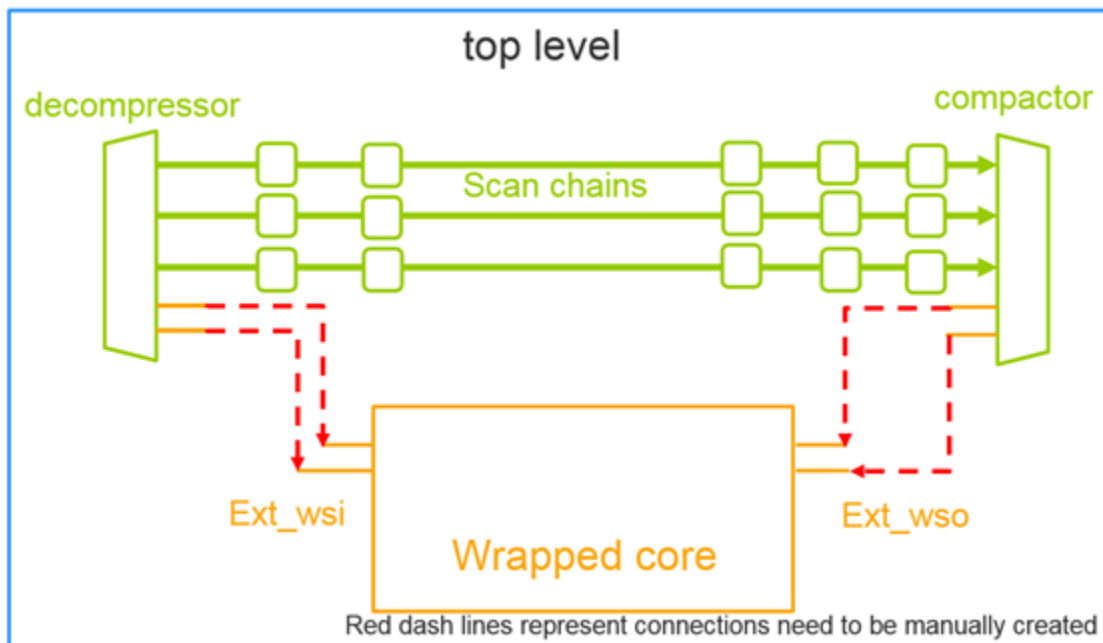
You perform this step in the flow only if the wrapped core has at least one sub-wrapped core.

If your design contains no sub-wrapped cores, skip this step in the flow and proceed directly to [“Perform Top Chip Graybox Generation, ATPG Pattern Generation, and Wrapped Core Pattern Retargeting”](#) on page 211.


The most important logic to support hierarchical ATPG is the connection from the top level EDT scan in ports to the primary inputs that you created for each wrapper chain to cover external faults to all child cores.

Figure 5-26 illustrates this connection at an abstract level, marked in red.

Figure 5-26. Top Level EDT Connection to Child Cores



Note

 The line numbers used in this procedure refer to the command flow dofile in “[Example Dofile for Pre-ATPG Connections for Top Chip](#)” on page 211.

Prerequisites

- You must have performed the two-pass DFT insertion as described in “[Perform Two-Pass Insertion and Synthesis for Top Chip](#)” on page 206.
- You must have inserted scan using your third-party scan insertion tool per the guidelines provided in “[Third-Party Scan Insertion for Top Chip](#)” on page 207.

Procedure

1. Load the design. (See lines 1-9).
2. Change to insertion mode. (See line 12).
3. Make the connections. (See lines 12-32).
4. Save the design. (See line 36).
5. Write the design back to the TSDB. (See lines 38-40).

Examples

Example Dofile for Pre-ATPG Connections for Top Chip

```
1 # load the design
2 set_context dft -no_rtl
3 set_tsdb_output_directory ../tsdb_outdir
4 open_tsdb \ ../../cores_will_be_wrapped/<child_core_name>/tsdb_outdir
5 read_cell_library ../../library/tessent/adk.tcelllib
6 read_design <top_level_name> -design_id <rtl2> -no_hdl
7 read_verilog ../../<synthesis_path>/<top_level_name>.vg
8 read_design <child_core_name> -design_id <scan> -view graybox
9 set_current_design <top_level_name>
10
11 # make the ATPG connections:
12 set_system_mode insertion
13
14 # disconnect empty scan out from ground
15 delete_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_out[20]
16 delete_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_out[21]
17 delete_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_out[22]
18 delete_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_out[23]
19 ...
20 # connect top level EDT scan in to child core primary input of wrapper
21 # chain
22 create_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_in[20]
23 /<child_core_instance_1>/ext_wsi[0]
24 create_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_in[21]
25 /<child_core_instance_1>/ext_wsi[1]
26 ...
27 # connect top level EDT scan out to child core primary output of wrapper
28 # chain
29 create_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_out[20]
30 /<child_core_instance_1>/ext_wso[0]
31 create_connection /chip_top_rtl2_tessent_edt_c1_inst/edt_scan_out[21]
32 /<child_core_instance_1>/ext_wso[1]
33 # repeat for all ext_wsi/wso of all child core instances
34
35 # save design after insertion
36 write_design -output_file <top_level_name>_scan.vg -replace
37
38 # Specify the design level before writing out a softlink of the design
39 set_design_level physical_block
40 write_design -tsdb -softlink_netlist -verbose
41 exit
```

Perform Top Chip Graybox Generation, ATPG Pattern Generation, and Wrapped Core Pattern Retargeting

This step in the flow creates the graybox model of the top chip, generates ATPG patterns, and retargets those patterns for each child core in the design.

Prerequisites

- You must have completed the [“Perform Two-Pass Insertion and Synthesis for Top Chip”](#) on page 206 step for the top chip.

- You must have inserted scan with your third-party scan insertion tool per the guidelines cited in [“Third-Party Scan Insertion for Top Chip”](#) on page 207 for the top chip and written the scan-inserted netlist back into the Tessent Shell Database.
- If required, make ATPG connections using the process outlined in [“Make Pre-ATPG Connections with Third-Party Scan for Top Chip”](#) on page 209 for the top chip.

Procedure

1. Perform top level ATPG. Refer to [“Performing Top-Level ATPG Pattern Retargeting”](#) on page 177.

See [“Top Level ATPG Pattern Generation Example”](#) on page 213 for details on top-level ATPG. Verification of these patterns is a must to ensure that the circuit functions.

See [“Pattern Retargeting of Each Child Core Example”](#) on page 214 for details on the retargeting of patterns for each child core.

Depending on tester channel availability on how many cores can be run in parallel, the internal mode ATPG patterns from each of the lower-level cores can be retargeted to the chip-level top.

2. Save the design and write the patterns to the TSDB using the [write_tsdb_data](#) command.

```
write_tsdb_data -replace
```

Examples

Top Level ATPG Pattern Generation Example

```
# Import design:
set_context pattern -scan -design_id <scan>
set_tsdb_output_directory ../tsdb_outdir

# this open_tsdb should import all tsdb of child cores
open_tsdb ../../wraped_cores/<child_core_name>/tsdb_outdir
read_cell_library ../../../../library/tessent/adk.tcelllib

# make sure to import all child cores in graybox view
read_design <child_core_name> -design_id <scan> -view graybox
read_design <top_level_name> -design_id <scan>
set_current_design <design_name>
set_design_level top

# Import core instances we want to active:
add_core_instances -module chip_top_gate2_tessent_occ_INCLK
add_core_instances -module chip_top_gate2_tessent_occ_pll_clock_0
add_core_instances -module chip_top_gate2_tessent_edt_cl

# Configure top level DFT signal values to edt_mode and all wrapped child
# core instances to external mode:
set_static_dft_signal_values edt_mode 1
set_static_dft_signal_values ltest_en 1
set_static_dft_signal_values tck_occ_en 1
set_static_dft_signal_values int_ltest_en 1

# configure child core by set_test_setup_icall
set_static_dft_signal_values ext_mode 1 -instance <child_core_name1>
set_static_dft_signal_values ext_ltest_en 1 -instance <child_core_name1>

# repeat for all child core instances

# generate and save pattern:
set_fault_type stuck
set_system_mode analysis
create_patterns
write_tsdb_data -replace

# write parallel testbench and serial testbench for simulation
write_patterns <top_level_name>_parallel.v -verilog -parallel
write_pattenrs <top_level_name>_serial.v -verilog -serial
# Repeat to generate transition patterns. Verification of these patterns
# are a must to ensure the circuit functions.
```

Pattern Retargeting of Each Child Core Example

```
# Import design:
set_context pattern -scan_retargeting
set_tsdb_output_directory ../tsdb_outdir

# this open_tsdb should import all tsdb of child cores
open_tsdb ../../wraped_cores/<child_core_name>/tsdb_outdir
read_cell_library ../../../../library/tessent/adk.tcelllib

# make sure to import all child cores except the one we are doing
# retargeting in graybox view
read_design <child_core_name> -design_id <scan> -view graybox
read_design <current_core_name> -design_id <scan> -view graybox
read_design <top_level_name> -design_id <scan>
set_current_design <top_level_name>
set_design_level top

# Read in the internal mode core description file of current child core
# by add_core_instance:
# -mode should match the mode while generating internal mode patterns
add_core_instances -instance <current_core_instance_name> \
  -core <current_core_name> -mode edt_int_stuck

# Configure top level DFT signal values to retargeting_mode for this exact
# child core:
set_current_mode retarget1_<current_child_core_name>_stuck
# configure top level to retargeting mode for current child core
set_static_dft_signal_values retargeting1_mode 1

# generate and save pattern:
# for transition fault ATPG, replace stuck with transition
set_fault_type stuck
import_clocks
set_system_mode analysis
create_patterns
write_tsdb_data -replace

# write parallel testbench and serial testbench for simulation
write_patterns <current_child_core_name>_parallel.v -verilog \
  -parallel
write_pattenrs <current_child_core_name>_serial.v -verilog \
  -serial
```

Tessent Shell Post-Layout Validation Flow

Performing physical place and route on pre-layout netlists results in post-layout netlists you must validate before you can proceed to tape-out.

This flow assumes that you are familiar with the pre-layout flows as described in “[Tessent Shell Flow for Flat Designs](#)” and “[Tessent Shell Flow for Hierarchical Designs](#),” especially as related to ATPG pattern generation.

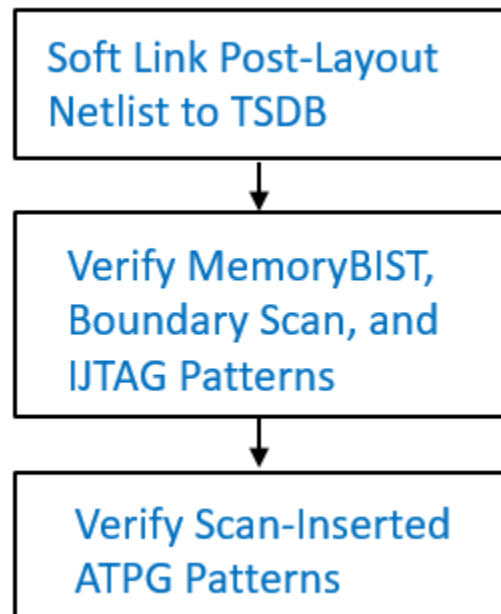
Overview of the Post-Layout Validation Flow	215
Soft Link TSDB and Post-Layout Netlist	216
Verify MemoryBIST, Boundary Scan and IJTAG Patterns	217
Verifying the Scan-Inserted ATPG Patterns.....	218
Post-Layout Validation When You Have Ungrouped IJTAG/OCC/EDT Logic.....	220

Overview of the Post-Layout Validation Flow

You can validate the post-layout netlist for hierarchical core, hierarchical top, and flat chip-level designs.

As shown in the following figure, you must first generate a soft link in the TSDB that points to the post-layout netlist. Then you verify the patterns for the MemoryBIST, boundary scan (if any) and IJTAG network, followed by verifying the post-scan-inserted ATPG patterns.

Figure 5-27. Post-Layout Validation Flow



This flow assumes that within the post-layout netlist, modules exist for the IJTAG network and inserted EDT and OCC instruments. That is, they remain distinct logical entities. Refer to “[Post-Layout Validation When You Have Ungrouped IJTAG/OCC/EDT Logic](#)” if you have ungrouped (unpreserved) logic.

Soft Link TSDB and Post-Layout Netlist

The soft link enables Tessent Shell to access the post-layout netlist for validation purposes. You are associating and linking the post-layout netlist (place-and-routed design) with all the pre-layout data files such as the ICL, TCD, and PDL.

Prerequisites

- You have previously performed the pre-layout flow so that you have the post-scan inserted DFT data files (ICL, PDL, TCD, and so on).
- You have a post-layout netlist as a result of place and route.

Procedure

1. Load the design, including the post-layout netlist (lines 1-15). Ensure that you specify a unique design ID, such as “post_layout”.
2. Add any black boxes, as applicable, and set the design level, which can be chip, physical_block, or sub_block (lines 17-22).
3. Save the updated netlist (line 23). Ensure that you use the -softlink_netlist option with the [write_design](#) command.

Using this switch references the post-layout netlist rather than copies it into the TSDB. This prevents duplication and enables the post-layout netlist to be updated without the need to repeat this step.

Examples

The following example generates an updated netlist for a wrapped core named processor_core that includes a soft link to the core’s post-layout netlist.

```
1 set_context patterns -ijtag -design_id post_layout
2
3 # Set the location of the TSDB
4 set_tsdb_output_directory ../tsdb_core
5
6 # Read the Tessent Cell Library
7 read_cell_library ../../../../library/tessent/adk.tcelllib
8
9 # Read the post-layout netlist and elaborate the design
10 read_verilog ../netlist/processor_core_layout.vg
11
12 # Read in the scan-inserted design data files generated during pre-layout
13 # The -no_hdl switch loads all relevant data except the original netlist
14 read_design processor_core -design_id gate -no_hdl -verbose
```



```
15
16 set_current_design processor_core
17
18 add_black_boxes -modules { \
19                     SYNC_1RW_8Kx16 \
20                     }
21
22 # Specify the design level before writing out the softlink
23 set_design_level physical_block
24 write_design -tsdb -softlink_netlist -verbose
25 exit
```

Verify MemoryBIST, Boundary Scan and IJTAG Patterns

Create and validate the MemoryBIST, boundary scan (if present), and IJTAG network patterns by creating and processing a patterns specification with the `create_patterns_specification` and `process_patterns_specification` commands.

Prerequisites

- You have generated a soft link in the TSDB that points to the post-layout netlist as described in “[Soft Link TSDB and Post-Layout Netlist](#).”

Procedure

1. Load the design by using `read_design` and pointing to the soft-linked post-layout netlist (lines 1-9).
2. If needed, add black boxes, and then check the design rules (lines 11-14).
3. Create, simulate, and check the test benches (see lines 16-24). Refer to “`create_patterns_specification`” and “`process_patterns_specification`” in the *Tessent Shell Reference Manual* for details.

Examples

Verify the Patterns With a New Patterns Specification

The following example verifies the patterns for a hierarchical core, `processor_core`.

```
1  set_context_patterns -ijtag -design_id post_layout
2  set_tsdb_output_directory ../tsdb_core
3
4  # Reading the tessent cell library
5  read_cell_library ../../../../library/tessent/adk.tcelllib
6
7  # Reading the soft-linked post-layout netlist from the tsdb_core
8  read_design processor_core -design_id post_layout -verbose
9  set_current_design processor_core
10
11 add_black_boxes -modules { \
12                     SYNC_1RW_8Kx16 \
13                     }
```

```
14 check_design_rules
15
16 create_patterns_specification
17 process_patterns_specification
18
19 set_simulation_library_sources -v ../../../../library/memory/
  SYNC_1RW_8Kx16.v
20 -v ../../../../library/verilog/adk.v
21
22 # Turn off clock monitoring when running simulation on post-layout netlist
23 run_testbench_simulations -simulation_macro_definitions
  TESSENT_DISABLE_CLOCK_MONITOR
24 check_testbench_simulations -report_status
25 exit
```

Verify the Patterns with a Customized Patterns Specification from Pre-Layout Signoff

You may have customized a patterns specification during pre-layout signoff. You can read in this patterns specification that was stored in the TSDB during pre-layout signoff. Use the `read_config_data` command instead of the `create_patterns_specification` command.

The following example shows how to read in a customized patterns specification from pre-layout netlist. The pre-layout patterns specification “processor_core_gate.patterns_spec_signoff” was used to create the patterns on the gate-level scan-inserted netlist.

```
set_context patterns -ijtag -design_id after_layout
set_tsdb_output_directory ../tsdb_core
# Reading the tessent cell library
# Read the soft-linked post-layout netlist and elaborate the design...
check_design_rules
read_config_data ../tsdb_core/patterns/ \
  processor_core_gate.patterns_spec_signoff
process_patterns_specification
# Read in the required libraries and simulate
...
```

Verifying the Scan-Inserted ATPG Patterns

Perform ATPG on the post-layout netlist and save the patterns.

Prerequisites

- In the SDC that was created during [ICL extraction](#) for the pre-layout DFT insertion flow, set the `tessent_get_preserve_instances` proc to `add_core_instances` when you do not need grayboxes. For example, when you are working with flat designs.

For wrapped cores, set the `tessent_get_preserve_instances` proc to `icl_extraction` to automatically include ICL instances in the grayboxes.

The SDC file is located in the TSDB directory under `dft_inserted_designs`. Refer to “[Timing Constraints \(SDC\)](#)” for more information.

Procedure

1. Load the design (lines 1-9). Ensure that you set the context to patterns -scan and read in the soft-linked post-layout netlist.
2. Set the current mode (lines 11-12). Specify a different name than that used during scan insertion and used during pre-layout pattern generation.
3. Perform the remainder of the ATPG pattern generation flow (see lines 14-35).

Examples

The following example shows how to verify scan-inserted ATPG patterns by using the patterns -scan context and a post-layout netlist. This example shows the flow for a flat design. For hierarchical designs, refer to “[Performing ATPG Pattern Generation: Wrapped Core](#)” for specifics related to wrapped cores and “[Top-Level ATPG Pattern Generation Example](#)” for a top-level example.

```
1  set_context patterns -scan
2  read_cell_library ../library/tessent/adk.tcelllib
3  read_cell_library ../library/mem_ver/memory.lib
4  # Point to the TSDB directory
5  set_tsdb_output_directory ../tsdb_rtl
6
7  # Reading the post-layout netlist
8  read_design cpu_top -design_id post_layout
9  set_current_design cpu_top
10
11 # Use a unique mode name
12 set_current_mode edt_stuck_final
13
14 report_dft_signals
15 # If Tessent Scan was used for scan insertion, can use the scan
    #configuration mode
16 import_scan_mode edt_mode
17 # Set the following DFT Signal values to use the boundary scan chain to
    #apply/capture values that would normally use the I/O pads
18 set_static_dft_signal_values int_ltest_en 1
19 set_static_dft_signal_values output_pad_disable 1
20 # Set the following DFT signal value to apply the shift_capture_clock to
    #the scan-tested network during capture phase of ATPG
21 set_static_dft_signal_value tck_occ_en 1
22 report_static_dft_signal_settings
23
24 set_system_mode analysis
25
26 # Generation of ATPG patterns
27 create_patterns
28 report_statistics -detailed_analysis
29 write_tsdb_data -replace
30
31 # Patterns written out for simulation
32 write_patterns patterns/cpu_top_stuck_parallel.v -verilog -parallel \
    -replace -scan -parameter_list {SIM_KEEP_PATH 1}
33 write_patterns patterns/cpu_top_stuck_serial.v -verilog -serial -replace \
    -parameter_list {SIM_KEEP_PATH 1}
```

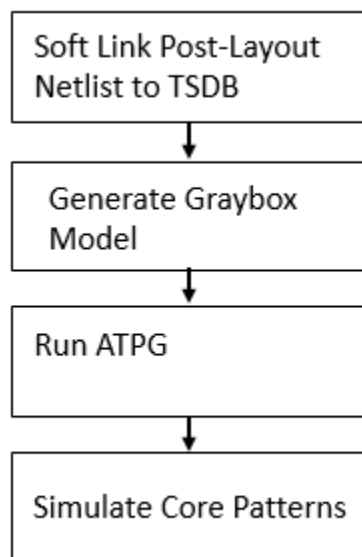
```
34 # Writing the STIL pattern for tester
35 write_patterns patterns/cpu_top_stuck.stil -stil -replace
36 exit
```

Post-Layout Validation When You Have Ungrouped IJTAG/OCC/EDT Logic

During layout, IJTAG, OCC, and EDT logic can become ungrouped, which means the hierarchy around this logic is dissolved so that the gates that were inside of the ungrouped instances become part of the next higher instance. Ungrouped test logic during layout can cause some of the automated setup for pattern generation to no longer operate seamlessly.

The automated flow relies on preservation of Tessent test logic's hierarchical names. Use the following post-layout validation flow to update the TSDB with the post-layout netlist and set up the design for ATPG. The flow assumes knowledge of the ATPG pattern generation flow for physical blocks as described in [“Performing ATPG Pattern Generation: Wrapped Core”](#).

Figure 5-28. Post-Layout Validation Flow with Ungrouped IJTAG/OCC/EDT Logic



The best way to avoid complications related to ungrouping in layout is to use the `tessent_get_preserve_instances` procedures in the generated SDC file to identify which instances must be preserved based on their intended uses. See the [“Synthesis Helper Procs”](#) section in the *Tessent Shell User's Manual*.

To use the `add_core_instances` command during ATPG or other post-layout steps, the hierarchy of the OCC and EDT logic must be preserved. The IJTAG logic nodes only need to be preserved if you plan to rerun ICL extraction on the post-layout netlist, but in most cases there is no need to do this.

Prerequisites

- If you do not preserve the instances, you must write out a TCD file for every mode of operation at the core level before you perform layout. In some cases, you may not need to generate patterns until after layout, but even then, you must run ATPG before layout to generate the core-level TCD files. Failure to perform this task could result in R14 or R15 rule check errors.

Procedure

1. Soft link the post-layout netlist to the TSDB. Refer to “[Soft Link TSDB and Post-Layout Netlist](#)” for more information.
2. Generate the graybox model.

Graybox models are only required for hierarchical cores, not for hierarchical top or flat designs. (The line numbers in this step refer to the example “[Generate the Graybox Model](#)” on page 222.)

- a. When loading the design (line 4), specify the same design that you used to write the post-layout netlist to the TSDB, for example “post_layout”.

Using the same design ID for the graybox model that you used for the post-layout netlist enables Tessent to access the full design view or the graybox model with the same design ID.

- b. Read the core description for external mode using the `add_core_instances` command (line 9).

Because you already ran the pre-layout ATPG step and saved the TCD file to the TSDB, the `add_core_instances` command can read the existing TCD file and add the core instances that are active in external mode.

- c. Use `analyze_graybox` to generate the graybox (line 13). If the IJTAG network was ungrouped in layout, the `analyze_graybox` command can automatically find and preserve the IJTAG SRI network as long as the default names of the logic have not changed.

3. If you are running in hierarchical mode, run ATPG on the core’s internal mode of the wrapped core to generate the ATPG patterns that you retarget at the top level of the chip. If you are running in hierarchical top or in flat mode, run ATPG.

The line numbers in this example refer to example “[Run ATPG on the Core’s Internal Mode](#)” on page 223.

- a. Specify a unique ATPG mode name with the `set_current_mode` command (line 9). Append the name with “_post_layout” or “_final” to clarify which mode to use for silicon diagnosis, and then set the current mode type to “internal”.
- b. Add the core instances using the design ID and mode from the pre-layout step (line 14).

This loads the TCD file that contains the information for the design's core instances. It is unnecessary to match these instances to the test logic instances that may have been ungrouped during layout.

- c. (Optional) use the `read_faults` command to merge the fault list from running external mode to find the total overall fault coverage of the wrapped core (line 36).

When you run ATPG in internal mode for transition patterns, you must do the following:

- Specify a unique name for the ATPG run with the `set_current_mode` command. For example, `edt_int_tdf_final`.
- Use the `add_core_instances` command to read the transition mode TCD. For example:

```
add_core_instance -current_design -design_id gate -mode
edt_int_transition
```

- Ensure that you set the correct fault type:

```
set_fault_type transition
```

- You can optimize the number of capture cycles used by the OCCs by specifying the optional `capture_window_size` parameter. The following command specifies a `capture_window_size` of 2:

```
set_core_instance_parameters -instrument_type occ
-parameter_values [list capture_window_size 2]
```

4. Run Verilog simulation of the core-level ATPG patterns.

Performing this task ensures that the patterns function as needed when they are retargeted at the parent level. For parallel load patterns, as specified by the `write_patterns -parallel` command, simulate all the patterns. For serial load patterns, a handful of patterns are sufficient; the run time for simulating gate-level serial load patterns is significant. The `set_pattern_filtering` command (line 27) is used to reduce the number of serial patterns saved for the simulation.

Examples

Generate the Graybox Model

The following example creates a graybox model for a post-layout processor_core design and saves the data under the same design ID.

```
1 set_context patterns -scan -design_id post_layout
2 set_tsdb_output_directory ../tsdb_outdir
3 read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
4 read_design_processor_core -design_id post_layout -verbose
5 read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v -interface_only
6 set_current_design processor_core
7 # Use the add_core_instances command to read the TCD file.
8 #import_scan_mode ext_mode
9 add_core_instances -current_design -design_id gate
```

```

10 check_design_rules
11 report_scan_cells
12 # Create and write the updated graybox model to the TSDB.
13 analyze_graybox
14 write_design -tsdb -graybox -verbose
15 exit

```

Run ATPG on the Core's Internal Mode

```

1  set_context patterns -scan -design_id post_layout
2  set_tsdb_output_directory ../tsdb_outdirRun/adk.tcelllib
3  read_design processor_core -design_id post_layout -verbose
4  read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v -interface_only
5  set_current_design processor_core
6  # Specify the current mode using a different name than what was used
7  # during scan insertion or pre-layout
8
9  set_current_mode edt_int_stuck_final -type internal
10
11 # Use the -design_id and -mode from pre-layout to read in the TCD of that
12 # mode
13
14 add_core_instance -current_design -design_id gate -mode edt_int_stuck
15 set_system_mode analysis
16 add_fault -all
17 report_statistics -detail
18 create_patterns
19 report_statistics -detail
20 # Store TCD, flat_model, fault list and patDB format files in the TSDB
21 # directory
22
23 write_tsdb_data -replace
24
25 # Write Verilog patterns for simulation
26 write_patterns patterns/processor_core_stuck_parallel.v -verilog -
  parallel -replace -parameter_list {SIM_KEEP_PATH 1}
27 set_pattern_filtering -sample_per_type 2
28 write_patterns patterns/processor_core_stuck_serial.v -verilog -serial -
  replace -parameter_list {SIM_KEEP_PATH 1}
29 # Optional Step - Can run in external mode and calculate the fault
30 # coverage of the core(both Internal and External) as described below.
31 # In order to understand the coverage of the faults testable by Internal
32 # mode it is necessary to eliminate the undetected faults that would
33 # otherwise be detected in External mode. This is done by merging the
34 # fault list from running the graybox in External mode with read_faults:
35
36 #read_faults -mode ext_multi_stuck -fault_type stuck -merge -verbose
37
38 # Final coverage of the core that includes both Internal and External
39 # modes
40 # report_statistics -detail
41 exit

```

Test Bench Generation and Simulation in RTL Mode

Test bench module generation and the Standard Test Interface (STI) infrastructure support complex port data types. A simulation wrapper is generated to connect a DUT with complex pins to the test bench environment.

The simulation wrapper port list includes ports of scalar and one-dimensional bus data types. The port names are the post-synthesis names of sub-bundle objects as inferred from the original RTL complex ports of the DUT. This approach isolates the DUT from the test bench environment and hides the complexity of its ports and other SystemVerilog dependencies.

Simulation Wrapper Creation	224
Test Bench Examples	227

Simulation Wrapper Creation

A simulation wrapper is created during `process_dft_specification` and `extract_icl`. This wrapper is needed only if the current design module includes complex ports.

You can also create the simulation wrapper manually with the `get_current_design`, `report_current_design`, and `write_design` commands.

If `read_design` encounters ports that can cause issues in simulation or in the TSDB flow, a warning is issued:

```
// Warning: The design 'sv_mod1' just read has ports with 'unpacked struct'/'enumerate'  
// datatypes declared in Global ($unit) or local scope ('sv_mod1' scope).  
// If you try to elaborate this design within its parent, you will get an error.  
// It will only work if this design is your current design.
```

These issues typically arise when port declarations use data types that are not visible to the tool. The warning indicates that a call to `set_current_design` of the parent block will raise additional warnings. Use `report_current_design` after running `set_current_design` to obtain more information about these ports.

The following example shows how the simulation wrapper is created during `extract_icl`:

```
// command: extract_icl
// Writing simulation wrapper : \
./tsdb_outdir/dft_inserted_designs/sv_mod1_RTL1.dft_inserted_design/
sv_mod1.sv09_simulation_wrapper
// Note: Updating the hierarchical data model to reflect RTL design
changes.
// Writing design source dictionary : \
./tsdb_outdir/dft_inserted_designs/
sv_mod1_RTL1.dft_inserted_design/sv_mod1.design_source_dictionary
// Flattening process completed, cell instances=111, gates=1310, PIs=470,
POs=166, CPU time=0.04 sec.
// -----
// Begin circuit learning analyses.
// -----
// Learning completed, CPU time=0.02 sec.
// -----
// Begin ICL extraction.
// -----
// ICL extraction completed, ICL instances=10, CPU time=0.16 sec.
// -----
// -----
// Begin ICL elaboration and checking.
// -----
// ICL elaboration completed, CPU time=0.09 sec.
// -----
// Writing ICL file : ./tsdb_outdir/dft_inserted_designs/
sv_mod1_RTL1.dft_inserted_design/sv_mod1.icl
// Writing consolidated PDL file: ./tsdb_outdir/dft_inserted_designs/
sv_mod1_RTL1.dft_inserted_design/sv_mod1.pdl
// Writing SDC file: ./tsdb_outdir/dft_inserted_designs/
sv_mod1_RTL1.dft_inserted_design/sv_mod1.sdc
// Writing DFT info dictionary: ./tsdb_outdir/dft_inserted_designs/
sv_mod1_RTL1.dft_inserted_design/
sv_mod1.dft_info_dictionary
```

The following example shows how the simulation wrapper is created during process_dft_specification:

```
// command: process_dft_specification
//
// Begin processing of /DftSpecification(sv_mod1,RTL2)
// --- IP generation phase ---
// Validation of IjtagNetwork
// Validation of OCC
// Validation of EDT
// Processing of IjtagNetwork
// Generating design files for IJTAG SIB module
sv_mod1_RTL2_tessent_sib_1
// Verilog RTL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_ijtag.instrument/sv_mod1_RTL2_tessent_sib_1.v
// IJTAG ICL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_ijtag.instrument/sv_mod1_RTL2_tessent_sib_1.icl
// Generating design files for IJTAG SIB module
sv_mod1_RTL2_tessent_sib_2
// Verilog RTL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_ijtag.instrument/sv_mod1_RTL2_tessent_sib_2.v
// IJTAG ICL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_ijtag.instrument/sv_mod1_RTL2_tessent_sib_2.icl
// Generating design files for IJTAG Tdr module
sv_mod1_RTL2_tessent_tdr_sri_ctrl
// Verilog RTL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_ijtag.instrument/sv_mod1_RTL2_tessent_tdr_sri_ctrl.v
// IJTAG ICL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_ijtag.instrument/sv_mod1_RTL2_tessent_tdr_sri_ctrl.icl
//
// Loading the generated RTL verilog files (2) to enable instantiating
the contained modules
// into the design.
// Processing of EDT
// Generating design files for EDT module
sv_mod1_RTL2_tessent_edt_edt1
// Verilog RTL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_edt.instrument/sv_mod1_RTL2_tessent_edt_edt1.v
// IJTAG ICL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_edt.instrument/sv_mod1_RTL2_tessent_edt_edt1.icl
// IJTAG PDL : ./tsdb_outdir/instruments/
sv_mod1_RTL2_edt.instrument/sv_mod1_RTL2_tessent_edt_edt1.pdl
// TCD : ./tsdb_outdir/instruments/
sv_mod1_RTL2_edt.instrument/sv_mod1_RTL2_tessent_edt_edt1.tcd
//
// Loading the generated RTL verilog files (1) to enable instantiating
the contained modules
// into the design.
// --- Instrument insertion phase ---
// Inserting instruments of type 'ijtag'
// Inserting instruments of type 'edt'
//
// Writing out modified source design in ./tsdb_outdir/
dft_inserted_designs/sv_mod1_RTL2.dft_inserted_design
// Writing simulation wrapper : ./tsdb_outdir/dft_inserted_designs/
sv_mod1_RTL2.dft_inserted_design/sv_mod1.sv09_simulation_wrapper
// Writing out specification in ./tsdb_outdir/dft_inserted_designs/
```

```
sv_mod1_RTL2.dft_spec
// Done processing of
DftSpecification(sv_mod1,RTL2
```

Related Topics

[get_current_design](#)

[read_design](#)

[report_current_design](#)

[write_design](#)

[set_current_design](#)

Test Bench Examples

SystemVerilog design definition with the top level named “top” and one SystemVerilog interface port named “Bus”:

```
typedef enum logic [2:0] {
    RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW
} color_t;

typedef struct packed{
    logic [2:0] eventbus;
} event_bus_packet_t;

typedef struct packed{
    color_t state_encoded;
    event_bus_packet_t dt_lcp_pulse;
    logic [3:0] eventbus_encoded;
} event_bus_split_packet_t;

interface MSBus;
    event_bus_split_packet_t Addr;
    logic [1:0] Data;
    logic RWn;
    logic Clk;
    modport Secondary (input Addr, RWn, Clk, output Data);
endinterface

module top(MSBus.Secondary Bus);
endmodule
```

Each bit-blasted bit of the DUT is connected to its associated test pattern wire using the post-synthesis pin name. The name of the test pattern bit is the same as that of the DUT pin bit, escaped with a backslash (\). The DUT instance (DUT_inst) is the generated simulation

wrapper, with the DUT itself instantiated in it. This table cross-references the DUT pin names and the test pattern bit names:

Table 5-1. Test Pattern Bit Name Assignments

DUT Pin of Port “Bus”	Test Pattern Bits
DUT_inst\Bus.Addr [10]	\Bus.Addr [10]
DUT_inst\Bus.Addr [9]	\Bus.Addr [9]
DUT_inst\Bus.Addr [8]	\Bus.Addr [8]
DUT_inst\Bus.Addr [7]	\Bus.Addr [7]
DUT_inst\Bus.Addr [6]	\Bus.Addr [6]
DUT_inst\Bus.Addr [5]	\Bus.Addr [5]
DUT_inst\Bus.Addr [4]	\Bus.Addr [4]
DUT_inst\Bus.Addr [3]	\Bus.Addr [3]
DUT_inst\Bus.Addr [2]	\Bus.Addr [2]
DUT_inst\Bus.Addr [1]	\Bus.Addr [1]
DUT_inst\Bus.Addr [0]	\Bus.Addr [0]
DUT_inst\Bus.RWn	\Bus.RWn
DUT_inst\Bus.Clk	\Bus.Clk
DUT_inst\Bus.Data [1]	\Bus.Data[1]
DUT_inst\Bus.Data [0]	\Bus.Data[0]

The test bench module generated is as follows, in part:

```
`timescale 1ns / 1ns
module TB;
  integer      _write_DIAG_file;
  integer      _DIAG_file_header;
  integer      _diag_file;
  integer      _diag_chain_header;
  integer      _diag_scan_header;
  integer      _last_fail_pattern;
  . . .
  wire \Bus.Addr[10], \Bus.Addr[9], \Bus.Addr[8], \Bus.Addr[7],
      \Bus.Addr[6], \Bus.Addr[5], \Bus.Addr[4], \Bus.Addr[3],
      \Bus.Addr[2], \Bus.Addr[1], \Bus.Addr[0], \Bus.RWn ,
      \Bus.Clk , \Bus.Data[1], \Bus.Data[0], ijtag_tck,
      ijtag_reset, ijtag_ce, ijtag_se, ijtag_ue, ijtag_sel,
      ijtag_si, ijtag_so;
  assign \Bus.Addr[10] = _ibus[19];
  assign \Bus.Addr[9] = _ibus[18];
  assign \Bus.Addr[8] = _ibus[17];
  assign \Bus.Addr[7] = _ibus[16];
  assign \Bus.Addr[6] = _ibus[15];
  assign \Bus.Addr[5] = _ibus[14];
  assign \Bus.Addr[4] = _ibus[13];
  assign \Bus.Addr[3] = _ibus[12];
  assign \Bus.Addr[2] = _ibus[11];
  assign \Bus.Addr[1] = _ibus[10];
  assign \Bus.Addr[0] = _ibus[9];
  assign \Bus.RWn = _ibus[8];
  assign \Bus.Clk = _ibus[7];
  assign ijtag_tck = _ibus[6];
  assign ijtag_reset = _ibus[5];
  assign ijtag_ce = _ibus[4];
  assign ijtag_se = _ibus[3];
  assign ijtag_ue = _ibus[2];
  assign ijtag_sel = _ibus[1];
  assign ijtag_si = _ibus[0];
  assign _sim_obus[2] = \Bus.Data[1] ;
  assign _sim_obus[1] = \Bus.Data[0] ;
  assign _sim_obus[0] = ijtag_so;
  . . .
  reg[71:0] mem [0:1864134];
  top_wrapper DUT_inst (. \Bus.Addr ( {\Bus.Addr[10], \Bus.Addr[9],
      \Bus.Addr[8], \Bus.Addr[7],
      \Bus.Addr[6], \Bus.Addr[5],
      \Bus.Addr[4], \Bus.Addr[3],
      \Bus.Addr[2], \Bus.Addr[1],
      \Bus.Addr[0] } ),
      . \Bus.RWn ( . \Bus.RWn ),
      . \Bus.Clk ( . \Bus.Clk ),
      . \Bus.Data ( { \Bus.Data[1], \Bus.Data[0] },
      .ijtag_tck(ijtag_tck),
      .ijtag_reset(ijtag_reset),
      .ijtag_ce(ijtag_ce),
      .ijtag_se(ijtag_se),
      .ijtag_ue(ijtag_ue),
      .ijtag_sel(ijtag_sel),
```

```
        .ijtag_si(ijtag_si),  
        .ijtag_so(ijtag_so));  
    . . .  
endmodule
```

Simulation wrapper:

```
module top_wrapper(input wire [10:0] \Bus.Addr ,  
                  input wire \Bus.RWn ,  
                  input wire \Bus.Clk ,  
                  output wire [1:0] \Bus.Data  
                  input wire ijtag_tck,  
                  input wire ijtag_reset,  
                  input wire ijtag_ce,  
                  input wire jtag_se,  
                  input wire ijtag_ue,  
                  input wire ijtag_sel,  
                  input wire ijtag_si,  
                  output wire ijtag_so);  
  
    MSBus tessent_intf_inst1();  
  
    assign tessent_intf_inst1.Addr = \Bus.Addr ;  
    assign tessent_intf_inst1.RWn = \Bus.RWn ;  
    assign tessent_intf_inst1.Clk = \Bus.Clk ;  
    assign \Bus.Data = tessent_intf_inst1.Data;  
  
    top_current_design(.Bus(tessent_intf_inst1),  
                      .ijtag_tck(ijtag_tck),  
                      .ijtag_reset(ijtag_reset),  
                      .ijtag_ce(ijtag_ce),  
                      .ijtag_se(ijtag_se),  
                      .ijtag_ue(ijtag_ue),  
                      .ijtag_sel(ijtag_sel),  
                      .ijtag_si(ijtag_si),  
                      .ijtag_so(ijtag_so));  
  
endmodule
```

The test bench dofile:

```
set_context dft -rtl -design_id RTL1
read_cell_library techlib_adk.tnt/current/tessent/adk.tcelllib
read_core_description design/mem/*.tcd_memory
read_verilog design/packages/types.sv -format sv2009
read_verilog -f design/sv.list -format sv2009
set_current_design top
set_design_level phys
set_dft_specification_requirements -memory_test on
add_clocks [get_ports Bus.Clk] -period 10ns
check_design_rules
set_spec [create_dft_spec]
// (the spec is populated here)
process_dft_spec
extract_icl
create_pattern_spec
process_pattern_spec
set_simulation_library_sources -v techlib_adk.tnt/current/verilog/adk.v
                                -logical_library_map_list memlib
                                ./memlib -v design/sv_v_files/pll.v
run_testbench_simulation -simulator_options {-t 10ps -L
memlib}
```

Hybrid TK/LBIST Flow for Flat Designs

Tessent Shell supports inserting LogicBIST controllers that share IP resources with EDT controllers to reduce hardware overhead. The process for performing this task is known as the hybrid TK/LBIST flow. In this flow, Tessent Shell automatically adds hybrid logic to the EDT controllers for IP resource sharing. The shared LogicBIST and EDT IP is often referred to as hybrid TK/LBIST IPs, or simply hybrid IPs.

This discussion builds on the Tessent Shell RTL and scan DFT insertion flow as described in “[Tessent Shell Flow for Flat Designs](#)” on page 112. Refer to *Hybrid TK/LBIST Flow User’s Manual* for details about the hybrid TK/LBIST flow and inserted architecture.

Tip

 This flow increments the basic Tessent Shell RTL and scan DFT insertion flow. To aid comprehension, ensure that you have reviewed the test case for flat designs.

Refer to the following test case for a detailed usage example of the flow described in this section. This test case illustrates hybrid IP insertion into a flat design with MemoryBIST in the first DFT insertion pass and EDT, OCC, and LogicBIST in the second DFT insertion pass.

```
tessent_example_hybrid_tk_lbist_flow_<software_version>.lbist
```

You can access this test case by navigating to the following directory:

```
<software_release_tree>/share/UsageExamples/
```

RTL and Scan DFT Insertion Flow With Hybrid TK/LBIST.....	232
Perform the First DFT Insertion Pass: Hybrid TK/LBIST.....	233
Perform the Second DFT Insertion Pass: Hybrid TK/LBIST.....	236
Perform Test Point Insertion	241
Perform Scan Chain Insertion (Hybrid Flow)	244
Performing ATPG Pattern Generation: Hybrid TK/LBIST.....	246
Performing LogicBIST Fault Simulation	249
Perform LogicBIST Pattern Generation.....	251

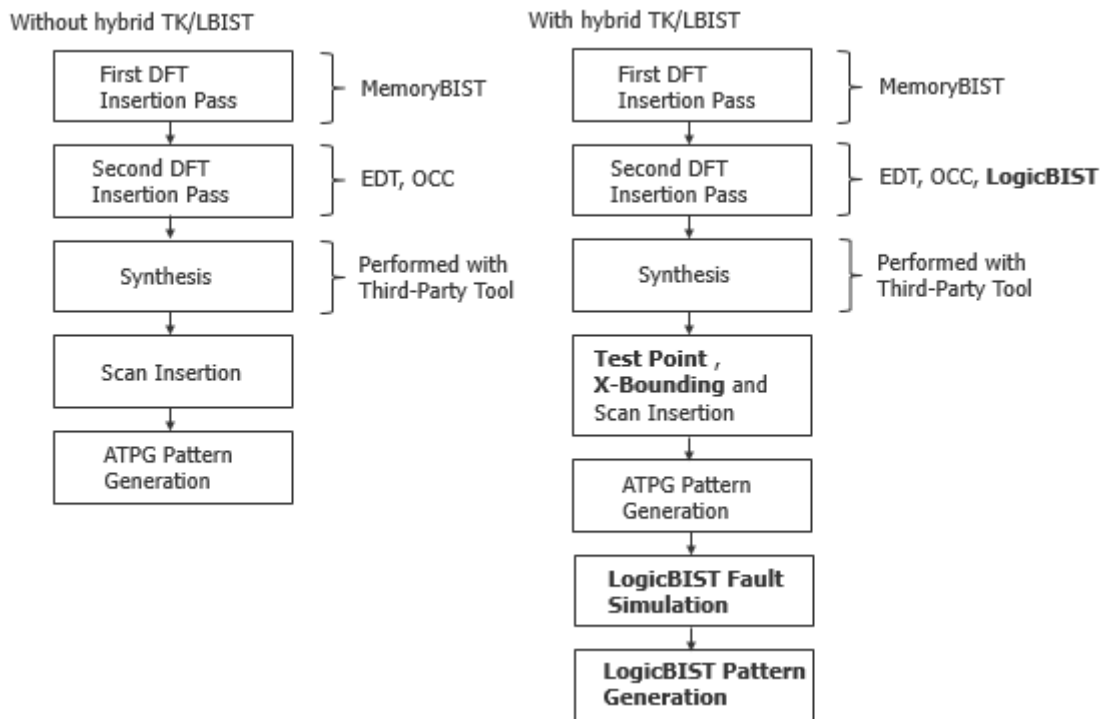
RTL and Scan DFT Insertion Flow With Hybrid TK/LBIST

Within the two-pass DFT insertion process for a flat design, insert the hybrid TestKompress/LogicBIST IP during the second DFT insertion pass. Adding LogicBIST to your Tessent Shell flow introduces several new steps: X-bounding, test point insertion, and LogicBIST fault simulation and pattern generation. This hybrid solution adds self-test capabilities and is key to satisfying the reliability requirements of the ISO 26262 automotive safety standard.

Note

The test case that illustrates the hybrid TK/LBIST flow does not include boundary scan insertion. Refer to “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” on page 116 for information about inserting boundary scan in the first DFT insertion pass.

Figure 5-29. Two-Pass Insertion Flow With Hybrid TK/LBIST




Perform the First DFT Insertion Pass: Hybrid TK/LBIST

When performing the hybrid TK/LBIST flow to insert the hybrid IP, perform the first pass of the two-pass DFT and scan insertion process as usual—insert MemoryBIST and boundary scan. In addition, for the hybrid TK/LBIST flow, enable controller chain mode (CCM) test by specifying a dedicated control signal.

CCM enables you to generate ATPG patterns that target the hybrid-IP logic so that you can test the test logic. Refer to “[Controller Chain Mode](#)” in the *Hybrid TK/LBIST User’s Manual* for details.

Refer to “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” on page 116 for additional information.


Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-13](#) on page 234.

Procedure

1. Load the RTL design data and set the design parameters. (See lines 1-11.)

Note

 “rtl1” is the recommended naming convention for the design ID for the first insertion pass, but you can specify any name. Refer to “[Loading the Design](#)” for more information about setting the design ID. The default design ID for the current test case is “rtl.”

2. Add a DFT signal to use for controller chain mode. (See lines 13-16.) You must register the DFT signal name and then add the signal.

By default, Tessent Shell adds a pin at the current design level to control CCM. You can save this pin by using a TDR register to control CCM; do this by specifying the `add_dft_signal -create_with_tdr` switch.

Whether or not you are inserting MemoryBIST or boundary scan, you must add the CCM DFT signal in the first DFT insertion pass because its generated hardware is used in the second pass when you insert the hybrid IP.

3. Set the `set_dft_specification_requirements` command to “on” for `-memory_bist`. (See lines 18-19.)
4. Define the design clocks. (See lines 21-23.)
5. Check the design rules and set the system mode to analysis. (See lines 25-26.)
6. Create the DFT specification. (See lines 28-31.)
7. Generate the DFT hardware, IJTAG network connectivity, and test patterns. (See lines 33-43.)
8. Run simulations to verify the design. (See lines 45-50.)

Examples

Example 1

The following dofile example shows a typical command flow as detailed in the preceding procedure. The highlighted command lines are unique to the process for inserting hybrid IP in a two-pass DFT insertion process.

Example 5-13. Dofile Example for First Pass, Hybrid TK/LBIST With MemoryBIST

```
1 # Load the design
```

```

2 set_context dft -rtl
3 set_tsdb_output_directory ../tsdb_outdir
4 read_verilog ../rtl/piccpu_rtl.v
5 read_cell_library ../lib/tessent/adk.tcelllib \
6   ../lib/tessent/picdram.atpglib
7 set_design_source -format tcd_memory -y ../rtl -extension memlib
8
9 set_current_design piccpu
10
11 set_design_level physical_block
12
13 # Add DFT signal for controller chain mode test
14 add_dft_signal_controller_chain_mode -create_with_tdr
15
16 # Specify the DFT requirements
17 set_dft_specification_requirements -memory_test on
18
19 # Define memory clock
20 add_clocks 0 ramclk -period 100ns
21 add_clocks 0 clk -period 100ns
22
23 check_design_rules
24 set_system_mode analysis
25
26 # Create and report the DFT specification
27 # This test case reads in the DFT specification from a separate file
28 read_config_data mbist_ip.dftspec
29 report_config_data
30
31 # Generate and insert the hardware
32 process_dft_specification
33
34 extract_icl
35
36 # Generate MemoryBIST and IJTAG network verification patterns
37 create_patterns_specification
38
39 report_config_data
40
41 process_patterns_specification
42
43 # Run and check test bench simulations
44 set_simulation_library_sources -v {../lib/verilog/adk.v} \
45   -y ../lib/verilog -extension v
46
47 run_testbench_simulations
48 check_testbench_simulations -report_status
49
50 exit

```

Example 2

Typically, you insert MemoryBIST or boundary scan in the first DFT insertion pass before inserting the hybrid IP (and OCC) in the second DFT insertion pass. The following sample dofile shows a first DFT insertion pass without MemoryBIST or boundary scan, in which you are adding the DFT signal for controller chain mode test.

Example 5-14. Dofile Example for First DFT Pass, Hybrid TK/LBIST Only

```

# Load the design
set_context dft -rtl -design_id rtl1
set_tsdb_output_directory ../tsdb_outdir
set_design_sources -format tcd_memory -y ../rtl -extension memlib
read_verilog ../rtl/piccpu_rtl.v -verbose
read_cell_library ../library/tessent/adk.tcelllib \
  ../rtl/picdram.atpglib
set_current_design piccpu
set_design_level physical_block

# Specify the clocks
add_clocks 0 clk -period 100ns
add_clocks 0 ramclk -period 100ns

# Add DFT signal for controller chain mode test
add_dft_signal controller_chain_mode -create_with_tdr

check_design_rules

# Create DFT specification
set_spec [create_dft_specification]
set_config_value -in $spec user_rtl_cells on

process_dft_specification

extract_icl

# Generate patterns
create_patterns_specification

# Create patterns specification and create simulation test benches
process_patterns_specification

# Run and check testbench simulations
set_simulation_library_sources -v ../library/verilog/adk.v \
  -v ../library/memory/ram.v -y ../library/verilog -extensionv
run_testbench_simulations -simulator_option +nowarnTSCALEexit

```

Perform the Second DFT Insertion Pass: Hybrid TK/LBIST

In the second DFT insertion pass with the hybrid TK/LBIST flow, insert the EDT, OCC, and LogicBIST instruments.

Note

The line numbers used in this procedure refer to the command flow dofile in “[Example 1: Dofile Flow for the Second DFT Insertion Pass: Hybrid TK/LBIST](#)” on page 238 unless otherwise noted.

Procedure

1. Load the design and set the environment (see lines 1-6). Refer to “[Loading the Design](#)” on page 121 for more information.
2. Define the DFT signals (see lines 8-28). Refer to “[Specifying and Verifying the DFT Requirements](#)” on page 123 for more information.

Note

 Hybrid TK/LBIST has additional DFT requirements for the clock signals.

3. Create the LogicBIST test point and X-bounding signals with a TDR (see lines 23-24).
4. Add a core instance for the MemoryBIST mini-OCC for LogicBIST test. (See lines 30-31.)

```
add_core_instances -instances [get_instances
*_tessent_sib_sti_inst]
```

This is required when you have inserted MemoryBIST in the first DFT insertion pass.

5. [Creating the DFT Specification](#). (See lines 38-50.) Include a SIB for LogicBIST by specifying lbist in the create_dft_specification command.

```
create_dft_specification -sri_sib_list {edt occ lbist}
```

Customize the generated DFT specification as follows. See “[Example 2: DFT Specification Example for Second DFT Insertion Pass with Hybrid TK/LBIST](#)” on page 239.

- a. For OCC, set the static clock control to external and the capture trigger to capture_en.

When static_clock_control is set to external, the OCC has an N-bit input (where N equals the number of shift register bits), which is driven by the NcpIndexDecoder. This is what gives the OCC programmability for LogicBIST.

When capture_trigger is set to capture_en, the OCC is capable of handling a free-running slow clock. In LogicBIST applications, the slow clock is a free-running clock (whereas for ATPG this comes from a top-level tester-controlled clock).

For details about OCC for the hybrid TK/LBIST flow, refer to “[Tessent OCC TK/LBIST Flow](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

- b. Specify a [LogicBist](#) wrapper that contains both a Controller wrapper and an NcpIndexDecoder wrapper.

This wrapper causes Tessent Shell to automatically convert the EDT controller into a hybrid controller for the hybrid TK/LBIST flow. In the Controller/Connections wrapper, you must specify the controller_chain_enable property if you are using a TDR to control CCM. Specify the full path to the pin or port name.

The NcpIndexDecoder wrapper specifies the named capture procedures (NCPs) for LogicBIST test. For details, refer to “[NCP Index Decoder](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

- c. In the EDT/Controller wrapper, define the [LogicBistOptions](#) if you are not using the default values.

When you specify the LogicBIST wrapper, the tool adds a LogicBistOptions wrapper to the EDT controller automatically populated with default values.

The misr_input_ratio property provides a way to specify the MISR size. The automatic (default) ratio results in the lowest hardware with a small MISR.

You can control how many chains are masked by a single mask register bit using the chain_mask_register_ratio property. By default, the ratio is 1:1; there are as many bits in the chain mask register as there are number of scan chains.

Specify the low-power shift options specifically for hybrid IP usage separately from the low-power shift options for the EDT controller.

6. Process the DFT specification to generate the test hardware, including LogicBIST (see line 52). Refer to “[Generating the EDT, Hybrid TK/LBIST, and OCC Hardware](#)” on page 130 for more information.
7. Extract the ICL information (see line 54). Refer to “[Extracting the ICL Module Description](#)” on page 130 for more information.
8. Generate the patterns and simulate the test bench (see lines 56-63). Refer to “[Generating ICL Patterns and Running Simulation](#)” on page 131 for more information.

Examples

Example 1: Dofile Flow for the Second DFT Insertion Pass: Hybrid TK/LBIST

The following dofile example shows a typical command flow. The highlighted command lines are unique to the process for inserting Tessent Shell LogicBIST and hybrid IP instruments in a two-pass DFT insertion process. In this example, the dofile reads in the DFT specification that defines the instruments to be inserted.

```

1  set_context dft -rtl -design_id rtl2
2  set_tsdb_output_directory ../tsdb_outdir
3  read_design piccpu -design_id rtl1
4  read_cell_library ../lib/tessent/adk.tcelllib \
5    ../lib/tessent/picdram.atpglib
6  set_current_design piccpu
7
8  # Add the DFT signals
9  # For scan test MemoryBIST-related logic
10 # Define tck_occ_en to access mini-OCC during LogicBIST
11 add_dft_signal ltest_en memory_bypass_en tck_occ_en
12
13 # Scan test shift/capture and edt clocks are driven from the
14 # test clock (saves a port)
15 add_dft_signal scan_en edt_update -source_node {scan_en edt_update }
```

```

16 add_dft_signal test_clock -source_node test_clock
17 add_dft_signal edt_clock shift_capture_clock -create_from_other_signals
18 # Alternatively, they can be directly driven from a primary port as
19 # follows
20 # add_dft_signal edt_clock shift_capture_clock -source_node {edt_clock
21 # shift_capture_clock}
22
23 # Required DFT signals for hybrid TK/LBIST
24 add_dft_signals control_test_point_en observe_test_point_en X_bounding_en
25
26 add_dft_signals int_ltest_en ext_ltest_en
27
28 set_dft_specification_requirements -logic_test on
29
30 # Add the MemoryBIST mini-OCC for LogicBIST test
31 add_core_instances -instances [get_instances *_tessent_sib_sti_inst]
32
33 # add_clock clk -period 100ns
34 # Need to define it because of ICL clock port tracing
35
36 check_design_rules
37
38 # Create DFT specification
39 # Populated later in this dofile
40 set_spec [create_dft_specification -sri_sib_list {occ edt lbist } ]
41
42 # You can set this property if there are no library cells
43 # set_config_value -in $spec use_rtl_cells on
44
45 report_config_data $spec
46
47 # Read in the DFT specification data
48 # This test case reads in the DFT specification from a separate file
49 # This file is illustrated in the next example
50 read_config_data logic_instruments.dftspec -in_wrapper $spec -replace
51
52 process_dft_specification
53
54 extract_icl
55
56 create_patterns_specification
57 process_patterns_specification
58
59 set_simulation_library_sources -v {../lib/verilog/adk.v} \
60     -y ../lib/verilog -extension v
61
62 run_testbench_simulations
63 check_testbench_simulations -report_status
64
65 exit

```

Example 2: DFT Specification Example for Second DFT Insertion Pass with Hybrid TK/LBIST

The following example illustrates a DFT specification customized to include the wrappers for the LogicBIST controller and additional LogicBIST options for the EDT controller.

```

1 Occ {

```

```

2  ihtag_host_interface: Sib(occ);
3  capture_trigger: capture_en;
4  static_clock_control: both;
5  Controller(clk) {
6    clock_intercept_node: clk;
7  }
8  Controller(ramclk) {
9    clock_intercept_node: ramclk;
10 }
11 }
12
13 # Include the LogicBIST controller
14 # Example LogicBIST only; modify for your design requirements
15 LogicBist {
16   ihtag_host_interface : Sib(lbist);
17   Controller(c0) {
18     burn_in : on ;
19     pre_post_shift_dead_cycles : 8 ;
20     SingleChainForDiagnosis { Present : on ; }
21     ShiftCycles {max: 40; hardware_default : 1024;}
22     CaptureCycles {max: 4;}
23 # Hardware default is max
24     PatternCount {max: 10000; hardware_default : 1024;}
25 # Hardware default is 0
26     WarmupPatternCount { max : 512;}
27     ControllerChain {
28       present : on;
29       clock : tck;
30     }
31     Connections {
32       shift_clock_src: clk;
33       controller_chain_enable : piccpu_rtl_tessent_tdr_sri_ctrl_inst/ \
34         controller_chain_mode;
35     }
36   }
37   NcpIndexDecoder{
38     Ncp(clk_occ_ncp) {
39       cycle(0): OCC(clk);
40       cycle(1): OCC(clk);
41     }
42     Ncp(ramclk_occ_ncp) {
43       cycle(0): OCC(ramclk);
44       cycle(1): OCC(ramclk);
45     }
46 # References to piccpu_rtl_tessent_sib_1 only applicable when
47 # inserting MemoryBIST
48     Ncp(ALL_occ_ncp) {
49       cycle(0): OCC(clk) , OCC(ramclk) , piccpu_rtl_tessent_sib_1;
50     }
51     Ncp(sti_occ_ncp) {
52       cycle(0): piccpu_rtl_tessent_sib_1 ;
53       cycle(1): piccpu_rtl_tessent_sib_1 ;
54     }
55   }
56 }
57
58 # Example EDT only; modify for your design requirements
59 # Note the additional LogicBIST options for the hybrid TK/LBIST flow

```



```
60 EDT {
61     ijtag_host_interface : Sib(edt);
62     Controller (c0) {
63         longest_chain_range : 20, 60;
64         scan_chain_count : 10;
65         input_channel_count : 1;
66         output_channel_count : 1;
67         ShiftPowerOptions {
68             present : on ;
69             min_switching_threshold_percentage : 15 ;
70         }
71         LogicBistOptions {
72             misr_input_ratio : 1 ;
73             chain_mask_register_ratio : 1 ;
74             ShiftPowerOptions {
75                 present : on ;
76                 default_operation : disabled ;
77                 SwitchingThresholdPercentage { min : 25 ; }
78             }
79         }
80     }
81 }
```

Example 3: DFT Signals Required for the Hybrid TK/LBIST Flow

The following dofile snippet shows the DFT signals required if you are performing the hybrid TK/LBIST flow only (without MemoryBIST or boundary scan in the first DFT insertion pass).

```
# For logic test
add_dft_signals ltest_en -create_with_tdr
add_dft_signals scan_en edt_update -source_node \
    { scan_en edt_update }
# You can create edt_clock and shift_clock from a test clock
add_dft_signals test_clock -source_node { scan_test_clock }
add_dft_signals edt_clock shift_capture_clock -create_from_other_signals

add_dft_signals int_ltest_en ext_ltest_en

# Required for TK/LBIST IP
add_dft_signals observe_test_point_en -create_with_tdr
add_dft_signals control_test_point_en -create_with_tdr
add_dft_signals x_bounding_en -create_with_tdr
```

Related Topics

[Sib](#)

Perform Test Point Insertion

The two-pass DFT insertion flow with hybrid TK/LBIST includes a step for test point insertion prior to performing scan chain insertion. Inserting test points increases the testability of a design by improving controllability and observability during scan testing.

The tool inserts observation points that enable the tool to observe test responses during scan cycles and control points that activate and receive pseudo-random values during built-in self test. For details, refer to “[Test Points for LBIST Test Coverage Improvement](#)” in the *Tessent Scan and ATPG User’s Manual*.

Before inserting the test points, specify a DFT signal that enables X-bounding signals. X-bounding is the process of preventing signals originating from X-generating nets (from non-scan cells, black boxes, and primary inputs) from reaching the scan cells. The tool inserts a MUX with an X-bounding enable signal that is controlled by a top-level pin. X-bounding ensures that only valid binary values propagate through the scan cells during test. For details, refer to “[X-Bounding](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

Prerequisites

- Prior to test point insertion, perform synthesis as described in section “[Performing Synthesis](#)” on page 132.”

Procedure

1. Specify the following command to set the DFT context for test point insertion:

```
set_context dft -test_point -no_rtl -design_id gate1
```

For test point insertion, you must use a gate-level netlist. Ensure that you specify a design ID with a unique name from the design ID for scan chain insertion that you perform after test point insertion.

2. Load the cell libraries.

```
read_cell_library ../lib/tessent/adk.tcelllib ../rtl/  
picdram.atpglib
```

3. Specify the same output directory that you used in the first and second DFT passes.

```
set_tsdb_output_directory ../tsdb_outdir
```

4. Load the synthesized netlist. For example:

```
read_verilog ../03.synthesis/piccpu.vg
```

5. Load the design data for the DFT hardware you previously inserted. For example:

```
read_design piccpu -design_id rtl2 -no_hdl -verbose
```

6. Set the maximum number of test points you want to add. For example:

```
set_test_point_analysis_options -total_number 50
```

Typically, the maximum number of test points should be 1%-2% of the number of flops in the design.

7. Set the test point-related insertion options.

```
set_test_point_insertion_options -observe_point_share 5
```

The `control_test_point_en` and `observe_test_point_en` enable signals are required for test point insertion, and they are automatically inferred from the `control_test_point_en` and `observe_test_point_en` DFT signals you previously added.

- Specify the type of test points you want to insert into the design.

```
set_test_point_types { lbist_test_coverage edt_pattern_count }
```

Specify both LogicBIST test coverage and EDT pattern count test point types so that the tool generates test points to reduce pattern count and improve random pattern testability.

- Specify to insert test logic on the set and reset signals to make them controllable when you insert scan chains.

```
set_test_logic -set on -reset on
```

This command increases test coverage.

- Prohibit test point insertion for Tessent-inserted scan-resource instruments (SRIs)—EDT, OCC, and LogicBIST.

```
add_notest_point [ get_instance *tessent* ]
```

You can only insert test points into scan-tested instruments (STIs).

- Analyze and optimize the test points.

```
analyze_test_points
```

Tessent Shell returns a log file that lists the test points that are inserted into the tool. You can write out a test point dofile that lists the test point locations. You can use this dofile to edit the test points you want to insert.

```
write_test_point_dofile tpi.dofile -all -replace
```

- Insert the test point and scan logic.

```
insert_test_logic
```

To generate a script to use with third-party test point insertion tools, use the following command to target the insertion script for Design Compiler.

```
insert_test_logic -write_in_tsdb off -write_insertion_script \  
testpoint_dc.tcl -insertion dc -replace
```

Examples

Example 5-15. Dofile Example for Test Point Insertion

```
set_context dft -test_point -no_rtl -design_id gate
read_cell_library ../lib/tessent/adk.tcelllib ../rtl/picdram.atpglib
set_tsdb_output_directory ../tsdb_outdir
read_verilog ../03.synthesis/piccpu.vg
read_design piccpu -design_id rtl2 -no_hdl -verbose

set_current_design piccpu
add_input_constraint reset -C0

set_test_point_analysis_options -total_number 50

# Following command inferred when X-bounding enable DFT signal was added
# -xbounding_enable [ get_dft_signal x_bounding_en ]
set_test_point_insertion_options -observe_point_share 5
set_test_point_types { lbist_test_coverage edt_pattern_count }
set_test_logic -set on -reset on
# no test points in Tessent-inserted IPs
add_notest_point [ get_instance *tessent* ]

set_system_mode analysis

analyze_test_points


insert_test_logic
report_test_logic

exit
```

Perform Scan Chain Insertion (Hybrid Flow)

Scan chain insertion for the hybrid TK/LBIST flow includes additional commands for connecting the scan enable signal, performing DRC, specifying non-scannable design objects, and X-bounding.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-16](#) on page 245.

Procedure

1. Load the design data and set the design parameters. (See lines 1-6.) Ensure that you specify a unique design ID name from the name you used for test point insertion.
2. Set DRC handling to issue warnings for E9 and E11 DRC checks. (See lines 8-10.)

```
set_drc_handling E09 W
set_drc_handling E11 W
```

These DRC checks look for possible bus contention issues. By default, Tessent Shell ignores these rules. Set these checks to issue warnings so that the X-bounding algorithm checks them and fixes any X-source contention issues.

3. Specify the pins/ports to exclude from X-bounding. (See lines 12-13.) For example:

```
set_xbounding_options -exclude { reset }
```

Exclude the pins/ports that are guaranteed to have a known value during fault simulation and never propagate an X value to the MISR. The tool does not insert X-bounding muxes at the specified pins/ports, or at the logic that is driven by these pins.

4. Run X-source analysis with the `analyze_xbounding` command to identify memory elements that might capture an unknown during LogicBIST. (See lines 17-19.)
5. Perform wrapper analysis and insertion. (See lines 21-34.)
6. Connect the scan chains to the EDT block, analyze the scan chains, and insert the scan chains. (See lines 36-50.)

Examples

The following dofile example shows a typical command flow. The highlighted command lines are unique to the process for inserting Tessent Shell LogicBIST and hybrid IP instruments in a two-pass DFT insertion process.

Example 5-16. Dofile Example for Scan Chain Insertion: Hybrid TK/LBIST

```
1 set_context dft -scan -design_id gate2
2 read_cell_library ../lib/tessent/adk.tcelllib \
3   ../lib/tessent/picdram.atpglib
4 set_tsdb_output_directory ../tsdb_outdir
5 # Reading in test point-inserted design
6 read_design piccpu -design_identifier gate1 -verbose
7
8 # Enable DRCs for X-bounding
9 set_drc_handling E09 w
10 set_drc_handling E11 w
11
12 # Set x-bounding options
13 set_xbounding_options -exclude {reset}
14
15 set_system_mode analysis
16
17 # Run X-source analysis
18 analyze_xbounding
19 report_xbounding -verbose -ignored_x_sources on
20
21 ## Perform wrapper analysis and insertion
22 # Exclude the edt_channel in and out ports from wrapper analysis
23 # The ijtag * edt_update ports are automatically excluded
24 set_wrapper_analysis_options -exclude_ports \
25   [ get_ports {*_edt_channels_*} ]
26
27 # Add a new wrapper dedicated cell on reset
```

```
28 set_dedicated_wrapper_cell_options on -ports { reset }
29 set_wrapper_analysis_options -input_fanout_flop_threshold 100 \
30   -output_fanin_flop_threshold 40
31
32 # Perform wrapper cell analysis
33 analyze_wrapper_cells
34 report_wrapper_cells -Verbose
35
36 # Find the edt_instance
37 # Instance of module *edt_lbist_c0
38 set edt_instance [get_instances -of_icl_instances [get_icl_instances \
39   -filter tessent_instrument_type==mentor::edt]]
40
41 # Connect scan chains to the EDT signals
42 add_scan_mode int_mode -type internal -edt_instances $edt_instance
43 add_scan_mode ext_mode -type external -chain_count 2
44 analyze_scan_chains
45
46 analyze_scan_chains
47 report_scan_chains
48
49 # Insert scan chains and write the scan-inserted design into the TSDB
50 insert_test_logic
51
52 report_scan_chains
53 report_scan_cells
54
55 exit
```

Related Topics


[Performing Scan Chain Insertion: Wrapped Core](#)

Performing ATPG Pattern Generation: Hybrid TK/LBIST

ATPG pattern generation for the hybrid TK/LBIST flow includes a process for generating controller chain mode patterns to test the TK/LBIST logic itself.

For details, refer to “[Performing Pattern Generation for CCM in the TSDB Flow](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-17](#) on page 247.

Prerequisites

- Perform EDT pattern generation as described in “[Performing ATPG Pattern Generation](#)” on page 135.

Procedure

1. Load the design and set the environment (see lines 1-12). Refer to “[Loading the Design](#)” on page 121 for more information.

2. Enable CCM (see lines 16-17).

```
set_static_dft_signal_values controller_chain_mode 1
```

You had added a DFT signal for `ccm_en` during IP creation, which is now being configured. If this is a port (default), then this command is not required.

3. Define the scan chain for CCM (see lines 19-22).
4. Specify `tck` or `edt_clock` for CCM, depending on which you are using in your implementation (see lines 24-25).

5. Define the pin constraints (see lines 33-37).

You must turn off core clock and reset activity. If `ccm_en` was not added as a DFT signal, then also include a pin constraint for it (enabled).

6. For retargeting, specify to pulse the CCM clock during shift. (See line 39.) The following command is only required when you use `edt_clock` for CCM and `edt_clock` is a top-level port, or when you use `tck` for CCM.

```
set_procedure_retargeting_options -pulse_during_shift edt_clock
```

The tool automatically generates a test procedure file that configures the `scan_en` and `shift_capture_clock` DFT signals. If the `edt_clock` is derived from `test_clock`, do not specify the `set_procedure_retargeting_options` command.

7. Specify the `set_current_mode` command with a unique name to indicate that you are generating CCM patterns (see line 41). For example:

```
set_current_mode ccm_stuck
```

8. Create and write the patterns (see lines 43-60).

Examples

The following example generates CCM ATPG patterns. The highlighted statements illustrate additional considerations for CCM.

Example 5-17. Dofile Example for ATPG with Controller Chain Mode: Hybrid TK/LBIST

```
1 ### DESIGN VARIABLES
2 set DesignName piccpu
3 set DesignLevel physical_block
4
5 set_context patterns -scan
6 set_tsdb_output_directory ../tsdb_outdir
7 read_cell_library ../lib/tessent/adk.tcelllib \
```


```
8     ../lib/tessent/picdram.atpglib
9 set_design_source -format tcd_memory -y ../lib/tessent -extension memlib
10
11 # Read in the scan-inserted design
12 read_design $DesignName -design_id gate2
13
14 report_dft_signals
15
16 # Enable CCM
17 set_static_dft_signal_values controller_chain_mode 1
18
19 add_scan_groups grp1
20 # These are the required scan chains for CCM
21 add_scan_chains ccm_chain grp1 control_chain_scan_in \
22     control_chain_scan_out
23
24 # Add edt_clock or tck as the primary clock source for CCM
25 add_clock 0 tck
26
27 # Specify clocks driven by primary-input ports; this is optional
28 add_clocks 0 clk
29 add_clocks 0 ramclk
30 add_clocks 0 reset
31 add_clocks 0 test_clock
32
33 # Define the pin constraints
34 add_input_constraints clk -c0
35 add_input_constraints ramclk -c0
36 add_input_constraints reset -c0
37 add_input_constraints test_clock -c0
38
39 set_procedure_retargeting_options -pulse_during_shift edt_clock
40
41 set_current_mode ccm_stuck
42
43 set_system_mode analysis
44 add_fault -module [ get_module *tessent* ]
45 report_clocks
46
47 create_patterns
48 report_statistics -detailed_analysis
49
50 report_statistics -instance piccpu_rtl2_tessent_lbist
51 report_statistics -instance \
52     piccpu_rtl2_tessent_lbist_ncp_index_decoder_inst
53 report_statistics -instance piccpu_rtl2_tessent_single_chain_mode_logic
54 report_statistics -instance piccpu_rtl2_tessent_edt_lbist_c0_inst
55
56 write_tsdb_data -replace
57 write_patterns ${DesignName}_ccm_stuck_parallel.v -verilog -parallel \
58     -replace -parameter_list {SIM_KEEP_PATH 1}
59 write_patterns ${DesignName}_ccm_stuck_serial.v -verilog -serial \
60     -replace -parameter_list {SIM_KEEP_PATH 1}
61
62 exit
```


Performing LogicBIST Fault Simulation

Perform LogicBIST fault simulation on the scan-inserted, gate-level design. LogicBIST fault simulation generates pseudo-random, parallel pattern sets.

Refer to “[Parallel Versus Serial Patterns](#)” in the *Tessent Scan and ATPG User’s Manual* for more information.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-18](#) on page 250.

Procedure

1. Load the design and set the environment (see lines 1-12). Refer to “[Loading the Design](#)” on page 121 for more information.
2. Set the current test mode to a unique name for the new pattern set you are creating to test the hybrid IP. (See line 7.)
3. Import the core’s internal mode data. (See line 9.)

Tessent Shell imports the scan-inserted design data for the EDT and OCC logic. For LogicBIST fault simulation, the tool requires EDT for PRPG/compactor/MISR configuration and chain tracing, and OCC for the clocks. Using the [import_scan_mode](#) command assumes that you used Tessent Scan to perform scan chain stitching.

For LogicBIST fault simulation, only the internal mode data is valid.

4. Add the hybrid TK/LBIST core instances. (See lines 11-12.) For example:

```
add_core_instances -instance piccpu_rtl2_tessent_lbist
```

For the hybrid TK/LBIST flow, you must explicitly add the LogicBIST controller core.

5. Specify the capture procedure names with the [set_lbist_controller_options](#) command. (See lines 16-18.)

Include the names of all Named Capture Procedures (NCPs) that are defined in the DftSpecification, along with their active percentages, regardless of whether you intend to use them in LogicBIST simulation.

The tool defines the NCPs in the Tessent Core Description (TCD) file while processing the DftSpecification. You can override the automatically generated NCPs with the [read_procfile](#) command.

6. Define the DFT signals (see lines 23-37). In addition to enabling the `ltest_en` and `int_ltest_en` logic test control signals, you must specify the following signals for the hybrid TK/LBIST flow:

```
set_static_dft_signal_values control_test_point_en 1
set_static_dft_signal_values observe_test_point_en 1
set_static_dft_signal_values xbounding_en 1
set_static_dft_signal_values tck_occ_en 1
set_static_dft_signal_values controller_chain_mode 0
```

7. Set the PLL external capture clocks if your design has a PLL that is a driving clock, but the PLL itself is driven by external clocks. (See lines 43-46.)

This indicates the number of cycles the NCPs take with respect to the LogicBIST controller clock, as specified with the `-fixed_cycles` switch.

8. Specify the maximum number of pseudo-random patterns you want the tool to simulate. (See line 48.)
9. Set the pattern source to LogicBIST and run fault simulation. (See line 51.)
10. Write out the parallel test bench and save all the data into the TSDB. (See lines 53-57.)

Examples

The following dofile example shows a typical command flow as detailed in the preceding procedure. The highlighted text illustrates additional considerations for the hybrid TK/LBIST flow.

Example 5-18. Dofile Example for LogicBIST Fault Simulation


```
1 set_context pattern -scan -design_id gate2
2 set_tsdb_output_directory ../tsdb_outdir
3 read_cell_library ../lib/tessent/adk.tcelllib ../rtl/picdram.atpglib
4 read_design piccpu
5 set_current_design piccpu
6
7 set_current_mode lbist
8
9 import_scan_mode int_mode
10
11 # Explicitly add the LogicBIST controller core
12 add_core_instances -instance piccpu rtl2 tessent_lbist
13 # EDT and OCC are loaded with the int_mode imported scan mode
14
15
16 # Define the number of patterns per NCP capture window
17 set_lbist_controller_options -capture_procedure {clk_occ_ncp 70 \
18     ramclk_occ_ncp 10 ALL_occ_ncp 10 sti_occ_ncp 10}
19
20 add_input_constraint test_clock -c0
21 add_nofault [get_instance *tessent*]
22
23 # Set the DFT signals
24 set_static_dft_signal_values ltest_en 1
```

```
25 # The following DFT signal is inferred by ltest_en 1 if you have memory
26 # set_static_dft_signal_value memory_bypass_en 1
27
28 set_static_dft_signal_values control_test_point_en 1
29 set_static_dft_signal_values observe_test_point_en 1
30 set_static_dft_signal_values x_bounding_en 1
31
32 set_static_dft_signal_values int_mode 1
33 # The following DFT signal is inferred by int_mode 1
34 # set_static_dft_signal_values int_ltest_en 1
35
36 set_static_dft_signal_values tck_occ_en 1
37 set_static_dft_signal_values controller_chain_mode 0
38
39 report_static_dft_signal_settings
40
41 set_system_mode analysis
42
43 # Specify the number of cycles the NCPs take with respect to the LogicBIST
44 # controller clock
45 # This ensures that the tool runs the testproc correctly
46 set_external_capture_options -fixed_cycles 4
47
48 set_random_patterns 1000
49 add_faults -all
50
51 simulate_patterns -source bist -store_patterns all
52
53 # Write the parallel pattern Verilog testbench
54 write_patterns piccpu_lbist_parallel.v -verilog -parallel -replace \
55     -parameter_list {SIM_KEEP_PATH 1 SIM_POST_SHIFT 3}
56 # Save the TCD, PatternDB, flat model, and fault list to the TSDB
57 write_tsdb_data -replace
58
59 exit
```

Perform LogicBIST Pattern Generation

Generate LogicBIST patterns using the scan-inserted and LogicBIST fault simulated design. The process includes IJTAG pattern retargeting from the extracted ICL module description for the design. Tessent Shell translates (or retargets) the PDL commands so that you can control the LogicBIST controller through the TAP controller/IJTAG infrastructure.

Note

 Do not confuse IJTAG retargeting with ATPG (or scan) pattern retargeting as described in “[Hierarchical DFT Terminology](#)” on page 142.

Procedure

1. Load the design and set the environment (see lines 1-12). Refer to “[Loading the Design](#)” on page 121 for more information.

2. Set the context to patterns -ijtag and set the design ID to the name of the scan-inserted, gate-level netlist generated during scan chain insertion.

When you specify the -ijtag switch, Tessent Shell automatically accesses the ICL module description for the current design, which enables IJTAG retargeting mode.

3. Define clocks and constraints (see lines 7-10).
4. Generate and validate the IJTAG patterns for the design (see lines 14-16).
5. Run and check the test bench simulations (see lines 18-23). The following step is important for the hybrid TK/LBIST flow:
 - a. Specify the simulator option to keep all the logic gates for simulation.

The following example applies to Questa® SIM:

```
run_testbench_simulations -simulator_options \  
    { -voptargs="+acc" }
```

For correct pattern simulation, use the applicable simulator option to ensure that the necessary design logic is not optimized away during elaboration.

Tessent Shell simulates the logic test operations only, which means the test bench does not connect all the pins in the design. The tool issues warnings for the unconnected pins. To filter these warnings out, you can use the `run_testbench_simulations -simulation_option +nowarnTSCALE` option.

Examples

The following dofile shows a command flow to generate IJTAG patterns for LogicBIST. Highlighted text illustrates additional considerations for the hybrid TK/LBIST flow.

```
1  set_context patterns -ijtag -design_id gate2  
2  read_cell_library ../lib/tessent/adk.tcelllib ../rtl/picdram.atpglib  
3  set_tsdb_output_directory ../tsdb_outdir  
4  read_design piccpu  
5  set_current_design piccpu  
6  
7  # This is the clock driving the internal PLL  
8  add_clocks 0 clk -period 100ns  
9  # Scan enable has to be tied to low  
10 add_input_constraint scan_en -c0  
11  
12 set_system_mode analysis  
13  
14 read_config_data ../lbist_mbist_pattern.patspec  
15  
16 process_patterns_specification  
17  
18 set_simulation_library_sources -v { ../lib/verilog/adk.v \  
19     ../lib/verilog/picdram.v ../lib/verilog/SYNC_1R1W_256x16.v }  
20  
21 run_testbench_simulation -simulator_options { -voptargs="+acc" \  
22     -quiet} -wait
```

```
23 check_testbench_simulations -report_status  
24  
25 exit
```

Running Multi-Load ATPG on Wrapped Core Memories with Built-In Self Repair

If your design has wrapped cores that include repairable memories, and you want to test the logic around the memories at speed, then you use multi-load ATPG patterns.

Overview of Multi-Load ATPG on Memories for Wrapped Cores With Built-in Self Repair 254

Performing Multi-Load ATPG Pattern Generation..... 255

Performing Multi-Load Top-Level ATPG Pattern Retargeting..... 258

Overview of Multi-Load ATPG on Memories for Wrapped Cores With Built-in Self Repair

When generating ATPG multi-load patterns for memories with Built-In Self Repair (BISR), the memories inside the wrapped cores are tested first. If any memories need to be repaired because of defects that are present inside the memory, then the BISR registers are programmed such that the contents of these registers vary based on the repair solution that is stored in the fuses at the parent level.

To generate multi-load ATPG patterns, the BISR registers need to be excluded from the retargetable ATPG patterns that you created for the wrapped core. At the wrapped core level, a `test_setup_iCall` resets the BISR registers, resulting in all spare elements being unallocated before ATPG is executed. Only the memory control ports and data ports participate in the ATPG patterns; the memory repair ports are excluded. If memory defects are detected and repaired via the BISR registers, the retargeted ATPG patterns also pass.

You must provide an ATPG model of the eFuse during pattern generation to meet the E14 design rule requirements. If such a model is not available, you can instead provide an input constraint on the `fuseValue` output pin of the eFuse interface module. To do this, create a pseudo-port associated with the `fuseValue` output pin of the fuse box interface instance. Then, add a constant 0 input constraint on the “`fuseValue`” pseudo-port, as shown in the following example:

```
add_primary_inputs chip_rtl_tessent_mbisr_controller_inst/\
  chip_rtl_tessent_mbisr_generic_fusebox_interface_instance/fuseValue \
  -pseudo_port_name fuseValue
add_input_constraints -c0 fuseValue
```

For more details, refer to the [add_primary_inputs](#) and [add_input_constraints](#) commands.

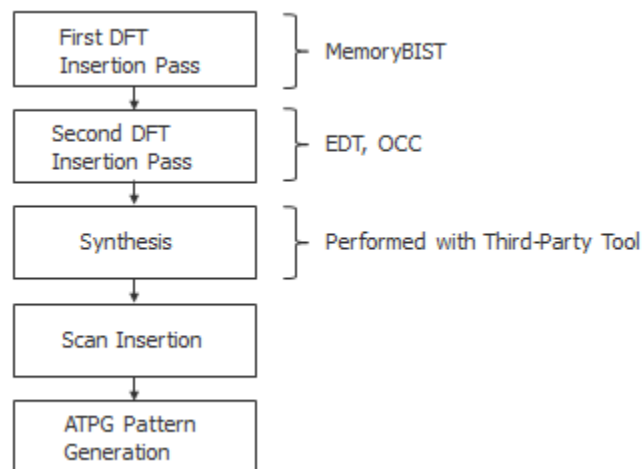
Performing Multi-Load ATPG Pattern Generation

When you need to generate multi-load ATPG patterns when the memory is not bypassed but used during ATPG there is no difference in the flow steps used for inserting DFT. You use this procedure to bypass the memories.

The DFT insertion at the wrapped core level for memories with BISRs is similar to the two-pass pre-scan DFT insertion process for wrapped cores with the primary difference being the ATPG pattern generation. There are no changes to creating the graybox model, and running the wrapped core in external mode for stuck and transition patterns by bypassing the memories. There are no changes for running the wrapped core in internal mode for stuck and transition patterns by bypassing the memories.

You should be familiar with the two-pass pre-scan DFT insertion flow as described in “[RTL and Scan DFT Insertion Flow for Physical Blocks](#)” on page 146, especially as related to performing ATPG pattern generation. [Figure 5-30](#) shows this flow.

Figure 5-30. Two-Pass Insertion Flow for RTL, Wrapped Cores




The `init_bisr_chains` iProc contains Tcl code you can use to initialize the BISR registers at any level (core or chip-level) as follows:

- At the wrapped core level, the iProc performs an asynchronous clear of the BISR registers.
- At the chip-level, the iProc initiates a FuseBox controller power-up emulation.

The power-up emulation clears the BISR registers if the fuse box has not been programmed, or initializes the BISR registers with the repair data if memory repair information was programmed in the fuse box. See “[Creating and Inserting the BISR Controller](#)” in *Tessent MemoryBIST User's Manual*.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-19](#) on page 257.

Prerequisites

- You have performed the “[RTL and Scan DFT Insertion Flow for Physical Blocks](#)” on page 146 up to the ATPG pattern generation step.

Procedure

1. Load the Tessent Cell Library for the memory. (See lines 1-9.)
2. Load the design. (See lines 11-13.)
3. Set the DFT Signal memory_bypass to 0, so memory is not bypassed. (See lines 24-26.)
4. Load the PDL for the Memory BISR chains, which has an iProc (init_bisr_chains) that is called with `set_test_setup_icall -non_retargetable`. (See lines 28-35.)

The init_bisr_chains iProc contains Tcl code you can use to initialize the BISR registers at any level (core or chip-level) as follows:

- At the wrapped core level, the iProc performs an asynchronous clear of the BISR registers.
 - At the chip level, the iProc initiates a FuseBox controller power-up emulation.
5. Generate the multi-load ATPG patterns. (See lines 39-40.)
 6. Write the design data and patterns to the TSDB. (See lines 42-44.)
 7. Write out the Verilog test benches for simulation. (See lines 47-50.)

Results

At the wrapped core level, if you inject a fault in the repairable memory, and then run the multi-load ATPG Pattern, the run should fail. This check shows that if a repairable memory has defect that is not repaired then when you retarget the multi-load ATPG patterns from the top level, then the multi-load ATPG pattern run will fail.

You must run the memoryBIST inside the wrapped core to determine if the memory is repairable. If it is, then the repairable information from Built-In Redundancy Analysis (BIRA) to BISR must be stored and the repair performed by storing the repairable contents using the fusebox repair solution that you use at the top-level.

Examples

The following example for the repairable memory “MGC_SYNC_1RW_1024x8_C” generates ATPG patterns that is run by the memory.

Example 5-19. Multi-Load ATPG Pattern Generation for Memories with BISR

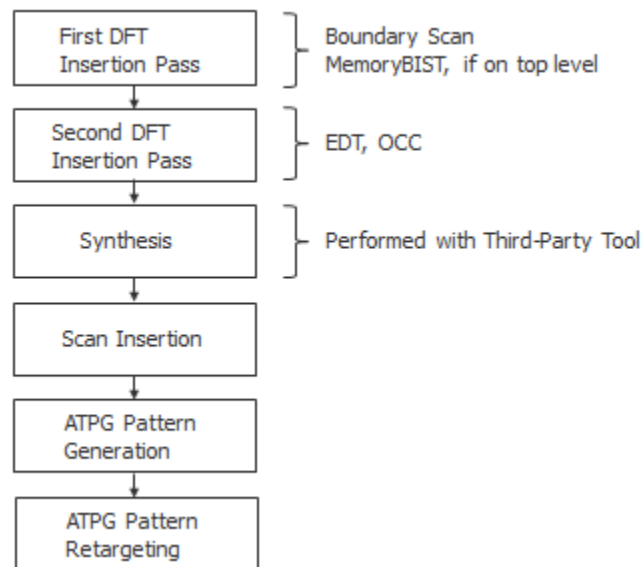
```
1  set_context patterns -scan
2
3  # Set the location of the TSDB. Default is the current working directory.
4  set_tsdb_output_directory ../tsdb_outdir
5
6  # Read Tessent Library
7  read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
8  # Read the Tessent Cell Library for the memory
9  read_cell_library ../../../../library/memories/MGC_SYNC_1RW_1024x8_C.atpglib
10
11 # Read in the scan inserted netlist/design
12 read_design processor -design_id gate -verbose
13 set_current_design processor
14
15 # Specify the current mode using a different name than what was used
16 # with the add_scan_mode command
17 set_current_mode edt_int_multi_load_atpg -type internal
18 report_dft_signals
19
20 # Extract the scan chains using the internal mode specified during
21 # scan insertion with the add_scan_mode command
22 import_scan_mode int_mode -fast_capture_mode on
23
24 # Memory is used during ATPG, by setting the DFTSignal memory_bypass
25 # to "0"
26 set_static_dft_signal_values memory_bypass_en 0
27
28 # Read in the PDL for the memory BISR
29 source \
30 ../tsdb_outdir/instruments/processor_rtl1_mbisr.instrument/\
31 processor_rtl1_tessent_mbisr.pdl
32
33 # Specify the iProc init_bisr_chains for the memory BISR as
34 # non_retargetable
35 set_test_setup_icall init_bisr_chains -non_retargetable
36
37 report_statistics -detail
38 # Generate patterns
39 create_patterns
40 report_statistics -detail
41
42 # Store TCD, flat_model, fault list and patDB format files in the TSDB
43 # directory
44 write_tsdb_data -replace
45
46 # Write test benches for Verilog simulation
47 write_patterns patterns/processor_multi_load_parallel.v \
48   -verilog -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
49 write_patterns patterns/processor_multi_load_serial.v
50   -verilog -serial -replace -parameter_list {SIM_KEEP_PATH 1}
51 exit
```

Performing Multi-Load Top-Level ATPG Pattern Retargeting

When you have wrapped cores with repairable memories, then at the chip-top level you need to insert a memory BISR controller. This can be done along with the TAP, boundary scan, and MemoryBIST insertion if there are any memories present at the top-level.

You should be familiar with the RTL and scan DFT insertion flow for the top as described in “[RTL and Scan DFT Insertion Flow for the Top Chip](#)” on page 165, especially as related to performing ATPG pattern retargeting. [Figure 5-31](#) shows this flow.


Figure 5-31. Two-Pass Insertion Flow for RTL, Top Level



For retargeting multi-load ATPG patterns where the memories are not bypassed from a lower level wrapped core that has repairable memories, specify the correct retargeting mode signal. Subsequently, use the [add_core_instances](#) command to load the specific core with the correct ATPG mode that you want to retarget.

Finally, you need to source the PDL of the BISR chain with the [iProc](#) `init_bisr_chains`. This PDL was created when the BISR controller RTL was generated at the chip-level. The `iProc` detects the presence of the fuse box controller and initiates a `PowerUpEmulation`. When the `iProc` is called and no fuse box controller is found, the `iProc` performs an asynchronous clear of the BISR chains using the primary inputs (`bisr_reset` port).

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 5-20](#) on page 259.

Prerequisites

- You have performed the “[RTL and Scan DFT Insertion Flow for the Top Chip](#)” on page 165 up to the ATPG retargeting step.

Procedure

1. Set the context to pattern retargeting. (See lines 1-2.)
2. Specify the TSDB directory and open the TSDB. (See lines 4-8.)
3. Read the Tessent Cell Library. (See lines 10-11.)
4. Load the Verilog design. (See lines 13-15.)
5. Set the retargeting from the lower-level patterns. (See lines 20-24.)
6. Read the PDL of the memoryBISR chains that has the iProc “init_bisr_chains”. This PDL is different than the one created at the wrapped core level. (See lines 30-34.)
7. Change mode to analysis. (See line 36.)

Examples

The following example retargets the ATPG pattern from the lower level to the chip-level.

Example 5-20. ATPG Pattern Retargeting for Memories with BISR

```

1  # Set the context to retarget ATPG Patterns from lower level child cores
2  set_context pattern -scan_retargeting
3
4  # Point to the TSDB directory
5  set_tsdb_output_directory ../tsdb_outdir
6
7  # Open all the TSDB of the child cores
8  open_tsdb ../../wrapped_cores/processor/tsdb_outdir
9
10 # Read the tessent cell library
11 read_cell_library ../../library/standard_cells/tessent/adk.tcelllib
12
13 # Read the verilog
14 read_design chip_top -design_id gate
15 read_design processor -design_id gate -view graybox -verbose
16
17 set_current_design chip_top
18
19
20 # Retarget Transition patterns from processor
21 set_current_mode retarget1_processor_multi_load
22 set_static_dft_signal_values retargeting1_mode 1
23 add_core_instances -instances {processor_inst1 processor_inst2} \
24   -core processor -mode edt_int_multi_load_atpg
25
26 report_core_descriptions
27 import_clocks -verbose
28 report_clocks
29

```

```
30 # Read the PDL of the MBISR chains that has the iProc "init_bisr_chains"
31 # that needs to be called before running the ATPG pattern.
32 source ../tsdb_outdir/instruments/chip_top_rt11_mbisr.instrument/\
33 chip_top_rt11_tessent_mbisr.pdl
34 set_test_setup_icall init_bisr_chains -front
35
36 set_system_mode analysis
37 report_clocks
38
39 # write the TCD file for chip-level in the TSDB outdir
40 write_tsdb_data -replace
41 # Read the patterns to be retargeted
42 read_patterns -module processor -fault_type transition
43 set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]
44
45
46 # Write Verilog patterns for simulation
47 write_patterns patterns/processor_edt_multi_load_retargeted.v -verilog \
48 -parallel -replace -begin 0 -end 7 -scan -parameter_list {SIM_KEEP_PATH 1}
49 write_patterns patterns/processor_edt_multi_load_retargeted_serial.v \
50 -verilog -serial -replace -Begin 0 -End 2 \
51 -parameter_list {SIM_KEEP_PATH 1}
52 exit
```

Built-in Self Repair in Hierarchical Tessent MemoryBIST Flow

You use the BISR registers to control the repair ports of repairable memory. The BISR logic insertion tasks include inserting BISR chains in a block and connecting a BISR controller to existing BISR chains, to an external fuse box, and to system logic. This omits the additional BISR capabilities of Repair Sharing and Fast BISR Loading.

You perform the first two tasks in a bottom-up, hierarchical design methodology where BISR chains are inserted in lower-level blocks before being connected at the chip top level to the BISR controller. This is the most frequently used method for inserting the repair logic.

You complete the BISR insertion task with the memoryBIST insertion in the same pass at the block or chip level. If the BISR insertion task is carried out separately from the memory BIST insertion, you must perform BISR insertion after you have completed all memoryBIST insertion tasks at the block or chip level in a single pass.

If you use Tessent Scan, do not scan test the BISR hardware if you plan to generate multi-load patterns for logic test. Tessent Scan does this automatically.

In contrast, if you use a third-party scan insertion tool, then the `non_scannable_instance_list` in the `dft_info_dictionary` should be honored and is located in the TSDB in the `dft_inserted_designs` directory—see “[RTL and DFT Insertion Flow With Third-Party Scan](#)” on page 189.

For additional information, see “[First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan](#)” on page 168.

[Repair Sharing Overview](#)

[Fast BISR Loading Overview](#)

Performing Block Level BISR Insertion	261
Performing Chip Level BISR Insertion	264
Verification of Block and Chip Level BISR	266

Performing Block Level BISR Insertion

The insertion of BISR chains is automatic if memories instantiated in the design have spare resources described in their memory library file. BISR registers associated to repairable memories are connected together to form scan chains.

Assigning Memories to Power Domains

By default, all repairable memories are part of the same power domain, and the tool creates a single BISR chain. If, however, more than one power domain exists, multiple BISR chains are required.

The tool determines the power domain of a memory instance by its `bisr_power_domain_name` attribute, which you specify in the following two ways:

- Reading a CPF or UPF file corresponding to the design using the [read_cpf](#) or [read_upf](#) command. This is the recommended method.
- Manually setting the `bisr_power_domain_name` attribute using the [set_attribute_value](#) command.

The following example defines power domains with UPF:

UPF:

```
create_power_domain pd_A
create_power_domain pd_AA -element {mem3 mem4}
create_power_domain pd_AB -element {mem5 mem6}
```

Generate BISR segment order file:

```
BisrSegmentOrderSpecification {
  PowerDomainGroup(pd_AA) {
    OrderedElements {
      mem3;
      mem4;
    }
  }
  PowerDomainGroup(pd_AB) {
    OrderedElements {
      mem5;
      mem6;
    }
  }
}
```

For additional implementation details, see “[Inserting BISR Chains in a Block](#)” in the *Tessent MemoryBIST User's Manual*.

Controlling the BISR Chain Order

BISR chains are connected according to the content of the `BisrSegmentOrderSpecification` wrapper containing a list of memory instances defining the BISR chain order.

This file is generated automatically when [check_design_rules](#) successfully completes. When a DEF file is provided, the BISR segments are ordered using an algorithm that optimizes the routing based on the memory coordinates. If a DEF file is not provided, the memories are sorted alphabetically within each power domain group.

If you wish to change the generated BISR change order, the you can manually modify the `<design_name>.bISR_segment_order` file and change the order of memory instance names before executing the [process_dft_specification](#) command.

Turning off Insertion of BISR Registers

It is possible to turn off the generation of BISR registers for specific memory instances.

You do this by issuing the [set_memory_instance_options](#) command as follows:

```
set_memory_instance_options memory_inst -use_in_memory_bISR_dft_specification off
```

Excluding Child Block BISR Chains

When you do not implement memory repair in a parent design but integrate sub-blocks or physical blocks that already contain BISR chains, you must not connect or use these chains, and you must properly tie the chains off.

Note



This is not a typical use model and is rarely implemented.

Related Topics

[Inserting BISR Chains in a Block](#)

[Excluding Child Block BISR Chains](#)

Performing Chip Level BISR Insertion

As with the block level, the insertion of BISR chains is automatic if memories instantiated in the design at the chip level have spare resources. When you insert chip level BISR, you must also choose a functional repair clock and connect the BISR controller to an existing BISR chain, an external fuse box, and to system logic.

Functional Repair Clock	264
Connection of the BISR Controller to Existing BISR Chains	264
Connection of the BISR Controller to an External Fuse Box	265
Connection of the BISR Controller to System Logic	266

Functional Repair Clock

The distributed architecture and conservative clocking methodology used for self-repair require some care in the selection of the functional repair clock used to apply repairs during chip power up. You should use a functional clock of 10 MHz or less in that functional mode.

For implementation details, refer to “[Inserting BISR Chains in a Block](#)” in the *Tessent MemoryBIST User's Manual*.

Connection of the BISR Controller to Existing BISR Chains

As with block level BISR insertion, BISR chains are connected according to the content of the `BisrSegmentOrderSpecification` wrapper containing a list of memory instances defining the BISR chain order as well as lower level blocks containing pre-inserted BISR chains.

The tool automatically generates this file when the `check_design_rules` command successfully completes.

The following example shows two instances of the same block (blockA, instances blockA_clka_i1 and blockA_clkb_i2), which has two BISR chains partitioned by two power domains (pd_AA and pd_AB):

```
BisrSegmentOrderSpecification {
  PowerDomainGroup (pd_AA) {
    OrderedElements {
      blockA_clka_i1/pd_AA_bisr_si;
      blockA_clkb_i2/pd_AA_bisr_si;
    }
  }
  PowerDomainGroup (pd_AB) {
    OrderedElements {
      blockA_clka_i1/pd_AB_bisr_si;
      blockA_clkb_i2/pd_AB_bisr_si;
    }
  }
}
```

As with the block level BISR insertion, if you change the generated BISR chain order, the `<design_name>.bisr_segment_order` file, you can manually modify this file and change the order of memory instance names before executing the [process_dft_specification](#) command.


Connection of the BISR Controller to an External Fuse Box

The BISR controller hardware is created at the top level of the chip automatically. The BISR controller is accessed using the TAP. The BISR chain control ports are automatically connected to the BISR controller. The BISR controller is also connected to the fuse box. A typical implementation of BISR is with an external fuse box.

The hardware implements several functions and can be used to perform the following operations:

- Compress the repair information and write the result into the fuse box
- Decompress the fuse box contents and shift the contents into the chip BISR chain
- Initialize or observe BISR chain content via the TAP
- Read and program fuses via the TAP

Note

 A BISR controller with an *internal* fuse box is supported. For details, refer to [“Implementing and Verifying Memory Repair”](#) in the *Tessent MemoryBIST User's Manual*.

The design instance for the external fuse box must already be instantiated in the design. Typically, the fuse box is instantiated within a module that also contains interface logic. All input ports of the module should be tied off, and the output ports should be left open. When

executing the [process_dft_specification](#) command, the tool disconnects all input ports and then reconnects the ports to the BISR controller module.

When using an external fuse box, you must set the `fuse_box_location` property in the `MemoryBisr/Controller` wrapper to `external`. The `fuse_box_interface_module` property located in the same wrapper can be used to specify the library module for the external fuse box. If this is not specified, the library module is inferred from the design instance specified in the `design_instance` property of `ExternalFuseBoxOptions` wrapper. If neither of these properties are specified and only a single `tcd_fusebox` file exists in the design, the fuse box module is inferred from this `tcd_fusebox` description.

The core description for the external fuse box can automatically be read in during module matching using the “`set_design_sources -format tcd_fusebox`” command or directly using the [read_core_descriptions](#) command.

If the instantiated module has a core description with a `FuseBoxInterface` wrapper, then connections between the fuse box controller and the fuse box interface are completed automatically. If a core description is not available, or not complete, explicit connections can be made in the `MemoryBisr/Controller/ExternalFuseBoxOptions/ConnectionOverrides` wrapper.

Related Topics

[Fuse Box Interface](#)

[Connecting the BISR Controller to an External Fuse Box](#)

Connection of the BISR Controller to System Logic

Typically, system logic is connected to the BISR controller for initiating memory repair and monitoring the progress of the operation. All connections are specified in the `DftSpecification` configuration file in the `MemoryBisr:Controller` section.

- The BISR controller input `clk` must be driven by an appropriate functional clock. The connection is made by specifying the `repair_clock_connection` property in the `DftSpecification/MemoryBisr/Controller` wrapper.
- The BISR controller input `resetN` is the signal used to reset the BISR chains and initiate memory repair. The connection is made by specifying the `repair_trigger_connection` property in the `DftSpecification/MemoryBisr/` wrapper.

Verification of Block and Chip Level BISR

You must verify the BISR at both the block and chip level to guarantee functional hardware.

Block Level BISR Verification

The default signoff is the `PatternsSpecification` you generate with Tessent Shell.

You generate this using the `command`, which tests the connectivity of the BISR chain using a pattern named `MemoryBisr_BisrChainAccess`.

You can create future verification patterns as follows:

- Executing fault-inserted MemoryBIST
- Performing BIRA-to-BISR capture
 - Scan external BISR chain into the internal BISR chain
- Executing post-repair memory BIST

Chip Level BISR Verification

The default signoff is the `PatternsSpecification` that you generate with Tessent Shell using the `create_patterns_specification` command for the BISR hardware. `PatternsSpecification` verifies the `FuseBoxAccess`, `BisrChainAccess`, and `Autonomous` modes of operation of the BISR controller and BISR chains via the TAP.

You do not need to run `memoryBIST` with fault-inserted memories at the top level of the chip. This type of verification is better performed at the block level where `memoryBIST` can be run on the full address space of all memories.

Related Topics

[PatternsSpecification](#)

[create_patterns_specification](#)

[Verifying BISR at the Block Level](#)

[Top-Level Verification and Pattern Generation](#)

Chapter 6

Tessent Shell Automotive Workflow

This chapter describes a pre-layout DFT insertion workflow that helps you meet the safety, quality, and reliability requirements of the ISO 26262 “Road vehicles - Functional safety” standard. The usage model provides capability for both manufacturing and in-system chip testing and diagnosis. It also provides examples of how to run, and how to generate UDFM files for, Automotive-Grade ATPG.

This chapter highlights the important steps required for you to achieve the optimal automotive DFT strategy early in your design process. At a high level, this is a comprehensive hybrid TK/LBIST two-pass RTL and scan DFT insertion flow that includes MemoryBIST, MBISR, advanced BAP, boundary scan, and in-system test.

Tip

i This is an advanced-level workflow, which assumes knowledge about the Tessent Shell two-pass DFT insertion process. Familiarize yourself with the basic hybrid TK/LBIST flow for flat models as described in “[Hybrid TK/LBIST Flow for Flat Designs](#)” on page 232. In addition, because this is a hierarchical design, the workflow described in this chapter follows the bottom-up DFT insertion process as described in “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141.

Refer to the following test case for a detailed usage example of the flow described in this section:

```
tessent_automotive_reference_flow_<software_version>.tgz
```

You can access this test case by navigating to the following directory:

```
<software_release_tree>/share/UsageExamples/
```

Introduction to Tessent Automotive	270
Test Case Overview and Objective.	271
Core-Level DFT Insertion for Automotive	276
DFT Insertion Flow for the Processor Core Physical Block	277
DFT Insertion Flow for the GPS Baseband Physical Block.....	307
Top-Level DFT Insertion for the Automotive Flow	314
First DFT Insertion Pass: Top with MemoryBIST, BISR, and Boundary Scan.....	314
Second DFT Insertion Pass: Top with EDT, OCC, and In-System Test	320
Scan Insertion for the Top Design	325
ATPG Pattern Generation for the Top Design	327
ATPG Pattern Retargeting for the Top Design.....	328

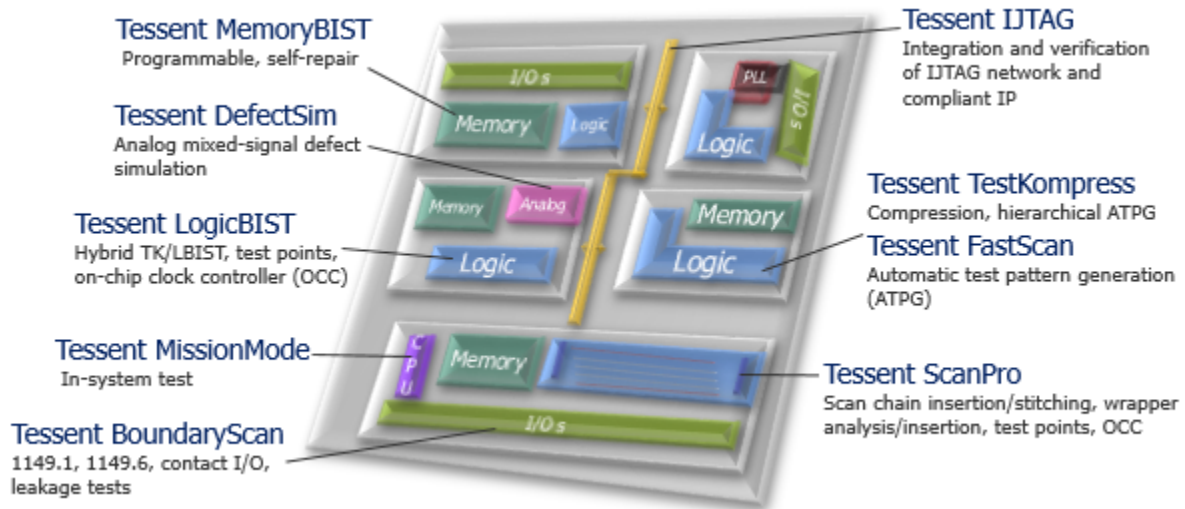
Interconnect Bridge/Open UDFM Generation for the Top Design	328
Cell-Neighborhood UDFM Generation for the Top Design.....	328
Automotive-Grade ATPG Pattern Generation for the Top Design	329
UDFM Generation for Cell-Aware ATPG	329
TCA Based Pattern Sorting.....	331
Functional Mode Fault Tolerance for Static IJTAG Signals.....	333

Introduction to Tessent Automotive

Designs with aggressive quality and reliability requirements, such as those required to comply with the ISO 26262 standard, must exercise efficient test strategy to ensure high quality with very low defective parts per billion (DPPB). One of the key ISO 26262 requirements is the ability to test safety-critical portions of a system during power-on and periodically during in-system operation.

The Tessent product suite provides a flexible design flow that supports a high-level of automation and integration, creating an RTL-based hierarchical automotive flow that you can use for implementation of a DFT solution. It provides design rule checking, test planning, integration, and verification at the RTL- and gate-level, and it achieves high coverage for functional logic, memories, and test IPs.

Figure 6-1. Integrated Tessent DFT Solution for Automotive



Tessent automotive includes the following features:

- Low-impact in-system test.
 - Low resource requirements to design system-side logic for in-system test.
 - Interface to access IJTAG network at the chip or block level.
- Unified environment.

- Design introspection capabilities.
- Full access to design and pattern data.
- All information stored in one place, the Tessent Shell Database (TSDB).
- Test scheduling flexibility using an IJTAG infrastructure.
 - The MissionMode controller used for in-system test can access and operate any instrument in the IJTAG network.
 - Any LogicBIST or MemoryBIST (IJTAG-compliant IP) test can be converted to in-system test.
 - Run the same or different tests multiple times during in-system operation.
- Verification setup.
 - Test time is automatically calculated.
 - ROM data contents are also generated for the MissionMode controller.
 - Designer can easily verify in-system test with Verilog simulation.
- Area optimization
 - The hybrid TK/LBIST IP shares the compression logic for scan and LogicBIST.
 - Several optimization schemes for MemoryBIST.
- Fully hierarchical flow.
 - RTL-level generation and insertion.
 - Enable flow automation by using DFT signals.
 - Pattern re-targeting to generate and validate patterns at any level of the design hierarchy.
- Direct access to MemoryBIST.
 - Advanced BIST access port (BAP) to access MemoryBIST and consider limited test time of in-system test.
 - Direct control of algorithm, operation set, step, and memories.
 - Incremental repair to improve reliability of automotive application.

Test Case Overview and Objective

In the `tessent_automotive_reference_flow` test case, you perform an RTL and scan DFT pre-layout insertion process in a bottom-up manner for each core and sub-block in the design. After inserting DFT into all the lower-level physical blocks, perform the same insertion process into

the parent blocks until you reach the top level of the chip. You also create ATPG core-level and top-level patterns and perform ATPG pattern re-targeting of the wrapped cores at the parent physical block and top level.

Note



The test case and this chapter assume that you understand the Tessent concepts and terms for hierarchical DFT as described in “[Hierarchical DFT Overview](#)” on page 96.

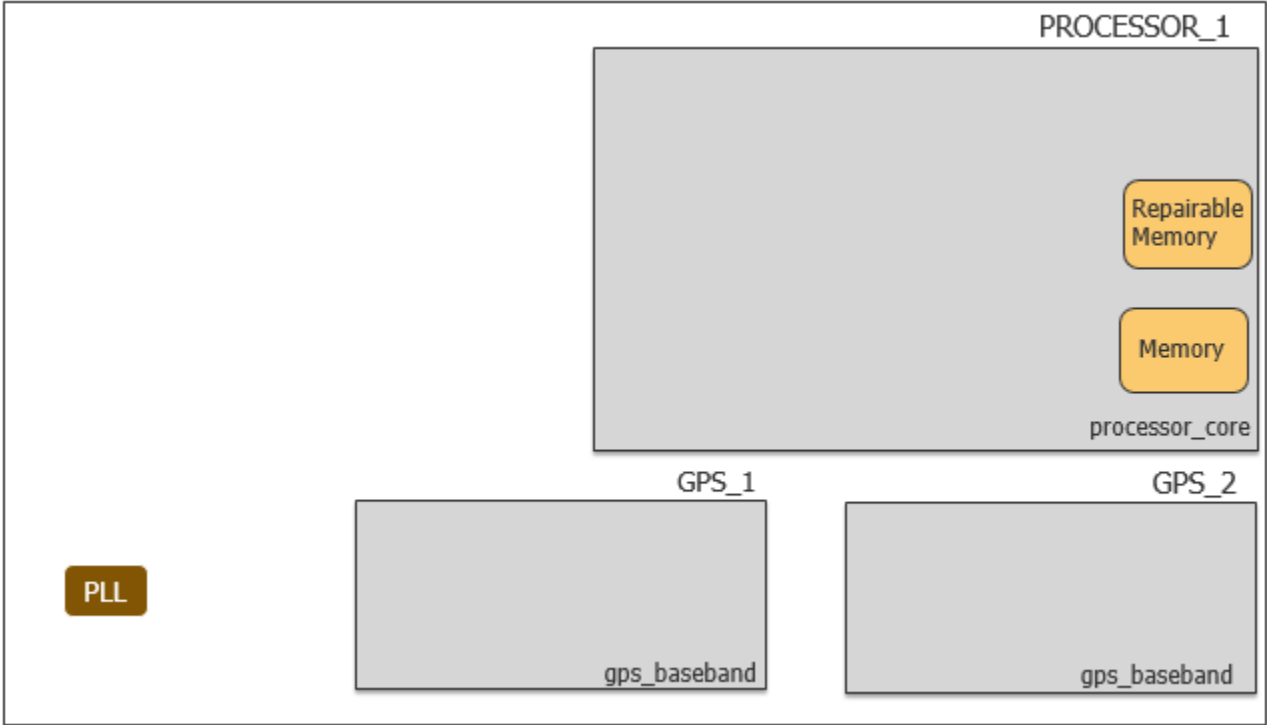
Your objective is to integrate the DFT IP and provide a mechanism to test your logic and memories during in-system operation. Considering your in-system test time constraints and coverage requirements, you must:

- Analyze and decide on the algorithms you want to run on the memories in-system for power-on reset (PoR) and periodic test.
- Analyze and decide the number of pseudo-random patterns during PoR and periodic test. (This depends on the scan chain length and shift clock frequency.)
- Insert test point and x-bounding cells for LogicBIST.
- Use the advanced BAP capability to provide a low-latency protocol for configuring the MemoryBIST controller, executing Go/NoGo tests, and monitoring the pass/fail status for in-system and manufacturing test.
- Insert the MissionMode (also known as “in-system test”) IP to access IJTAG instruments in system, and, depending on the length of your IJTAG network, insert the in-system test controller within individual cores as well as at the top level. As a reference, the tool executes both CPU-based and DMA-based in-system test options.
- Test the inserted test logic. That is, the LogicBIST and EDT IP blocks in addition to testing the core design logic. Do this by using [controller chain mode](#), which enables you to generate ATPG patterns that target the hybrid EDT/LBIST blocks and LogicBIST controller in addition to the single chain mode logic and in-system test controller. While MemoryBIST IPs can be scan tested, under EDT test mode you can test them in system through LogicBIST.

Design Overview

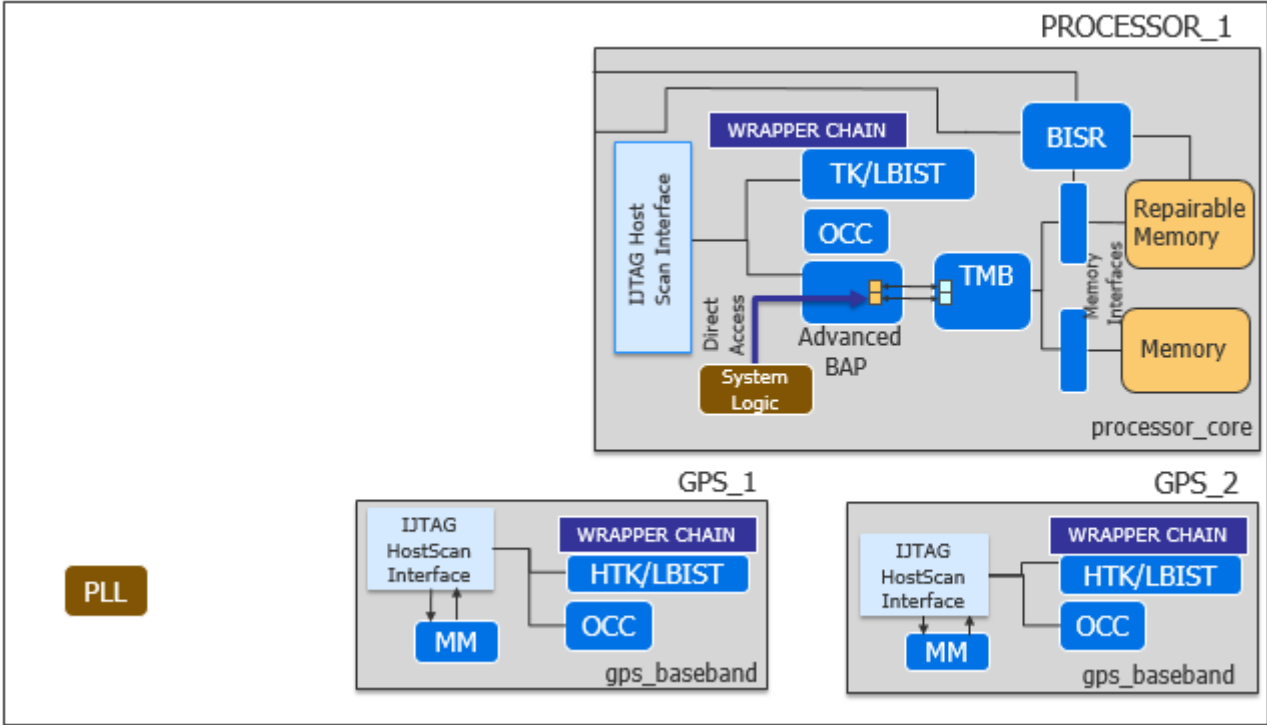
The test case uses an UltraSPARC (open-source) design with a processor core and two GPS basebands. The processor core design includes non-repairable and repairable memory, and the GPS baseband is a logic-only core that is instantiated twice.

Figure 6-2. Initial Design for Automotive Test Case



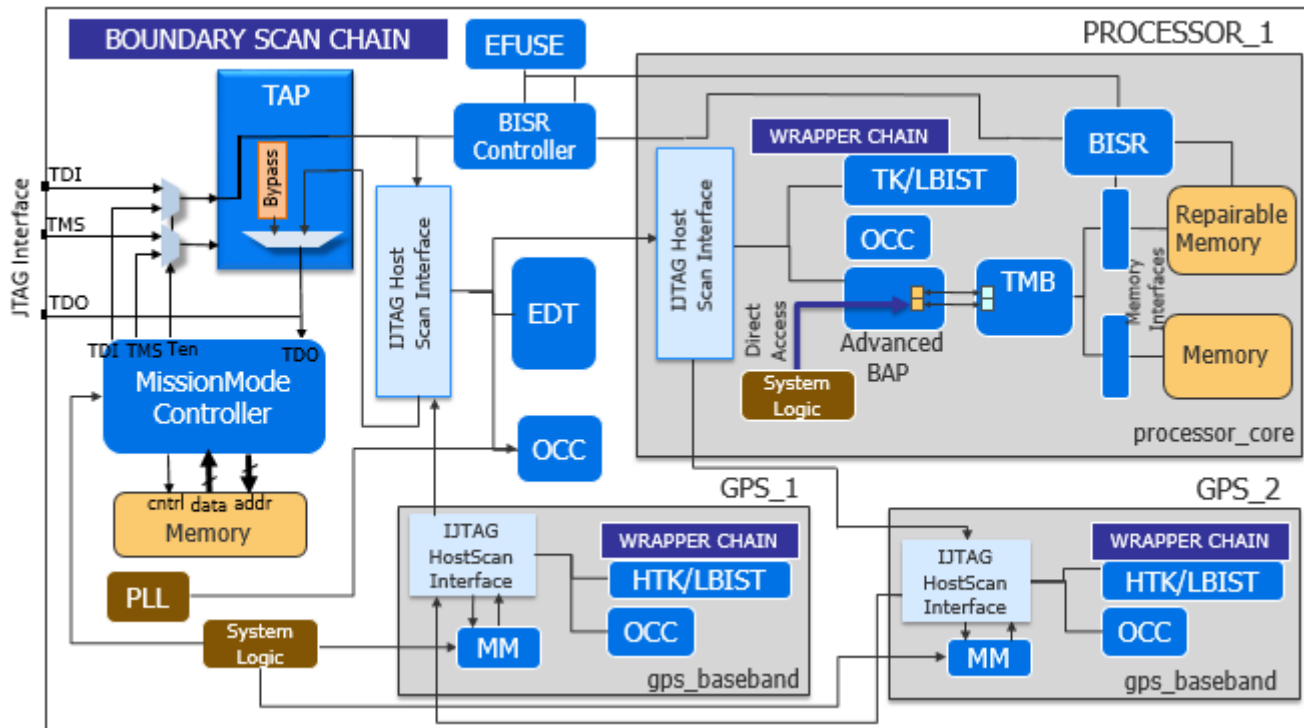
Following the bottom-up flow, you insert the following DFT logic at the core level.

Figure 6-3. DFT Integration at the Core Level for Automotive



After you have inserted the core-level IP, you perform top-level DFT logic insertion and integration, followed by ATPG and pattern retargeting.

Figure 6-4. DFT Integration at the Top Level for Automotive

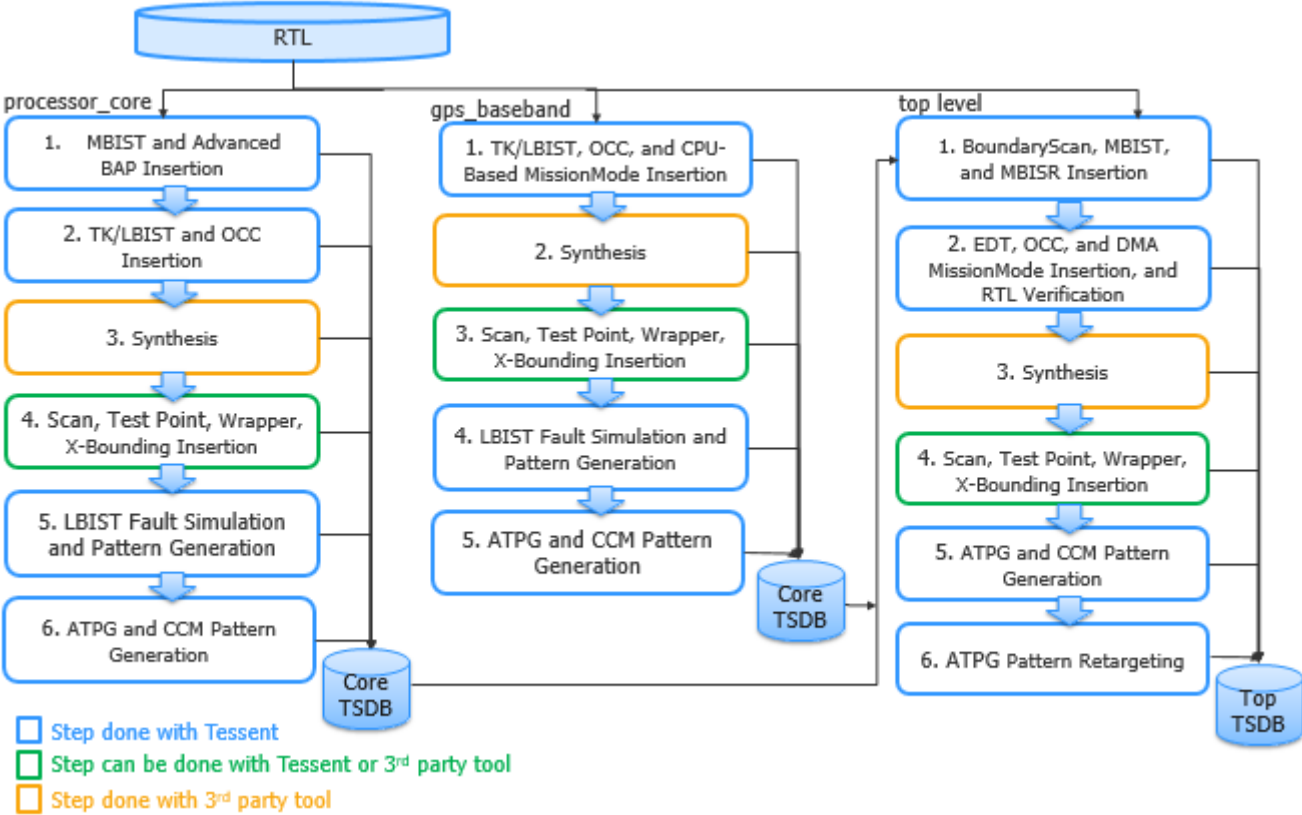


To test your IJTAG instrument in system and achieve high test coverage for your automotive IC, you insert a MissionMode controller (also called the In-System Test, or IST, controller) that intercepts the TAP controller and run MemoryBIST tests. This test case inserts a direct memory access (DMA) MissionMode controller at the top level and a CPU-based MissionMode (MM) controller within the GPS baseband cores. Depending on your IJTAG network length, you might or might not need an in-system test controller at the core. A single DMA MissionMode controller could be sufficient to run MemoryBIST and other IJTAG tests.

DFT Insertion Flow

The following figure shows that Tessent Shell generates a TSDB for each core, and then at the top level, it uses the information stored in the core-level TSDBs for DFT integration.

Figure 6-5. DFT Insertion Flow for Automotive Test Case



For details about the TSDB data flow, refer to “TSDB Data Flow for the Tessent Shell Flow” on page 339.

Core-Level DFT Insertion for Automotive

The process for inserting DFT into hierarchical designs is performed in a bottom-up process starting from the lowest level blocks. This section describes how to perform the DFT insertion flow for the processor core and GPS baseband cores.

For general information about the RTL and scan DFT insertion flow for physical blocks, refer to “[RTL and Scan DFT Insertion Flow for Physical Blocks](#)” on page 146. (The section relates to a hierarchical test case; however, the basic flow remains the same.)

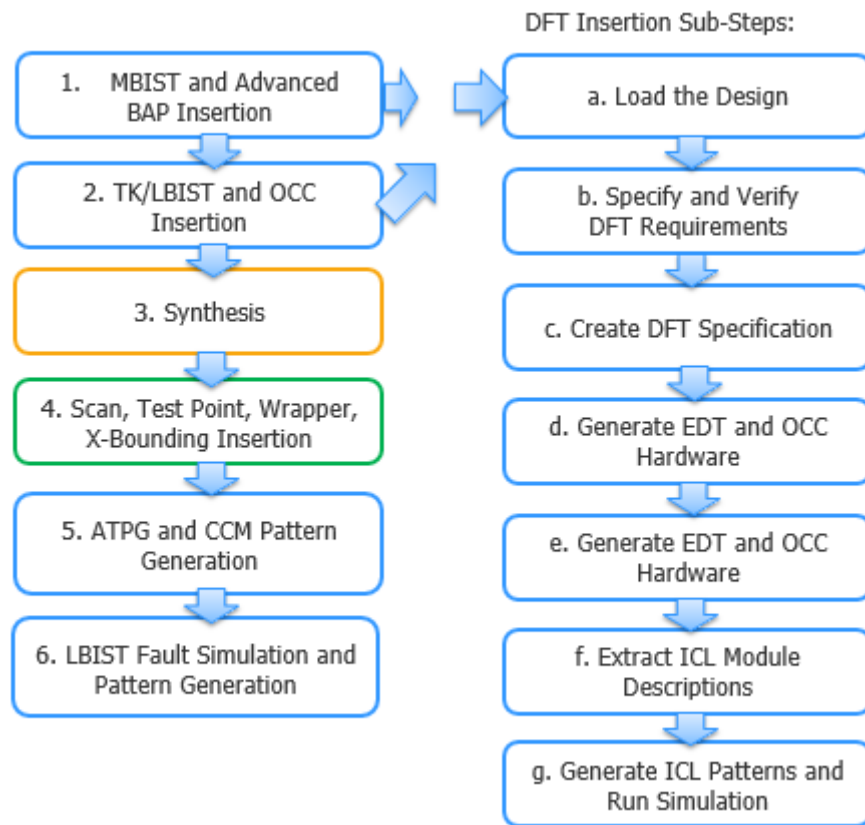
DFT Insertion Flow for the Processor Core Physical Block	277
First DFT Insertion Pass: processor_core	278
Second DFT Insertion Pass: processor_core	281
Test Point, X-Bounding, and Scan Insertion: processor_core	285
ATPG Pattern Generation: processor_core	289
LogicBIST Fault Simulation: processor_core	291
LogicBIST Pattern Generation: processor_core	292
Interconnect Bridge/Open UDFM Generation: processor_core	294
Cell-Neighborhood UDFM Generation: processor_core	296
Automotive-Grade ATPG Pattern Generation: processor_core	300
DFT Insertion Flow for the GPS Baseband Physical Block	307
DFT Insertion Pass With In-System Test: gps_baseband	308
LogicBIST Fault Simulation With One NCP: gps_baseband	312
Interconnect Bridge/Open UDFM Generation: gps_baseband	312
Cell-Neighborhood UDFM Generation: gps_baseband	313
Automotive-Grade ATPG Pattern Generation: gps_baseband	313

DFT Insertion Flow for the Processor Core Physical Block

In addition to the MemoryBIST logic, during the first DFT insertion pass, insert the advanced BAP and the built-in self-repair (BISR) IP, which includes built-in redundancy analysis (BIRA). You can control whether to test the memory in the same controller step or not, depending on memory power domains, test algorithms, and other design factors.

The advanced BAP enables certain feature overrides in the hardware default (hw_default) operating mode of the MemoryBIST controller and reduces test time by eliminating shift cycles to serially configure the controllers in the JTAG environment.

Figure 6-6. DFT Insertion Flow for processor_core




First DFT Insertion Pass: processor_core	278
Second DFT Insertion Pass: processor_core	281
Test Point, X-Bounding, and Scan Insertion: processor_core	285
ATPG Pattern Generation: processor_core	289
LogicBIST Fault Simulation: processor_core	291
LogicBIST Pattern Generation: processor_core	292
Interconnect Bridge/Open UDFM Generation: processor_core	294

Cell-Neighborhood UDFM Generation: processor_core	296
Automotive-Grade ATPG Pattern Generation: processor_core.....	300

First DFT Insertion Pass: processor_core

During the first DFT insertion pass, generate a default DFT specification for MemoryBIST that includes the BISR/BIRA, and then edit the DFT specification to include the additional connections for the advanced BAP.

Note


 The line numbers used in this procedure refer to the command flow dofile in [Example 6-1](#) on page 279.

For in-system tests, you can apply MemoryBIST during the power-on/off phase of the device operation or periodically while the device is operating. For details, refer to the [Tessent MissionMode User's Manual](#).

Procedure

1. Load the RTL design data and set the design parameters. (See lines 1-15.)

Note

 “rtl1” is the recommended naming convention for the design ID for the first insertion pass, but you can specify any name. Refer to “[Loading the Design](#)” for more information about setting the design ID.

2. Set the [set_dft_specification_requirements](#) command to “on” for -memory_bist. (See lines 17-18.)

When memory_test is on, memory_bist, memory_bisr_chains, and memory_bisr_controller default to auto; otherwise, they default to off. memory_bisr_chain auto changes to on for physical blocks having repairable memory, so the tool inserts a BISR chain with a BIRA engine created within the MemoryBIST controller.

3. Define the design clocks. (See lines 20-22.)
4. Check the design rules. (See lines 24.)
5. Create the DFT specification. (See lines 26-28.)
6. Edit the DFT specification to add the advanced BAP. (See lines 30-76.)

You can configure the connections to control the BAP directly through your system logic, thus bypassing the IJTAG network. Or, you can provide a way to control the advanced BAP as part of the IJTAG network. For this test case, implement the first strategy. When utilizing the advanced BAP feature, specify the options you plan to control directly using the ExecutionSelections wrapper in the DFT specification, and

then specify the BAP connections to the system logic. You can specify the connections within the DftSpecification wrapper or through a post-insertion procedure.

7. Generate the DFT hardware, JTAG network connectivity, and test patterns. (See lines 79-86.)
8. Run simulations to verify the design. (See lines 88-92.)

Examples

The following dofile example shows a typical automotive command flow for a core-level first DFT insertion. Modify the DFT specification for you design requirements.

Example 6-1. Dofile Example for First DFT Insertion Pass, processor_core

```
1 # Load the design
2 set_context dft -rtl -design_id rtl1
3 set_tsdb_output_directory ../tsdb_outdir
4 read_cell_library ../../lib/standard_cells/tessent/adk.tcelllib \
5 set_design_source -format tcd_memory -y ../../library/memories
6     -extension tcd_memory
7 read_verilog ../../library/memories/SYNC_1RW_32x16_RC.v
8     -interface_only -exclude_from_file_dictionary
9 read_verilog ../../library/memories/SYNC_1RW_8Kx16_RC.v
10    -interface_only -exclude_from_file_dictionary
11 read_verilog -f rtl_file_list
12
13 set_current_design processor_core
14
15 set_design_level physical_block
16
17 # Specify the DFT requirements
18 set_dft_specification_requirements -memory_test on
19
20 # Define memory clock
21 add_clocks 0 dco_clk -period 3
22 add_clocks 0 directclk -period 12
23
24 check_design_rules
25
26 # Create and report the DFT specification
27 set_spec [create_dft_specification]
28 report_config_data $spec
29
30 # Edit DFT specification to include additional advanced BAP connections
31 add_config_element DftSpecification(processor_core,rtl1)/MemoryBist/
32     BistAccessPort/DirectAccessOptions
33 set_config_value
34 DftSpecification(processor_core,rtl1)/MemoryBist/
35     BistAccessPort/DirectAccessOptions/direct_access_on
36 set_config_value
37 DftSpecification(processor_core,rtl1)/MemoryBist/
38     Controller(c1)/DirectAccessOptions/ExecutionSelections
39 set_config_value
40 DftSpecification(processor_core,rtl1)/MemoryBist/
41     Controller(c1)/DirectAccessOptions/algorithm on
```


```
42 set_config_value
43 DftSpecification(processor_core,rtl1)/MemoryBist/
44     Controller(c1)/DirectAccessOptions/operation_set on
45 set_config_value
46 DftSpecification(processor_core,rtl1)/MemoryBist/
47     Controller(c1)/DirectAccessOptions/memory on
48 set_config_value
49 DftSpecification(processor_core,rtl1)/MemoryBist/
50     Controller(c1)/DirectAccessOptions/step on
51 set_config_value
52 DftSpecification(processor_core,rtl1)/MemoryBist/Controller(c1)/
53     AdvancedOptions/extra_algorithms {SMARCHCHKKB SMARCHCHKBCI SMARCH
54     SMARCHCHKBCIL}
55
56 report_config_data $spec
57
58 # Specify BAP connection to system logic
59 read_config_data -in $spec/MemoryBist/BistAccessPort/Connections/
60     -from_string {
61     DirectAccess {
62         controller_select      : DBAP_control/ctrl_select ;
63         step_select            : DBAP_control/step_select ;
64         step_select_enable     : DBAP_control/step_select_en ;
65         algorithm_select       : DBAP_control/algo_select ;
66         algorithm_select_enable : DBAP_control/algo_select_en ;
67         operation_set_select   : DBAP_control/opset_select ;
68         operation_set_select_enable : DBAP_control/opset_select_en ;
69         ClockDomain (-) {
70             clock      : DBAP_control/clkout ;
71             reset      : reset_n ;
72             test_start : DBAP_control/start;
73             test_done  : DBAP_control/done ;
74             test_pass  : DBAP_control/pass ;
75         }
76     }
77 }
78
79 # Generate and insert the hardware
80 process_dft_specification
81
82 extract_icl
83
84 # Generate test bench verification patterns
85 create_patterns_specification
86 process_patterns_specification
87
88 # Run and check test bench simulations
89 set_simulation_library_sources -y ../../../../library/memories/
90     -extension v -v ../../../../library/standard_cells/verilog/adk.v
91 run_testbench_simulations
92 check_testbench_simulations
93
94 exit
```


Second DFT Insertion Pass: processor_core

In the second DFT insertion pass, insert the EDT, OCC, and LogicBIST instruments. Use the hybrid TK/LBIST process to share the EDT and LogicBIST IP. Control the clocking scheme for the LogicBIST pattern using a named capture procedure (NCP) index decoder, which decodes the NCP index output of the hybrid controller into clock sequences that are generated by Tessent OCCs across all capture procedures.

For details about OCC for the hybrid TK/LBIST flow, refer to “[Tessent OCC TK/LBIST Flow](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 6-2](#) on page 282 unless otherwise noted.

Procedure

1. [Loading the Design](#).
2. Follow the steps described in “[Specifying and Verifying the DFT Requirements: DFT Signals for Wrapped Cores](#)” on page 152, ensuring that you also include the required DFT signals for the hybrid TK/LBIST flow and for controller chain mode. (See lines 3-31.)
3. [Creating the DFT Specification](#) with SIBs for EDT, OCC, and LogicBIST, and edit the default specification to include the OCC, hybrid IP, and LogicBIST. (See lines 33-125.)

Use the `read_config_data` command to edit the DFT specification as follows:

- Specify the [OCC](#) wrapper and specify your clock intercept node for the OCC.

For details about OCC for the hybrid TK/LBIST flow, refer to “[Tessent OCC TK/LBIST Flow](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

- Specify the EDT wrapper to include a [LogicBistOptions](#) wrapper, and specify a MISR ratio of one. This reduces the chances of aliasing at the MISR (for better debugging capability) while trading off the area.

If you want to use the `edt_clock` as a shift clock source for LogicBIST, specify a dash (-) value in the interface wrapper, which caused the tool to not create a `shift_clock_src` port.

- Specify a [LogicBist](#) wrapper that contains both a Controller wrapper and an `NcpIndexDecoder` wrapper. Tessent Shell automatically converts the EDT controller into a hybrid TK/LBIST controller.

You must specify the clocking combinations to be used during TK/LBIST test. The tool synthesizes the NCP index decoder and generates named capture procedures based on this description. For details, refer to “[NCP Index Decoder](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

Note

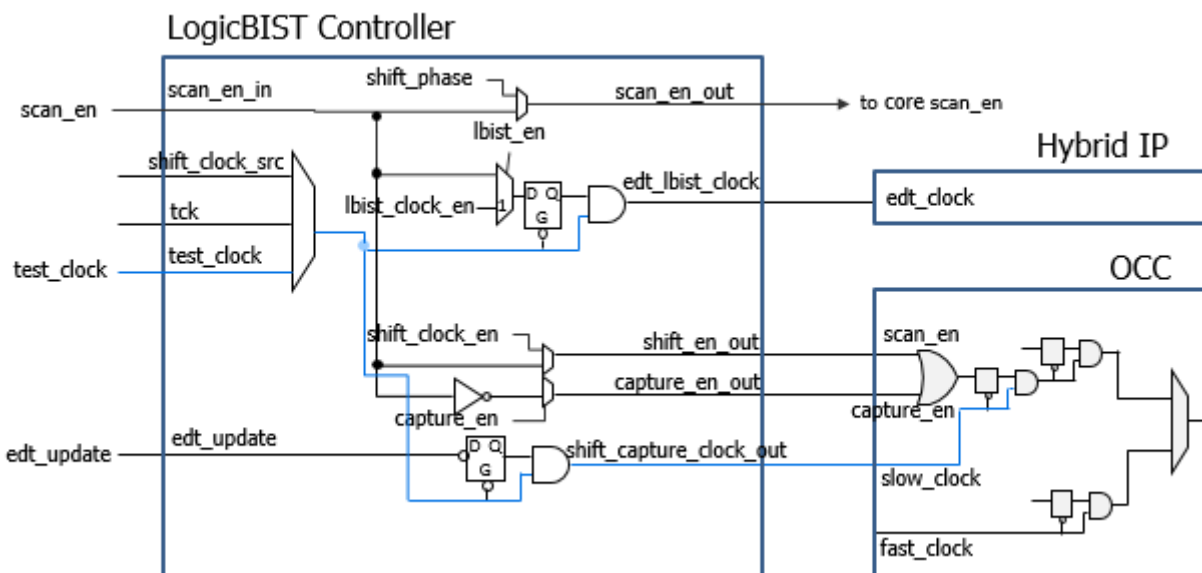
When you have only one NCP, the tool does not generate an NCP index decoder. You can use this configuration to implement a static clock sequence loaded through the ICL network. (You notice this for the GPS baseband block.)

4. [Generating the EDT, Hybrid TK/LBIST, and OCC Hardware](#), plus the LogicBIST hardware. (See line 125.)
5. [Extracting the ICL Module Description](#). (See line 127.)
6. [Generating ICL Patterns and Running Simulation](#). (See lines 129-140.)

Results

The following figure shows the clocking and DFT controls after the second DFT insertion pass.

Figure 6-7. Enhanced Clocking and DFT Controls After Second DFT Insertion Pass



Examples

The following dofile example shows a typical automotive command flow for a core-level second DFT insertion. Modify the DFT specifications for you design requirements.

Example 6-2. Dofile Example for Second DFT Insertion Pass, processor_core

```

1 // Load the design commands ...
2
3 set_dft_specification_requirements -logic_test on
4
5 # Add required DFT signals for logic test
6 add_dft_signals ltest_en -create_with_tdr
7 add_dft_signals scan_en edt_update test_clock
8     -source_node {scan_enable edt_update test_clock u }

```

```

9  add_dft_signals edt_clock shift_capture_clock -create_from_other_signals
10
11 # Add required DFT signals specific to hybrid TK/LBIST flow
12 add_dft_signals observe_test_point_en control_test_point_en x_bounding_en
13
14 # Add required DFT signal for MemoryBIST
15 # DFT signal used for scan-tested instruments such as MemoryBIST
16 add_dft_signals tck_occ_en
17
18 # Add DFT signals to bypass memories or run multi-load ATPG for memories
19 add_dft_signals memory_bypass_en
20 add_dft_signals mcp_bounding_en
21
22 # Add DFT signals required for hierarchical DFT (external, internal modes)
23 add_dft_signals int_ltest_en ext_ltest_en int_mode ext_mode
24
25 # Add DFT signal for controller chain mode
26 add_dft_signals controller_chain_mode
27
28 # Enable Observation Scan Technology (OST) for capturing per shift and
    capture
29 add_dft_signals capture_per_cycle_static_en
30
31 check_design_rules
32
33 # Create DFT specification
34 set spec [create_dft_specification -sri_sib_list {occ edt lbist } ]
35
36 Use report_config_syntax DftSpecification/edt|occ to see full syntax
37 report_config_data $spec
38
39 # Edit DFT specification to specify OCC's SIB and to insert OCC
40 read_config_data -in $spec -from_string {
41     OCC {
42         ijtag_host_interface : Sib(occ);
43         static_clock_control : external;
44     }
45 }
46
47 # Specify OCC insertion and intercept node
48 # This is a generic method for populating the OCC; modify for your design
49 # Scan enable and shift capture clock signals are automatically connected
50 # to the OCC instances
51 set id_clk_list [list \
52 dco_clk dco_clk \
53 directclk directclk
54 ]
55 foreach {id clk} $id_clk_list {
56 set occ [add_config_element OCC/Controller($id) -in $spec]
57 set_config_value clock_intercept_node -in $occ $clk
58 }
59
60 # Specify the hybrid EDT configuration
61 read_config_data -in $spec -from_string {
62     EDT {
63         ijtag_host_interface : Sib(edt);
64         Controller (c1) {
65             longest_chain_range : 100, 200 ;

```

```

66         scan_chain_count : 10;
67         input_channel_count : 1;
68         output_channel_count : 1;
69         LogicBistOptions {
70             misr_input_ratio : 1 ;
71             ShiftPowerOptions {
72                 present : on ;
73                 default_operation : disabled ;
74                 SwitchingThresholdPercentage {
75                     min : 25 ;
76                 }
77             }
78         }
79     }
80 }
81 }
82
83 # Specify the LogicBIST controller with NCP index decoder
84 read_config_data -in $spec -from_string {
85     LogicBist {
86         ijttag_host_interface : Sib(lbist);
87         Controller(1%ctrl_lbist) {
88             burn_in : on ;
89             pre_post_shift_dead_cycles : 8 ;
90             SingleChainForDiagnosis { Present : on ; }
91             ControllerChain {
92                 present : on;
93                 clock : tck;
94             }
95             Connections {
96                 scan_en_in : scan_enable ;
97                 controller_chain_scan_in : ccm_scan_in ;
98                 controller_chain_scan_out : ccm_scan_out ;
99             }
100            Interface {
101                shift_clock_src : - ;
102            }
103            ShiftCycles { max : 200 ; }
104            CaptureCycles { max : 7 ; }
105            PatternCount { max : 1024 ; }
106            WarmupPatternCount { max : 512 ; }
107        }
108
109        NcpIndexDecoder{
110            Ncp(dco_clk) {
111                cycle(0): OCC(dco_clk);
112                cycle(1): OCC(dco_clk);
113            }
114            Ncp(ALL) {
115                cycle(0): OCC(dcoo_clk);
116                cycle(1): OCC(directclk);
117            }
118            Ncp(sti_occ) {
119                cycle(0): processor_core_rtl1_tessent_sib_sti_inst ;
120            }
121        }
122    } // End of LogicBIST controller wrapper
123// End of report_config_data

```

```
124
125process_dft_specification
126
127extract_icl
128
129# Write script for design compilation
130write_design_import_script for_dc_synthesis.tcl -replace
131
132create_patterns_specification
133process_patterns_specification
134
135set_simulation_library_sources -v
136     ../../../../library/standard_cells/verilog/adk.v -y
137     ../../../../library/memories ../../../../library/standard_cells/verilog
138     -extension v
139
140run_testbench_simulations -simulator_option +nowarnTSCALE
141
142exit
```


Test Point, X-Bounding, and Scan Insertion: processor_core

After synthesizing the design using a third-party tool, perform test point analysis and insertion, X-bounding, wrapper analysis, and scan chain analysis and insertion.

- Test points — Inserting test points increases the testability of a design by improving controllability and observability during scan testing. This step is part of the hybrid TK/LBIST workflow; see “[Perform Test Point Insertion](#)” on page 241 for details.

In addition, a new type of observe point — the observation scan observe point (OP) — is inserted during test point insertion. The tool monitors observation scan OPs during every shift cycle and capture cycle compared with traditional OPs, which are monitored only during capture. Observation scan observe points are a part of observation scan technology (OST). For details, refer to “[Observation Scan Technology](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

Note


 For OST, you must add the capture_per_cycle_static_en DFT signal during the DFT insertion pass.

- X-bounding — X-bounding ensures that only valid binary values propagate through the scan cells during test. This prevents X sources from reaching the memory elements and corrupting the signature during LogicBIST. For details, refer to “[X-Bounding](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.
- Wrapper analysis — In hierarchical DFT, wrapper analysis prepares the functional scannable sequential elements (or “flops”) for reuse as wrapper cells. Use the [analyze_wrapper_cells](#) command.


- Scan chains — The tool inserts wrapper scan chains around the physical blocks that connects to each input and output. The input and output wrapper chains are exercised using the internal and external scan mode DFT control signals. For more information, see “[Performing Scan Chain Insertion: Wrapped Core](#)” on page 154.

In addition, you must configure the controller scan chains and add a scan mode for controller chain mode (CCM).

Note

 Tessent Shell works with any third-party scan insertion or other tools. However, because all Tessent products reside on the same code and database, you lose some automation with third-party tools.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 6-3](#) on page 287 unless otherwise noted.

Procedure

1. After loading the design and setting the design parameters, perform test point, X-bounding, wrapper, and scan insertion in the merged DFT context. (See lines 7.)
2. Read the design files and specify the requirements for test points and OST observe points. (See line 9-51.)
3. Post-test point insertion, specify the requirements for X-bounding. (See lines 60-70.)
4. Specify wrapper cell requirements and perform wrapper cell analysis. (See lines 72-77.)

For details, refer to “[Scan Insertion for Wrapped Core](#)” in the *Tessent Scan and ATPG User’s Manual*.

5. Activate controller chain segments. (See lines 79-82.)

By default, CCM scan segments in Tessent IP cores are not active to prevent them from being confused with standard scan elements. You must activate a CCM scan mode on Tessent IP instances before adding them to the scan mode population.

6. Specify the scan modes: internal, external, and controller chain. (See lines 84-94.)

For internal and external modes, connect the scan chains to the EDT block. For CCM mode, ensure that you only include CCM scan segments as valid scan elements.

7. Analyze scan chains and insert X-bounding, wrapper, and scan chain logic. (See lines 96-103.)

Examples

The following dofile shows a command flow for test point and scan insertion.

Example 6-3. Dofile Example for Test Point and Scan Insertion, processor_core

```

1  ##### Test Point, X-bounding, Wrapper, and Scan Insertion #####
2
3  # Open the previous TSDB directory if it is not in the current working
   directory
4  set_tsdb_output_directory ../tsdb_outdir
5
6  # Setting context to dft since inserting DFT into gate-level design
7  set_context dft -test_point -scan -no_rtl -design_id gate1
8
9  ##Read in the design (you may use -exclude_from_file_dictionary to
   exclude the netlist from the tessent database)
10 read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
11 # Read the synthesized netlist
12 read_verilog ../3.synthesize_rtl/processor_core_synthesized.vg
13
14 ##Read in the memory library model
15 set_design_sources -format tcd_memory -y ../../../../library/memories/
   -extension tcd_memory
16
17 ##Read in memory verilog model
18 read_verilog ../../../../library/memories/SYNC_1RW_32x16_RC.v
   -interface_only -exclude_from_file_dictionary
19 read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v -interface_only
   -exclude_from_file_dictionary
20
21 ##Read the design files from the RTL insertion pass
22 # Use the -no_hdl switch to skip reading the netlist as the synthesized
   netlist has already been read
23 read_design processor_core -design_id rtl2 -no_hdl
24
25 set_current_design processor_core
26
27 ##Setting the design level as physical_block
28 set_design_level physical_block
29
30 set_static_dft_signal_value ltest_en 1
31
32 report_static_dft_signal_settings
33
34 # Specify the number of testpoint (integer or integer %)
35 set_test_point_analysis_options -total_number 250
36
37 # Specify how many observe points to be observed by one scan cell
38 set_test_point_insertion_options -observe_point_share 5
39
40 # Specify both types of test points to reduce both pattern count
41 #   and improve random pattern testability
42 set_test_point_type { lbist_test_coverage edt_pattern_count }
43
44 # Specify capture per cycle observation for OST to observe per shift cycle
45 set_test_point_analysis_options -capture_per_cycle_observe_points on
46
47 # Setting the shift length to the anticipated scan chain length of the
   design for OST
48 set_test_point_analysis_options -minimum_shift_length 80

```

```
49
50 # Set mcp_bounding_en to 1 to gate BIST.CLK and prevents additional
    hardware bounding logic
51 set_static_dft_signal_values mcp_bounding_en 1
52
53 # Check design rule
54 set_system_mode analysis
55
56 # Report clocks and dft signals
57 report_clocks
58 report_dft_signals
59
60 # Analyze test points
61 analyze_test_points
62 write_test_point_dofile processor_core_tpi.dofile -all -replace
63
64 ## Exclude reset from X-bounding to avoid XB DRC. Handled by wrapper
    insertion. Automation in 19.2
65 set_xbounding_options -exclude {reset_n}
66
67 ## Perform X-bounding for LBIST
68 analyze_xbounding
69 report_xbounding -verbose > xbound_list
70 report_xbounding -verbose -ignored_x_sources on >
    xbound_list_with_ignored_x_source
71
72 # Exclude the edt_channel in and out ports from wrapper chain analysis.
    The ijtag_* edt_update ports are automatically excluded
73 set_wrapper_analysis_options -exclude_ports [ get_ports
    {*_edt_channels_*} ]
74
75 # Perform wrapper cell analysis
76 analyze_wrapper_cells
77 report_wrapper_cells -Verbose > wrapper_cell.rpt
78
79 # Set attribute value for in-built chains for controller_chain_mode
80 set_attribute_value [get_instances *tessent_single_chain_mode_logic_*]
    -name active_child_scan_mode -value controller_chain_mode
81 set_attribute_value [get_instances *tessent_lbist_inst] -name
    active_child_scan_mode -value controller_chain_mode
82 set_attribute_value [get_instance -of_module *_edt_lbist_c1] -name
    active_child_scan_mode -value controller_chain_mode
83
84 # Specify different modes (internal and external) how the chains need to
    be stitched
85 # The type internal/external and enable_dft_signal are inferred from
    registered DFT Signals(int_mode and ext_mode)
86 # Create a scan mode and specify EDT instance to connect the scan chains
    to
87 set ccm [get_scan_elements -of_child_scan_mode controller_chain_mode]
88 set core [remove_from_collection [get_scan_elements] $ccm]
89 #set core [remove_from_collection [get_scan_elements -class core] $ccm]
90 set edt_instance [get_instances -of_icl_instances [get_icl_instances
    -filter tessent_instrument_type==mentor::edt]]
91
92 add_scan_mode int_mode -edt $edt_instance -include_elements $core
    -enable_dft_signal int_mode
```

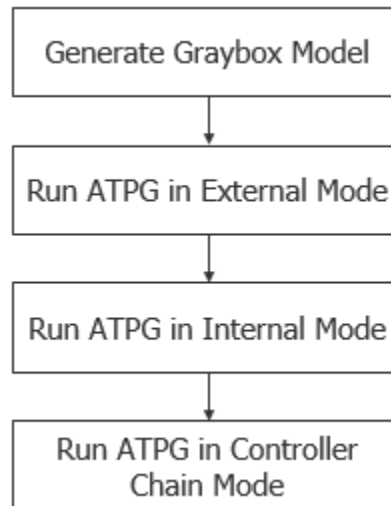


```
93 add_scan_mode controller_chain_mode -include_elements $ccm -chain_count 1
   -enable_dft_signal controller_chain_mode -si_port_format ccm_scan_in%d
   -so_port_format ccm_scan_out%d
94 add_scan_mode ext_mode -chain_count 2
95
96 # Analyze the scan chains and review the different scan modes and chains
   before stitching the chains
97 analyze_scan_chains
98
99 # Insert scan chains and writes the scan inserted design into tsdb_outdir
   directory
100 insert_test_logic
101
102 report_scan_chains
103 report_scan_cells > scan_cells.list
```

ATPG Pattern Generation: processor_core

Generating ATPG patterns for a wrapped core includes a step for generating a graybox model. In addition, you must perform ATPG twice, once for the core's external mode and once for the core's internal mode, which is typical for any hierarchical design. Controller chain mode, when used, also requires its own ATPG pass.

Figure 6-8. ATPG Pattern Generation Flow for processor_core



For information about graybox models, refer to “[Graybox Model](#)” on page 100.

Procedure

Perform ATPG as described in “[Performing ATPG Pattern Generation: Wrapped Core](#)” on page 158 with the following extra considerations:

- a. After running ATPG on the internal mode and saving the collateral data to the TSDB with the `write_tsdb_data` command (step 3-d), do the following:
 - i. Return to setup mode and turn off the `memory_bypass_en` signal.
 - ii. Read the PDL of the BISR chains that have the `init_bisr_chains` iProc. The procedure is located at:

```
/tsdb_outdir/instruments/  
processor_core_rtl1_mbisr.instrument/  
processor_core_rtl1_tessent_mbisr.pdl
```

Invoke the procedure as follows:

```
set_test_setup_icall init_bisr_chains -front
```

- iii. Run ATPG on the internal mode again and save the data with the `write_tsdb_data` command.

ATPG generates a multi-load pattern that can scan through the memory. Multi-load scan patterns are used to generate scan patterns through ROM and RAM memories. Before generating multi-load patterns on repairable memories, any repair information that was previously generated by the execution of MemoryBIST must be applied to the memory repair ports.

For details, refer to “[Running Multi-Load ATPG on Wrapped Core Memories with Built-In Self Repair](#)” on page 254.

- iv. Use the `read_faults` command to merge the fault list from running external mode to find the total overall fault coverage of the wrapped core.
- b. Run ATPG on the controller chain mode to create ATPG patterns for the following test logic IPs: hybrid controller, LogicBIST controller, single chain mode logic, and in-system test controller. Do the following:
 - i. Load the wrapped cores that contain the child wrapped cores.
 - ii. If you used Tessent Scan for scan insertion, specify “`import_scan_mode controller_chain_mode`” to import the controller chain mode.
 - iii. Specify a unique ATPG mode name for controller chain mode, such as `ccm`. For example:

```
set_current_mode ccm
```

- iv. After running DRC, generate the ATPG patterns, and store the TCD, flat model, fault list and PatDB files in the TSDB using the `write_tsdb_data` command.

The generated ATPG patterns are core-level patterns. You generate these patterns again at the top level.


- v. Use the `read_faults` command to merge the fault list from running internal mode to find the total overall fault coverage of the wrapped core.

LogicBIST Fault Simulation: processor_core

Perform LogicBIST fault simulation to generate pseudo-random, parallel pattern sets. Tessent Shell chooses the PRPG seed value based on the specified warm-up pattern and creates the signature for the MISR. The signature is stored in the TSDB. Specify the number of random patterns based on your in-system requirements for test time and LogicBIST coverage.

For details about seeding, refer to “[Warm-Up Patterns](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

Note

 The line numbers used in this procedure refer to the command flow in [Example 6-4](#) on page 292. Refer to “[Performing LogicBIST Fault Simulation](#)” on page 249 for additional details about some of the following steps.

Procedure

1. [Loading the Design](#). (See lines 1-10.)
2. Set the current test mode to a unique name for the new pattern set you create to test the hybrid IP. (See line 12.)
3. Import the core’s internal mode data—that is, the scan-inserted design data for the EDT and OCC logic. (See line 14.)

Using the `import_scan_mode` command assumes that you used Tessent Scan to perform scan chain stitching.
4. Add the hybrid TK/LBIST core instances. (See line 16.)

You must explicitly add the LogicBIST controller core.
5. Specify the capture procedure names and the count percentage to repeat the NCP once every 256 patterns. (See lines 18-19.)
6. Specify the order of the NCPs. (See lines 21-25.)
7. Read in the NCP testproc file. (See lines 29-32.)
8. Add the faults and specify the maximum number of pseudo-random patterns you want the tool to simulate. (See lines 34-35.)
9. Set the pattern source to LogicBIST and execute fault simulation. (See line 36.)

10. Write out the parallel test bench and save the data into the TSDB. (See lines 38-40.)

Examples

The following dofile example shows a typical command flow for LogicBIST fault simulation.


Example 6-4. Dofile Example for LogicBIST Fault Simulation, processor_core

```
1 set_context patterns -scan -design_id gate2
2 set_tsdb_output_directory ../tsdb_outdir
3 open_tsdb ../tsdb_outdir
4 read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
5 read_cell_library ../../../../library/memories/SYNC_8X16.atpglib
6 read_cell_library ../../../../library/memories/SYNC_1RW_32x16_RC.atpg
7
8 read_design processor_core -design_id gate1
9
10 set_current_design processor_core
11
12 set_current_mode lbist_sa -type internal
13
14 import_scan_mode int_mode
15
16 add_core_instances -module *tessent_lbist
17
18 set_lbist_controller_options -capture_procedures {ALL 90 directclk 8
19     sti_occ 2 }
20
21 # Specify the full pathname to the dofile located in the
22 # *_lbist_ncp_index_decoder.instrument directory in the TSDB
23 dofile ../tsdb_outdir/instruments/processor_core_rtl2
24     _lbist_ncp_index_decoder.instrument/processor_core_rtl2_tessent
25     _lbist_ncp_index_decoder.dofile
26
27 set_system_mode analysis
28
29 # Specify the full pathname to the testproc file located in the
30 # *_lbist_ncp_index_decoder.instrument directory in the TSDB
31 read_procfile ../tsdb_outdir/instruments/processor_core_rtl2
32     _lbist_ncp_index_decoder.instrument/processor_core_rtl2
33
34 add_faults -all
35 set_random_pattern 1024
36 simulate_patterns -source bist -store_patterns all
37
38 write_patterns patterns/processor_core_lbist_parallel.v -
39 verilog -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
40     write_tsdb -replace
```

LogicBIST Pattern Generation: processor_core

To program the LogicBIST controller, use the pattern specification in hardware default mode to generate test bench and pattern range-specific vectors.

Note

 The line numbers used in this procedure refer to the command flow in [Example 6-5](#) on page 293. Refer to “[Perform LogicBIST Pattern Generation](#)” on page 251 for additional details about some of the steps in the following.

Procedure

1. [Loading the Design](#), ensuring that you set the context to “patterns -ijtag”. (See lines 1-7.)

The design ID should be the same design you used for LogicBIST fault simulation.

2. Create the patterns specification. (See lines 15-16.)
3. Edit the patterns specification to specify the LogicBIST pattern configuration. (See lines 18-52.)

If you want to debug Xs in your MISR during simulation, you can enable debugging with the `logic_bist_debug` property.

4. Generate the IJTAG patterns for LogicBIST and run simulation. (See lines 55-63.)

Examples

The following dofile shows a command flow for generating IJTAG patterns for LogicBIST in the automotive Tessent Shell flow.

Example 6-5. Dofile Example for LogicBIST Pattern Generation, processor_core


```
1 set_context patterns -ijtag
2 set_tsdb_output_directory ../tsdb_outdir
3 read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
4 # read memory library models too
5 read_design processor_core -design_id gate2
6
7 set_current_design piccpu
8
9 # Report clock and DFT signal settings
10 report_static_dft_signal_settings
11 report_clocks
12
13 set_system_mode analysis
14
15 # Create patterns specification
16 set patt_spec [ create_patterns_specification ]
17
18 # Edit the patterns specification for LogicBIST pattern
19 # Example only; modify for your design requirements
20 read_config_data -replace -in $patt_spec -from_string {
21     AdvancedOptions {
22         ConstantPortSettings {
23             scan_enable : 0;
24         }
25     }
```

```
26     Patterns(ICLNetwork) {
27         ICLNetworkVerify(processor_core) {
28             }
29     }
30     Patterns(LogicBist_processor_core) {
31         ClockPeriods {
32             dco_clk : 3.00ns;
33             directclk: 12.00ns;
34             test_clock_u: 100.00ns;
35         }
36         SimulationOptions {
37 # You can opt to enable debugging Xs in the MISR during simulation
38             logic_bist_debug : off;
39         }
40         TestStep(serial_load) {
41             LogicBist {
42                 CoreInstance(.) {
43                     run_mode : run_time_prog;
44                     mode_name : lbist_sa ;//same as set_current_mode in
45                     lbist fault simulation
46                     begin_pattern : 0;
47                     end_pattern : 255 ;
48                     misr_compares : 1 ;
49                 }
50             }
51         }
52     }
53 }
54
55 process_patterns_specification
56
57 set_simulation_library_sources -y ../../../../library/memories/ -extension v
58     -v ../../../../library/standard_cells/verilog/adk.v
59
60 run_testbench_simulation -simulator_options { -voptargs="+acc" }
61
62 # Use the following command if simulation fails
63 check_testbench_simulations -report_status
64
65 exit
```

Interconnect Bridge/Open UDFM Generation: processor_core

If you want to perform Automotive-Grade ATPG targeting interconnect bridge and open defects, generate a UDFM for the interconnect bridge/open defects in your design. Then, read in the UDFM during Automotive-Grade ATPG.

Note


 The line numbers used in this procedure refer to the command flow in “[Dofile Example for Interconnect Bridge/Open UDFM Generation, processor_core](#)” on page 295. Refer to the following directory, which has a usage example described in this section:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/processor_core/  
11.extract_fault_sites
```

Procedure

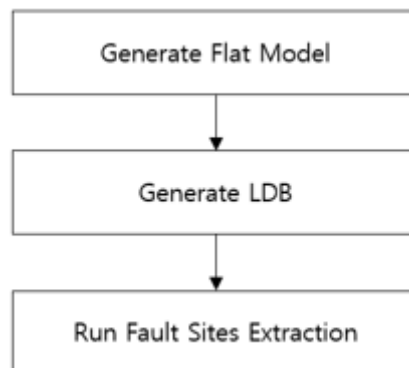
1. Load the post-layout design and generate a flat model. (See lines 1-19.)
2. Load the flat model. (See lines 21–23.)
3. Read in the layout data and generate an LDB. (See lines 25-34.)
4. Load the LDB by applying the `open_layout` command, and then apply the `extract_fault_sites` command with an output file name to generate a UDFM. (See lines 36-40.)

Note

 Use “all” as the `defect_types` option for the `extract_fault_sites` command to write out both bridges and opens to the UDFM file.

Examples

Figure 6-9. Interconnect Bridge/Open UDFM Generation Flow for processor_core



The following dofile shows a command flow for generating a bridge/open UDFM for use in the Automotive-Grade ATPG flow.

Example 6-6. Dofile Example for Interconnect Bridge/Open UDFM Generation, processor_core

```
1 set_context patterns -scan  
2 set design_name "processor_core"  
3
```


```
4 # Read Tessent Library
5 read_cell_library ../../../../library/
  NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_End/DFT/
  NangateOpenCellLibrary.tcelllib
6
7 # Read in the post layout netlist
8 read_verilog ../9.pnr/pnr_db/golden/${design_name}.post.vg
9
10 #Read in the memory library model
11 read_cell_library ../../../../library/memories/SYNC_1RW_8Kx16.atpglib
12 read_cell_library ../../../../library/memories/SYNC_1RW_32x16_RC.atpg
13 set_design_sources -format tcd_memory -y ../../../../library/memories/
  -extension tcd_memory
14
15 set_current_design $design_name
16
17 set_system_mode analysis
18
19 write_flat_model flat/${design_name}_post.flat -rep
20
21 set_context patterns -scan_diagnosis
22
23 read_flat_model flat/${design_name}_post.flat
24
25 set deffile [glob -directory ../9.pnr/pnr_db/golden ${design_name}.def]
26
27 set leffile [concat [glob -directory ../../../../library/
  NangateOpenCellLibrary_PDKv1_3_v2010_12/Back_End/lef
  NangateOpenCellLibrary.macro.lef] \
28 [glob -directory ../../../../library/
  NangateOpenCellLibrary_PDKv1_3_v2010_12/Back_End/lef
  NangateOpenCellLibrary.tech.lef] \
29 [glob -directory ../../../../library/memories
  SYNC_1RW_8Kx16.lef] \
30 [glob -directory ../../../../library/memories
  SYNC_1RW_32x16_RC.lef] \
31 ]
32
33 analyze_layout_hierarchy hier_db/${design_name}.hierdb -leffile $leffile
  -deffile $deffile -rep
34 create_layout ldb/${design_name}.ldb -rep -deffile $deffile -leffile
  $leffile
35
36 open_layout ldb/${design_name}.ldb
37
38 extract_fault_sites -output_file udfm/${design_name}_brdg_opn.udfm
  -defect_types all -rep
39
40 exit
```

Cell-Neighborhood UDFM Generation: processor_core

If you want to perform Automotive-Grade ATPG targeting cell-neighborhood defects, which may be located between cells, generate a UDFM for your design's cell-neighborhood defects. Use that UDFM during Automotive-Grade ATPG.

To generate a cell-neighborhood UDFM, extract cell-neighborhood pairs from the design LDB, then feed the cell pairs information into Tessent CellModelGen. Combine the UDFM files for all individual cell pairs into a single UDFM file. The required input data is the same as the data required for generating a cell-aware UDFM on standard cells. For information on Tessent CellModelGen or input requirements, refer to the *Tessent CellModelGen Tool Reference*.

Note

 The line numbers used in this procedure refer to the command flow in “[Dofile Example for Extracting Cell-Neighborhood Pairs, processor_core](#)” on page 298. Refer to the following directory, which has a usage example described in this section:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/processor_core/  
11.extract_fault_sites
```

Procedure

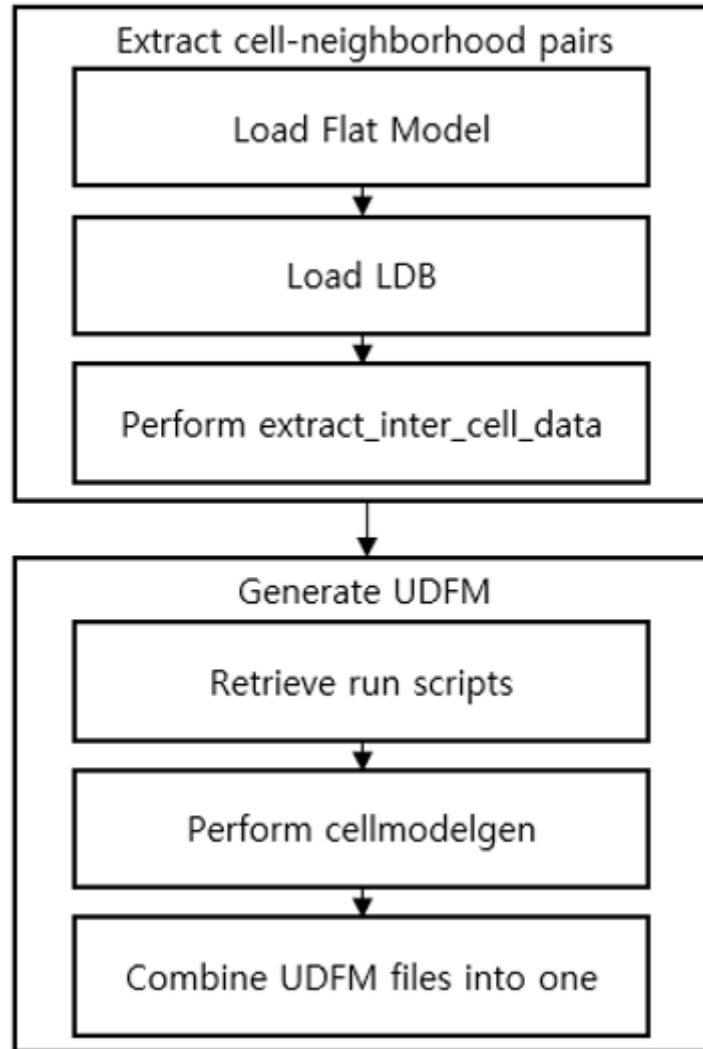
1. Extract cell-neighborhood pairs.
 - a. Load the flat model generated for interconnect bridge/open UDFM generation. (See lines 1-5.)
 - b. Load the LDB using the [open_layout](#) command, which is generated for interconnect bridge/open UDFM generation. (See line 7.)
 - c. Run the [extract_inter_cell_data](#) command, specifying an output file name to write the extracted cell-neighborhood pairs. (See line 9.)
2. Generate the UDFM using Tessent CellModelGen with the extracted cell-neighborhood pairs.
 - a. Before running Tessent CellModelGen, ensure that all the required input data is available and all the required tools are accessible.
 - b. Create an empty directory, and retrieve run scripts.


```
cellmodelgen -get_script all
```
 - c. Copy the *run_flow.merge* script into your working directory from the *lib/scripts* directory, which you get from step **b**.
 - d. Modify the *run_flow.merge* script for your design. Refer to the comments in the script for details. The output file from step **1**, the extracted inter-cell pairs, is the input to the *run_flow.merge* script.
 - e. Run the *run_flow.merge* script to run Tessent CellModelGen with the cell-neighborhood pairs. Specify the filename of the extracted cell-neighborhood pairs and a working directory name.
 - f. Copy the *run_export.merge* script into your working directory from the *lib/scripts* directory, which you get from step **b**.

- g. Modify the `run_export.merge` script for your working environment.
- h. Run the `run_export.merge` script to combine all single UDFM files on each of the cell pairs into a single UDFM file.

Examples

Figure 6-10. Cell-Neighborhood UDFM Generation Flow for processor_core



The following dofile shows a command flow for generating a bridge/open UDFM for use in the Automotive-Grade ATPG flow.

Example 6-7. Dofile Example for Extracting Cell-Neighborhood Pairs, processor_core

```
1 set_context patterns -scan_diagnosis
2
3 set design_name "processor_core"
4
```

```
5 read_flat_model flat/${design_name}_post.flat
6
7 open_layout ldb/${design_name}.ldb
8
9 extract_inter_cell_data -output_file inter_cell_data/
  ${design_name}_cellpairs.data -rep
10
11 exit
```

Automotive-Grade ATPG Pattern Generation: processor_core

Generating automotive-grade ATPG patterns is similar to regular ATPG pattern generation. It needs several UDFM files for cell-internal defects, interconnect bridge or open defects, and cell-neighborhood defects.

Refer to [Interconnect Bridge/Open UDFM Generation: processor_core](#) for interconnect bridge/open UDFM generation on your design, and [Cell-Neighborhood UDFM Generation: processor_core](#) for cell-neighborhood UDFM generation for your design.

Refer to “[UDFM Generation for Cell-Aware ATPG](#)” on page 329 for details on how to generate the cell-aware UDFM for your technology library.


As you do for regular ATPG, you must generate a graybox model; then, you must perform ATPG twice: once for the core’s external mode and once for its internal mode. When you run ATPG in external mode or internal mode, you can run ATPG with toff runs by loading only the UDFM files you want to target during an ATPG run. Also, you can run ATPG all together by reading in all UDFM files at once.

Automotive-Grade Topoff ATPG Pattern Generation	300
Automotive-Grade ATPG Pattern Generation Using Only UDFMs	304

Automotive-Grade Topoff ATPG Pattern Generation

This procedure describes how to run cell-aware ATPG toff when you already have ATPG patterns from previous runs.

Note

 The line numbers used in this procedure refer to the command flow in “[Dofile Example for Interconnect Bridge/Open UDFM Generation, processor_core](#)” on page 295. Refer to the following directory, which has a usage example described in this section:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/processor_core/  
12.generate_AGA_patterns
```

Prerequisites

- Existing patterns from previous ATPG runs.
- Existing UDFM file(s) for your technology library and design.

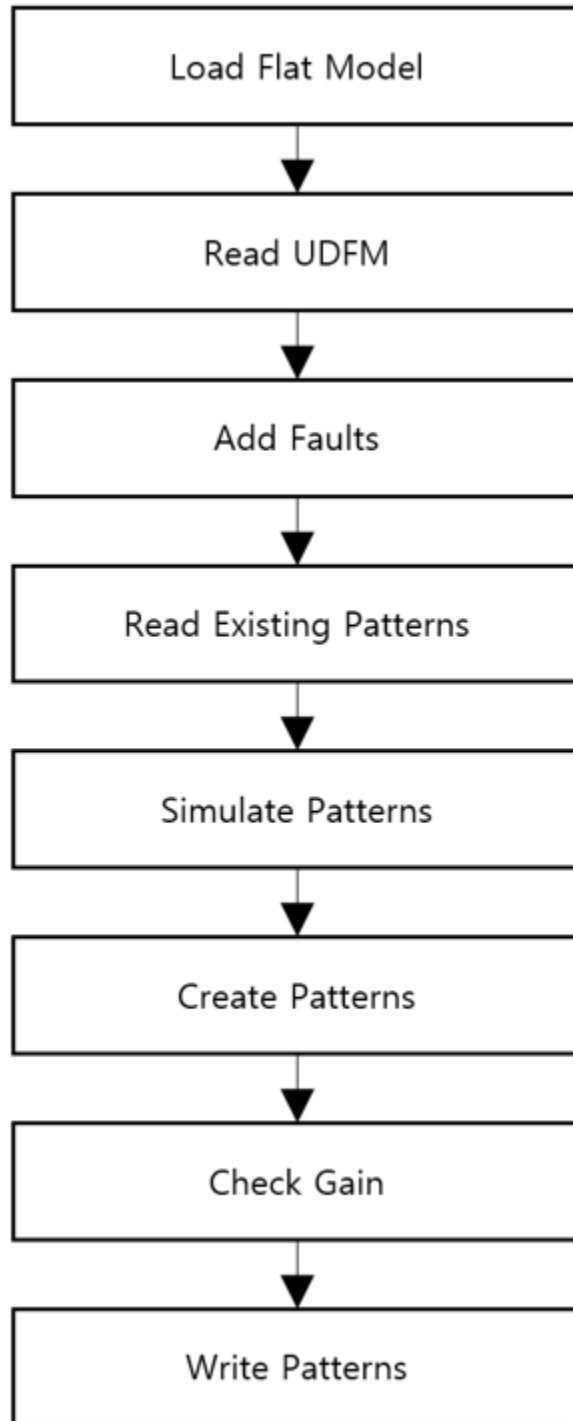
Procedure

1. Load the flat model generated from a post-layout design. (See lines 1–11.)
2. Set a fault type, read in the UDFM file for your standard cells, and add faults. (See lines 14-18.)

3. Read patterns from previous ATPG runs. (See line 20.)
4. Simulate the patterns to see the baseline test coverage. (See line 21.)
5. Create patterns. (See line 28.)
6. Check how much benefit there is with the newly-created patterns (see line 31). Set a baseline to see the benefit after step 4. (See line 24.)
7. Write the newly-created patterns. (See lines 34–40.)
8. Check final ATPG test coverage. (See lines 45-48)

Examples

Figure 6-11. ATPG Topoff Run Flow With a UDFM for processor_core



The following dofile shows a command flow for generating Automotive-Grade ATPG flow topoff patterns.

Example 6-8. Dofile Example for Running Topoff ATPG With a UDFM in Internal Mode for processor_core

```

1  set_context patterns -scan -design_id pnr
2  set design_name "processor_core"
3
4  # Specify the location of TSDB. Default is the current working directory
5  set_tsdb_output_directory ../tsdb_outdir
6
7  # Add more processors to run ATPG
8  add_processors localhost:4
9
10 # Read flat model
11 read_flat_model ../tsdb_outdir/logic_test_cores/
    ${design_name}_pnr.logic_test_core/${design_name}.atpg_mode_ATPG_int/
    ${design_name}_ATPG_int.flat.gz
12
13 # Cell-Aware Test Pattern Generation
14 set_fault_type udfm -static_fault
15 read_fault_sites ../../../../udfm_gen/stdlib/udfm/
    NangateOpenCellLibrary.udfm
16 report_fault_sites -undefined_cells
17 add_fault_sites -all
18 add_faults -all
19
20 read_patterns ../tsdb_outdir/logic_test_cores/
    ${design_name}_pnr.logic_test_core/${design_name}.atpg_mode_ATPG_int/
    ${design_name}_ATPG_int_stuck.patdb
21 simulate_patterns -source external
22
23 # Set baseline
24 report_udfm_statistics -set_baseline -group *OAI* *AOI* *OR* *AND* *INV*
    *BUF* *HA* *DFF* *MUX* *DLL*
25 report_statistic
26
27 # Generate patterns
28 create_patterns
29
30 # Check gain
31 report_udfm_statistics -group *OAI* *AOI* *OR* *AND* *INV* *BUF* *HA*
    *DFF* *MUX* *DLL*
32 report_statistic
33
34 write_patterns patterns/${design_name}_int_CAT_static_topoff.stil.gz -
    stil -rep
35 write_faults faults/${design_name}_int_CAT_static_topoff.faults.gz -rep
36
37 # Write Verilog patterns for simulation
38 write_patterns patterns/${design_name}_int_CAT_static_topoff_parallel.v -
    verilog -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
39 set_pattern_filtering -sample_per_type 2
40 write_patterns patterns/${design_name}_int_CAT_static_topoff_serial.v -
    verilog -serial -replace -parameter_list {SIM_KEEP_PATH 1}
41
42 # To understand the coverage of the faults testable by Internal mode it is
    necessary to


```

```
43 # eliminate the undetected faults that would otherwise be detected in
    External mode. This is done
44 # by merging the fault list from running the graybox in External mode
45 read_faults faults/${design_name}_ext_CAT_static_topoff.faults.gz -merge
46
47 # Final coverage of the core that includes both Internal and External
    modes
48 report_stat -detail
49
50 exit
```

Automotive-Grade ATPG Pattern Generation Using Only UDFMs

This procedure describes how to run cell-aware ATPG using only existing UDFMs.

Note

 The line numbers used in this procedure refer to the command flow in “[Dofile Example for Running ATPG With Only UDFMs in Internal Mode for processor_core](#)” on page 305.

Refer to the following directory, which has a usage example described in this section:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/processor_core/
12.generate_AGA_patterns
```

Prerequisites

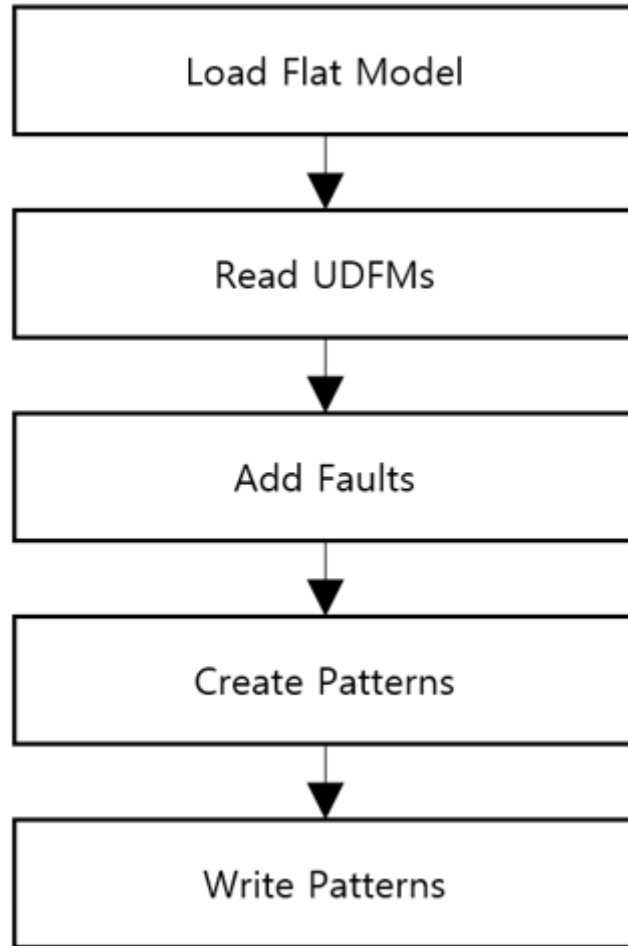
- Existing UDFM file(s) for your technology library and design.

Procedure

1. Load the flat model generated from a post-layout design. (See lines 1–11.)
2. Set a fault type and read in the UDFM files for your standard cells, interconnect bridge/open, and cell-neighborhood defects. Then add faults. (See lines 14-20.)
3. Create patterns. (See line 24.)
4. Write the patterns. (See line 27.)

Examples

Figure 6-12. ATPG Run Flow With All UDFM for processor_core



The following dofile shows a command flow for generating Automotive-Grade ATPG flow patterns using only UDFMs.

Example 6-9. Dofile Example for Running ATPG With Only UDFMs in Internal Mode for processor_core

```

1 set_context patterns -scan -design_id pnr
2 set_design_name "processor_core"
3
4 # Specify the location of TSDB. Default is the current working directory
5 set_tsdb_output_directory ../tsdb_outdir
6
7 # Add more processors to run ATPG
8 add_processors localhost:4
9
10 # Read flat model
11 read_flat_model ../tsdb_outdir/logic_test_cores/
    ${design_name}_pnr.logic_test_core/${design_name}.atpg_mode_ATPG_int/
    ${design_name}_ATPG_int.flat.gz
  
```

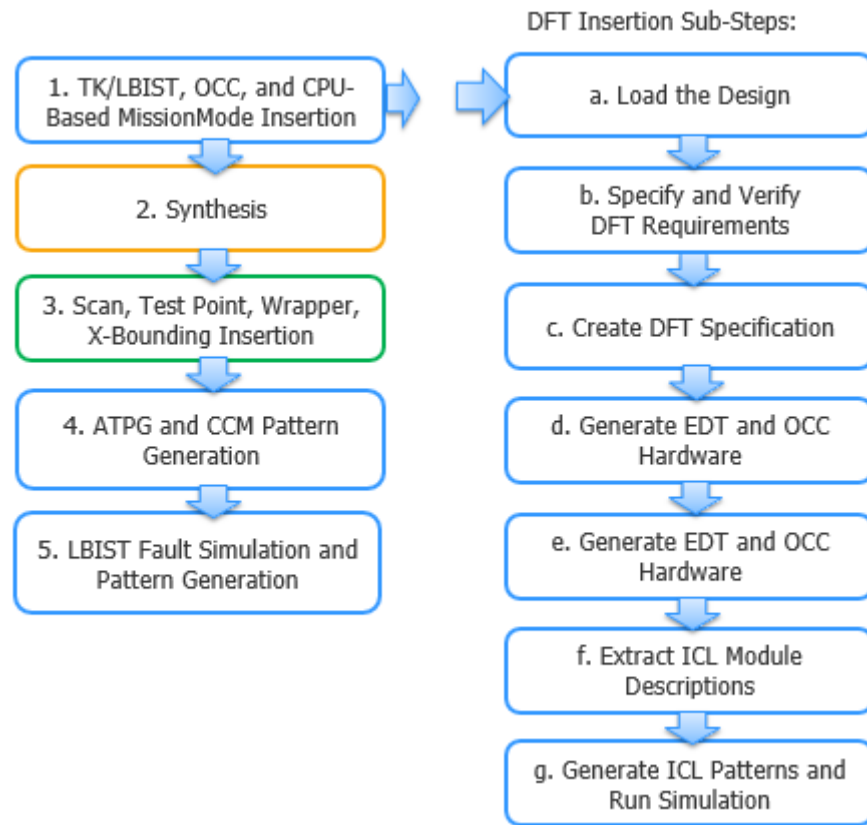
```
12
13 # Cell-Aware Test Pattern Generation
14 set_fault_type udfm -static_fault
15 read_fault_sites ../../../../udfm_gen/stdlib/udfm/
  NangateOpenCellLibrary.udfm
16 read_fault_sites ../11.extract_fault_sites/udfm/
  ${design_name}_brdg_opn.udfm
17 read_fault_sites ../11.extract_fault_sites/udfm/CELLS_neighbor.udfm
18 report_fault_sites -undefined_cells
19 add_fault_sites -all
20 add_faults -all
21 report_statistics -detail
22
23 # Generate patterns
24 create_patterns
25 report_statistics -detail
26
27 write_patterns patterns/${design_name}_int_AGA_static.stil.gz -stil -rep
28 write_faults faults/${design_name}_int_AGA_static.faults.gz -rep
29
30 # Write Verilog patterns for simulation
31 write_patterns patterns/${design_name}_int_AGA_static_parallel.v -verilog
  -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
32 set_pattern_filtering -sample_per_type 2
33 write_patterns patterns/${design_name}_int_AGA_static_serial.v -verilog
  -serial -replace -parameter_list {SIM_KEEP_PATH 1}
34
35 # To understand the coverage of the faults testable by Internal mode it is
  necessary to
36 # eliminate the undetected faults that would otherwise be detected in
  External mode. This is done
37 # by merging the fault list from running the graybox in External mode
38 #read_faults -mode ATPG_ext -fault_type udfm -merge
39 read_faults faults/${design_name}_ext_AGA_static.faults.gz -merge
40
41 # Final coverage of the core that includes both Internal and External
  modes
42 report_stat -detail
43
44 exit
```

DFT Insertion Flow for the GPS Baseband Physical Block

The GPS baseband design does not include memory. Since MemoryBIST is not required, you can skip the first DFT insertion pass as described for the processor core. Instead, perform one DFT pass to insert the hybrid IP, OCC, and in-system test.

In addition to illustrating how to insert in-system test at the block level, the flow for this block shows you how to perform LogicBIST fault simulation when a design only has one clock, and, thus, no NCP decoder logic.

Figure 6-13. DFT Insertion Flow for gps_baseband




This section highlights aspects of the flow for gps_baseband that differ from processor_core: DFT insertion and LogicBIST fault simulation. Refer to the test case for dofile details.

DFT Insertion Pass With In-System Test: gps_baseband	308
LogicBIST Fault Simulation With One NCP: gps_baseband	312
Interconnect Bridge/Open UDFM Generation: gps_baseband	312
Cell-Neighborhood UDFM Generation: gps_baseband	313
Automotive-Grade ATPG Pattern Generation: gps_baseband	313

DFT Insertion Pass With In-System Test: gps_baseband

This DFT insertion pass illustrates how to insert the in-system test controller at the core level. This could be a requirement for your design depending on the length of your IJTAG network. The tool generates a CPU-based controller, which is enabled from the system logic at the parent level.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 6-10](#) on page 309 unless otherwise noted.

Procedure

1. [Loading the Design](#). (See lines 1-8.)
2. Follow the steps described in “[Specifying and Verifying the DFT Requirements: DFT Signals for Wrapped Cores](#)” on page 152, ensuring that you also include the required DFT signals for the hybrid TK/LBIST flow and for controller chain mode. (See lines 10-35.)

Constrain the enable signal for the in-system test controller to 0 for manufacturing test. The tool emulates the control of this port during in-system test. (See lines 25-26.)

For the DFT signals, create a new port if a source node does not exist. The ports are created only if `set_design_level` is not chip.

3. [Creating the DFT Specification](#) with SIBs for EDT, OCC, and LogicBIST, and edit the default specification to insert the OCC, hybrid IP, and in-system test controllers. (See lines 37-138.)

As you did with the processor core, use the `read_config_data` command to edit the DFT specification.

- Specify the [OCC](#) wrapper and specify your clock intercept node for the OCC.

For details about OCC for the hybrid TK/LBIST flow, refer to “[Tessent OCC TK/LBIST Flow](#)” in the *Hybrid TK/LBIST Flow User’s Manual*.

- Specify the EDT wrapper to include a [LogicBistOptions](#) wrapper, and specify a MISR ratio of one.

The `edt_clock` and `edt_update` signals are automatically connected to EDT instances, and the EDT controller is built with bypass.

- Specify a [LogicBist](#) wrapper that contains both a Controller wrapper and an `NcpIndexDecoder` wrapper.

Because this core has a single clock, you only need one NCP; the tool does not generate the NCP index decoder.

- Specify an [InSystemTest](#) wrapper, ensuring that you specify that you are using the CPU interface.

Create a TCD segment for this instrument, and specify the in-system test controller connections using the Connections wrapper.

4. [Generating the EDT, Hybrid TK/LBIST, and OCC Hardware](#), plus the LogicBIST and in-system test hardware. (See line 139.)
5. [Extracting the ICL Module Description](#). (See line 141.)
6. [Generating ICL Patterns and Running Simulation](#). (See lines 143-151.)

Examples

The following dofile example shows a typical automotive command flow for a core-level first DFT insertion. Modify the DFT specifications for you design requirements.

Example 6-10. Dofile Example for DFT Insertion, `gps_baseband`

```

1 # Load the design
2 set_context dft -rtl -design_id rtl1
3 set_tsdb_output_directory ../tsdb_outdir
4 read_verilog ../rtl/gps_baseband.v
5
6 set_current_design gps_baseband
7
8 set_design_level physical_block
9
10 # Add required DFT signals for logic test
11 add_dft_signals ltest_en
12 add_dft_signals scan_en edt_update test_clock
13     -source_node {SCAN_EN_w edt_update test_clock_w }
14 add_dft_signals edt_clock shift_capture_clock -create_from_other_signals
15
16 # Add DFT signals required for hierarchical DFT (external, internal modes)
17 add_dft_signals int_ltest_en ext_ltest_en int_mode ext_mode
18
19 # Add DFT signal for controller chain mode
20 add_dft_signals controller_chain_mode
21
22 # Add required DFT signals specific to hybrid TK/LBIST flow
23 add_dft_signals observe_test_point_en control_test_point_en x_bounding_en
24
25 # Constrain the in-system test controller to 0 for manufacturing test
26 add_input_constraints mission_test_enable -C0
27
28 set_dft_specification_requirements -logic_test on
29
30 add_clocks clk -period 3ns
31
32 # Enable Observation Scan Technology (OST) for capturing per shift and
    capture
33 add_dft_signals capture_per_cycle_static_en
34
35 check_design_rules

```

```
36
37 # Create and report the DFT specification
38 set spec [create_dft_specification -sri_sib_list {edt occ lbist} ]
39 report_config_data $spec
40
41 # Edit DFT specification to specify OCC's SIB and to insert OCC
42 # Modify for your design requirements
43 read_config_data -in $spec -from_string {
44     OCC {
45         ijtag_host_interface : Sib(occ);
46         static_clock_control : external;
47     }
48 }
49
50 # Specify OCC insertion and intercept node
51 # This is a generic method for populating the OCC; modify for your design
52 # Scan enable and shift capture clock signals are automatically connected
53 # to the OCC instances
54 set id_clk_list [list \
55 clk clk\
56 ]
57 foreach {id clk} $id_clk_list {
58 set occ [add_config_element OCC/Controller($id) -in $spec]
59 set_config_value clock_intercept_node -in $occ $clk
60 }
61
62 # Specify the hybrid EDT configuration
63 read_config_data -in $spec -from_string {
64     EDT {
65         ijtag_host_interface : Sib(edt);
66         Controller (c1) {
67             longest_chain_range : 50, 65 ;
68             scan_chain_count : 60;
69             input_channel_count : 2;
70             output_channel_count : 2;
71             LogicBistOptions {
72                 misr_input_ratio : 1 ;
73                 ShiftPowerOptions {
74                     present : on ;
75                     default_operation : disabled ;
76                     SwitchingThresholdPercentage {
77                         min : 25 ;
78                     }
79                 }
80             }
81         }
82     }
83 }
84
85 # Specify the LogicBIST controller with NCP index decoder
86 read_config_data -in $spec -from_string {
87     LogicBist {
88         ijtag_host_interface : Sib(lbist);
89         Controller(1%ctrl_lbist) {
90             burn_in : on ;
91             pre_post_shift_dead_cycles : 8 ;
92             SingleChainForDiagnosis { Present : on ; }
93             ControllerChain {
```

```

94         present : on;
95         clock : tck;
96     }
97     Connections {
98         scan_en_in : SCAN_EN_w ;
99         controller_chain_scan_in : ccm_scan_in ;
100        controller_chain_scan_out : ccm_scan_out ;
101    }
102    Interface {
103        shift_clock_src : - ;
104    }
105    NcpOptions {
106        count : 1 ;
107    }
108    ShiftCycles { max : 200 ; }
109    CaptureCycles { max : 7 ; }
110    PatternCount { max : 1024 ; }
111    WarmupPatternCount { max : 512 ; }
112 }
113
114# Specify the in-system test controller
115read_config_data -in $spec -from_string {
116    InSystemTest {
117        Controller(c0) {
118            protocol : cpu_interface ;
119            host_interface : HostScanInterface(ijtag) ;
120            data_width : 8 ;
121            ControllerChain {
122                present : on ;
123                clock : tck ;
124                segment_per_instrument : on ;
125            }
126            Connections {
127                reset : hw_rstn; //reset of gps_baseband core
128                CpuInterface {
129                    clock : cpu_interface_clock ;
130                    enable : mission_test_enable ;
131                    write_en : from_cpu_write_en ;
132                    data_in : data_in;
133                    data_out : data_out;
134                }
135            }
136        }
137    }
138}
139process_dft_specification
140
141extract_icl
142
143# Write script for design compilation
144write_design_import_script for_dc_synthesis.tcl -replace
145
146set pat_spec [ create_patterns_specification ]
147process_patterns_specification
148
149run_testbench_simulations
150# If simulation fails use the command below to see which pattern failed
151check_testbench_simulations

```

152
153exit

LogicBIST Fault Simulation With One NCP: gps_baseband

The GPS baseband core contains one clock, and, therefore, one NCP. There is no need for NCP decoder logic so Tessent Shell does not generate the NCP index decoder during IP insertion. This means that you must explicitly specify the clock sequence during LogicBIST fault simulation.

The following dofile extract invokes the clock sequence during LogicBIST fault simulation. The sequence is generated by a TDR, and the static control clock is set to “both” during IP insertion.

Example 6-11. Dofile Example for LogicBIST Fault Simulation With One NCP

```
...

set_core_instance_parameters -instrument_type occ -parameter_values \
{static_clock_control internal clock_sequence 3}

add_bist_capture_range SINGLE_OCC_NCP 0 255

set_lbist_controller_options -programmable_ncp_list { SINGLE_OCC_NCP }

set_system_mode analysis

create_capture_procedures -name SINGLE_OCC_NCP \
-clock_sequence ([ get_name_list [ get_clock \
gps_baseband_rtl1_tessent_occ_clk_inst/ \
tessent_persistent_cell_clock_out_mux/y ]]) \
([ get_name_list [ get_clock \
gps_baseband_rtl1_tessent_occ_clk_inst/ \
tessent_persistent_cell_clock_out_mux/y ]])

...
```

Interconnect Bridge/Open UDFM Generation: gps_baseband

Generate an interconnect bridge and open UDFM on the gps_baseband module in the same way as you did for the processor_core module.

Refer to “[Interconnect Bridge/Open UDFM Generation: processor_core](#)” on page 294 to see how to generate an interconnect bridge/open UDFM on a wrapped core.

Refer to the following directory, which has an example on the `gps_baseband` module:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/gps_baseband/  
10.extract_fault_sites
```

Cell-Neighborhood UDFM Generation: `gps_baseband`

Generate a cell-neighborhood UDFM on the `gps_baseband` module in the same way as the `processor_core` module.

Refer to “[Cell-Neighborhood UDFM Generation: `processor_core`](#)” on page 296 to see how to generate a cell-neighborhood UDFM on a wrapped core.

Refer to the following directory, which has an example on the `gps_baseband` module:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/gps_baseband/  
10.extract_fault_sites
```

Automotive-Grade ATPG Pattern Generation: `gps_baseband`

Generate Automotive-Grade ATPG tophoff and UDFM patterns on the `gps_baseband` module in the same way as you did for the `processor_core` module.

Refer to “[Automotive-Grade ATPG Pattern Generation: `processor_core`](#)” on page 300 to see how to generate ATPG patterns on a wrapped core.

Refer to the following directory, which has an example on the `gps_baseband` module:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/gps_baseband/  
11.generate_AGA_patterns
```

Top-Level DFT Insertion for the Automotive Flow

You previously inserted CPU-based in-system test controllers inside the GPS baseband cores. At the top level, you now insert a DMA in-system test controller and associated memory block. In addition to a Verilog test bench, this controller requires memory or lookup table (LUT)-based logic to save both the controller instructions and the pattern data.

Typically, you use ROM for the DMA memory block, but you can use any storage mechanism as long as you configure it to behave like a clocked synchronous memory. You can perform MemoryBIST testing on this memory during manufacturing. For testing the memory in system, the process for applying power-on reset tests during initialization differs if you are using ROM, RAM, or LUT-based logic.

For general information about the RTL and scan DFT insertion flow for the top chip, refer to “[RTL and Scan DFT Insertion Flow for the Top Chip](#)” on page 165. (The documented process relates to a hierarchical test case; however, the basic flow remains the same.)


First DFT Insertion Pass: Top with MemoryBIST, BISR, and Boundary Scan	314
Second DFT Insertion Pass: Top with EDT, OCC, and In-System Test	320
Scan Insertion for the Top Design	325
ATPG Pattern Generation for the Top Design	327
ATPG Pattern Retargeting for the Top Design	328
Interconnect Bridge/Open UDFM Generation for the Top Design	328
Cell-Neighborhood UDFM Generation for the Top Design.	328
Automotive-Grade ATPG Pattern Generation for the Top Design	329
UDFM Generation for Cell-Aware ATPG	329
TCA Based Pattern Sorting	331

First DFT Insertion Pass: Top with MemoryBIST, BISR, and Boundary Scan


The top design includes I/O pads. Insert boundary scan plus MemoryBIST for any memories that are present at the top level; this includes the memory block for the DMA IST controller. Build the JTAG network for the wrapped cores at the top level. In addition, the design includes repairable memories in the processor core, so you must insert an efuse with an efuse interface and a BISR controller.

If you do not plan to implement hard repair, use the Tessent soft-repair only option.


Note

 You must perform the first DFT insertion pass, even if you do not use MemoryBIST or boundary scan to insert IJTAG to configure your child blocks. You cannot run logic test DRC in this first pass. Logic test DRC requires that the child blocks are connected to the network in order to be configured. Because of this, you can only run logic test DRC on the second DFT insertion pass, after the network is configured.

Note

 For general instructions about inserting MemoryBIST and boundary scan at the top level, refer to [“First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan”](#) on page 168. This test case adds in the BISR and DMA memory block to the baseline use case.

Note

 Because you cannot use the in-system test patterns from the DMA memory block to test that memory block, do not create in-system test patterns for testing the DMA memory.

If you plan to insert a LogicBIST controller at the same level as the DMA IST controller to generate in-system test logic test patterns (not illustrated by this test case), ensure that the DMA IST controller and its memory block have the same async clock source. In addition:

- Isolate the DMA memory block, its MemoryBIST logic, and the IST controller by pushing them into their own module.
- If isolation is not possible because of design considerations, exclude the observation logic inside the memory interface by disabling the `observation_xor_size` property. (The MemoryBIST controller and MemoryBIST interface are scannable logic.) This prevents a MISR signature difference between manufacturing and in-system LogicBIST tests.

For example:

```
set_config_value DftSpecification(processor_core,rtl1)/MemoryBist/  
Controller(c2)/Step/MemoryInterface(m1)/observation_xor_size off
```

In addition, turn off the memory library `DisableDuringScan` property, which removes both the gating logic at the memory inputs and the memory bypass mux that deactivates the memory control during scan and LogicBIST test. As needed, re-enable the memory bypass mux by setting the `scan_bypass_logic` property to `sync_mux`, as follows:

```
set_config_value DftSpecification(processor_core,rtl1)/MemoryBist/  
Controller(c2)/Step/MemoryInterface(m1)/scan_bypass_logic sync_mux
```

The memory bypass mux is supported in this scenario as long as you are directly connecting the memory data output to the IST controller data input.

Note

The line numbers used in this procedure refer to the command flow dofile in [Example 6-12](#) on page 317 unless otherwise noted.

Prerequisites

- To insert boundary scan, you must have an RTL design with instantiated I/O pads if you are using a chip-level design.
- For RTL netlists, you must have a Tessent cell library or the pad library.

Procedure

1. Load the design, including opening the TSDBs for the child cores: processor_core and gsp_baseband. (See lines 1-25.)

In addition, load the design for the DMA memory block, fusebox, and fusebox interface. The dofile example shows an example fusebox. You must include an efuse and efuse interface in your design unless you are opting for soft repair only.

2. Specify the DFT specification requirements. (See line 27.)

When you set the design level to chip, “-memory_test on” automatically initiates BISR insertion if BISR chains exist on blocks instantiated in the current design or repairable memories exist in the current design. This assumes that you have not changed the [set_dft_specification_requirements](#) -memory_bist, -memory_bisr_chains, and -memory_bisr_controller options from their default auto settings.

3. Identify TAP pins and pins that cannot add boundary scan cells. (See lines 29-41.)
4. Add DFT signals and clocks. (See lines 43-50.)
5. Create the connections for the CPU-based and DMA IST controllers. (See lines 52-68.)

You must connect the CPU-based IST controllers that you inserted at the block level so that they can be controlled by the system logic. In addition, create the connection between the DMA memory block and the DMA IST controller clock.

You can use a post-insertion script to connect the system logic with the IST controller.

6. Check the design rules and create the DFT specification. (See lines 70-76.)
7. Segment the boundary scan to be used during logic test. (See lines 78-79.)
8. Create the BISR controller connections for the clock, VDD, and fusebox. (See lines 81-91.)

To connect the BISR controller to the system logic, you must specify the connection for the BISR controller to use for repair clock, BISR reset, and VDD. Typically, system logic is connected to the BISR controller for initiating memory repair and monitoring the progress of the operation.

The BISR controller input clock must be driven by an appropriate functional clock. Specify the connection with the `repair_clock_connection` property in the `DftSpecification/MemoryBisr/Controller` wrapper. The BISR controller input signal resets the BISR chains and initiates memory repair. Specify the reset signal with the `repair_trigger_connection` property in the `DftSpecification/MemoryBisr/Controller` wrapper. Also, specify your fusebox module and its location.

9. Identify the functional pins to be shared with EDT channel pins and add the required auxiliary I/O logic. (See lines 93-100.)
10. Generate the hardware and extract ICL. (See lines 104-106.)
11. Generate ICL patterns and simulation test benches for the controllers and instruments in the current and lower physical instances. (See lines 108-115.)

To re-execute the lower physical instance simulations at higher levels, enable the `set_default_values simulate_instruments_in_lower_physical_instances` property.

12. Run and check test bench simulations. (See lines 117-130.)

Examples

The following dofile example shows a command dofile for the top-level first DFT insertion pass for the automotive flow.

Example 6-12. Dofile Example for Top-Level First DFT Insertion Pass, Automotive Flow

```

1 set_context dft -rtl -design_id rtl1
2 set_tsdb_output_directory ../tsdb_outdir
3 open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
4 open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir
5 read_cell_library ../../library/standard_cells/tessent/adk.tcelllib
6 read_verilog ../rtl/noncore_blocks/pll.v -blackbox
7
8 set_design_sources -format verilog \
9     -v ../rtl/noncore_blocks/pad8_io_macro.v \
10    -v ../rtl/noncore_blocks/iopad_sel.v \
11    ../rtl/noncore_blocks/iopad.v
12
13 # Read the memory block for the DMA IST controller too
14 read_verilog ../../library/memories/SYNC_1R1W_4096x8.v \
15     -interface_only -exclude_from_file_dictionary
16
17 read_verilog ../rtl/chip_top.v
18
19 # Read fusebox, example only
20 read_verilog fusebox/LVFuseBox.vb -exclude
21 read_verilog fusebox/genericFuseBox.vb
22 read_core_description fusebox/genericFuseBox.tcd_fbox
23
24 set_current_design chip_top
25 set_design_level chip
26

```

```
27 set_dft_specification_requirements -boundary_scan on -memory_test on
28
29 # Specify the TAP pins
30 set_attribute_value TCK -name function -value tck
31 set_attribute_value TDI -name function -value tdi
32 set_attribute_value TMS -name function -value tms
33 set_attribute_value TRST -name function -value trst
34 set_attribute_value TDO -name function -value tdo
35
36 # Specify pins that cannot add any boundary scan cells
37 set_boundary_scan_port_options TEST_CLOCK_top -cell_options dont_touch
38 set_boundary_scan_port_options EDT_UPDATE_top -cell_options dont_touch
39 set_boundary_scan_port_options SCAN_ENABLE -cell_options dont_touch
40 set_boundary_scan_port_options RESET_N -cell_options dont_touch
41 set_boundary_scan_port_options INCLK -cell_options dont_touch
42
43 # Add DFT signals
44 add_dft_signals scan_en -source_nodes SCAN_ENABLE
45
46 # Specify the clocks
47 add_clocks PLL_1/pll_clock_0 -reference REF_CLK -freq_multiplier 16
48 add_clocks REF_CLK -period 48
49 add_clocks TEST_CLOCK_top -period 10
50 add_clocks INCLK -period 10ns
51
52 # Create connections for CPU-based IST controller inserted in gps_baseband
53 proc process_dft_specification.post_insertion {topWrapper args} {
54 create_connection RDS_1/mission_test_enable_gps GPS_1/mission_test_enable
55 create_connection RDS_2/mission_test_enable_gps GPS_2/mission_test_enable
56 create_connection istb/write_en GPS_1/from_cpu_write_en
57 create_connection istb/write_en GPS_2/from_cpu_write_en
58 create_connection istb/clock GPS_1/cpu_interface_clock
59 create_connection istb/clock GPS_2/cpu_interface_clock
60 create_connection istb/data_out GPS_1/data_in
61 create_connection istb/data_out GPS_2/data_in
62 create_connection GPS_1/data_out istb/data_in1
63 create_connection GPS_2/data_out istb/data_in2
64
65 # Create connection for DMA IST controller
66 create_connection IST_memory/CLK
67 chip_top_rtl1_tessent_in_system_test_post_inst/dma_clock
68 }
69
70 check_design_rules
71 set_system_mode analysis
72
73 report_clocks
74
75 set_spec [create_dft_specification]
76 report_config_data $spec
77
78 # Segment the boundary scan to be used during logic test
79 set_config_value $spec/BoundaryScan/max_segment_length_for_logictest 80
80
81 # Create BISR controller connection for clock, VDD and fusebox
82 set_config_value -in $spec MemoryBisr/Controller/
83 repair_clock_connection PLL_1/pll_clock_0
84 set_config_value -in $spec
```

```


85 MemoryBisr/Controller/programming_voltage_source VDD/C
86 set_config_value -in $spec MemoryBisr/Controller/
87   repair_trigger_connection BISR_RESET/C
88 set_config_value -in $spec MemoryBisr/Controller/fuse_box_location
89   external
90 set_config_value -in $spec MemoryBisr/Controller/
91   fuse_box_interface_module genericFuseBox
92
93 # Add auxiliary MUX on the inputs and outputs used for EDT channel pins
94 read_config_data -in ${spec}/BoundaryScan -from_string {
95   AuxiliaryInputOutputPorts {
96     auxiliary_input_ports   : GPIO1_0, GPIO1_1, GPIO1_2, GPIO1_3;
97     auxiliary_output_ports  : GPIO2_0, GPIO2_1, GPIO2_2, GPIO2_3, GPIO1_0,
98     GPIO1_1 ;
99   }
100}
101
102report_config_data $spec
103
104process_dft_specification
105
106extract_icl
107
108# Include the lower-level physical instances for IJTAG retargeting
109set_defaults_value PatternsSpecification/SignOffOptions/
110   simulate_instruments_in_lower_physical_instances on
111
112# Create pattern test benches to verify the inserted DFT logic
113set_pat_spec [create_patterns_specification]
114
115process_pattern_specification
116
117set_simulation_library_sources -v ../../library/standard_cells/verilog/ \
118*.v
119-v ../rtl/noncore_blocks/p11.v \
120-v ../rtl/noncore_blocks/pad8_io_macro.v \
121-v ../rtl/noncore_blocks/iopad_sel.v \
122-v ../rtl/noncore_blocks/iopad.v \
123-v ../../library/memories/SYNC_1RW_8Kx16.v \
124-v ../../library/memories/SYNC_1RW_32x16_RC.v \
125-v ../../library/memories/SYNC_1R1W_4096x8.v \
126-v ../../library/standard_cells/verilog/adk.v \
127-v fusebox/LVFFuseBox.vb
128
129run_testbench_simulations -exclude {InSystemTest_0_MemoryBist_P1}
130check_testbench_simulations -report_status
131
132exit

```


Second DFT Insertion Pass: Top with EDT, OCC, and In-System Test

For the top level, the second DFT insertion pass for EDT and OCC include gray boxes, DFT signals for purposes of ATPG retargeting, and TAMs. In addition, for the automotive flow, you insert the DMA IST controller.

Note

 For top-level EDT and OCC, this procedure follows a typical second DFT insertion flow for a top design as detailed in “[Second DFT Insertion Pass: Inserting Top-Level EDT and OCC](#)” on page 171. Refer to this section for associated details.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 6-13](#) on page 321 unless otherwise noted.

Procedure

1. Load the design. (See lines 1-14.)

Ensure that you load in the TCD for the DMA memory block that is used for in-system test. (See line 10.)

2. Add DFT signals. (See lines 16-30.)

Ensure that you include a DFT signal for controller chain mode and define retargeting modes for each group of wrapped cores whose ATPG patterns you want to retarget.

3. Apply the [add_dft_modal_connections](#) command to specify the TAM and to connect the EDT channel IOs of the wrapped cores to top-level pins via the TAM. (See lines 32-76.)

Include connections for controller chain mode. For the processor core, use `retargeting_mode1`, and for the GPS baseband, use `retargeting_mode2`.

With Tessent Shell you can share functional pins as EDT channel pins. The dofile shows that the input pin `GPI01_0` is shared as an EDT channel input pin, and the output pin `GPI02_0` is shared as an EDT channel output pin. Different modes can also share input and output pins. For example, `GPI01_2` functions as both the processor core EDT mode channel input and the controller chain mode uncompressed input for the entire design.

4. Set the DFT specification requirements. (See line 78.)

5. Create and process the DFT specification, using the `read_config_data` command to add the EDT controller with bypass and the DMA IST controller. (See lines 83-150.)

The top-level EDT controller includes bypass, and the `edt_clock` and `edt_update` signals are automatically connected to EDT instances. The `connect_bscan_segments_to_lsb_chains` property defaults to `auto` and connects the

divided boundary scan segments from the previous insertion pass to the top-level EDT controller.

Using the `set_config_value` command, connect the top-level EDT channels that you left unconnected during step 3 to the GPIO ports. (See lines 108-114.) The controller chain connections are completed during scan insertion.

Finally, insert the DMA IST controller. (See lines 116-150.) Set the protocol property to `direct_memory_access`. To account for ATPG coverage of the IST controller under controller chain mode, you can specify a TCD scan segment. Specify the data width, address width, and `max_test_program_count` that determine the bus width of the IST controller's program index, and thus the number of patterns that are programmed for the controller. In addition, specify the connections for the DirectMemoryAccess interface. The DMA IST controller inserts a comparator circuit and outputs a fail flag and done bit.

6. Generate ICL patterns and simulation test benches for the instruments in the current and lower physical instances. (See lines 152-172.)

Create the in-system test patterns for the JTAG instruments you want to test. The dofile example illustrates applying the `MemoryBist_P1` pattern to test the memory inside the processor core. The resulting Verilog test bench file is named *FilePrefix_TestProgramIndex_PatternSetName.v*, and the memory file (for use with the DMA IST controller) is named *testbench_file_name.mem*.

7. Run and check test bench simulations. (See line 174-194.)

For in-system test, ensure that you point to the memory file that contains the control instructions for the test bench and the data necessary for interaction with the DMA IST controller. (See line 191.) This file has the same name as the test bench file with the addition of a ".mem" suffix. The test bench loads the memory file during simulation.

In addition, add three levels to the IST controller's actual location because execution occurs inside three levels in the `simulation_outdir` directory.

After running the in-system test patterns, run the remaining test bench simulations for the top and the physical blocks.

Examples

The following dofile example shows a command dofile for the top-level second DFT insertion pass for the automotive flow.

Example 6-13. Dofile Example for Top-Level Second DFT Insertion Pass, Automotive Flow

```

1 set_context dft -rtl -design_id rtl2
2 set_tsdb_output_directory ../tsdb_outdir
3 open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
4 open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir
5
6 read_design chip_top -design_id rtl1 -verbose
```

```
7 read_design processor_core -design_id gate1 -view graybox -verbose
8 read_design gps_baseband -design_id gate1 -view graybox -verbose
9
10 read_core_description ../../library/memories/SYNC_1R1W_4096x8.lvlib
11 read_verilog ../../library/memories/SYNC_1R1W_4096x8.v \
12     -interface_only exclude_from_file_dictionary
13
14 set_current_design chip_top
15
16 # Add DFT signals
17 add_dft_signals int_ltest_en output_pad_disable
18 add_dft_signals tck_occ_en
19 add_dft_signals ltest_en
20 add_dft_signals edt_mode
21 add_dft_signals controller_chain_mode
22
23 # These are used for top-level EDT
24 add_dft_signals test_clock edt_update \
25     -source_nodes {TEST_CLOCK_top EDT_UPDATE_top}
26 add_dft_signals shift_capture_clock edt_clock -create_from_other_signals
27
28 # Define retargeting modes
29 add_dft_signals retargeting1_mode retargeting2_mode retargeting3_mode \
30     retargeting4_mode
31
32 # TAM connections
33 add_dft_modal_connections -ports GPIO1_0 \
34     -input_data_destination_nodes * -enable_dft_signal edt_mode
35 add_dft_modal_connections -ports GPIO2_0 \
36     -output_data_source_nodes * -enable_dft_signal edt_mode
37 add_dft_modal_connections -ports GPIO1_0 -input_data_destination_nodes \
38     * -enable_dft_signal controller_chain_mode
39 add_dft_modal_connections -ports GPIO2_0 -output_data_source_nodes \
40     * -enable_dft_signal controller_chain_mode
41
42 # For processor_core
43 add_dft_modal_connections -ports GPIO1_2 -input_data_destination_nodes \
44     PROCESSOR_1/processor_core_rtl2_controller_c1_edt_channels_in[0] \
45     -enable_dft_signal retargeting1_mode
46 add_dft_modal_connections -ports GPIO2_2 -output_data_source_nodes \
47     PROCESSOR_1/processor_core_rtl2_controller_c1_edt_channels_out[0] \
48     -enable_dft_signal retargeting1_mode
49
50 # For gps_baseband
51 add_dft_modal_connections -ports GPIO1_2 -input_data_destination_nodes \
52     GPS_1/gps_baseband_rtl1_controller_c1_edt_channels_in[0] \
53     -enable_dft_signal retargeting2_mode
54 add_dft_modal_connections -ports GPIO1_2 -input_data_destination_nodes \
55     GPS_2/gps_baseband_rtl1_controller_c1_edt_channels_in[0] \
56     -enable_dft_signal retargeting2_mode
57 add_dft_modal_connections -ports GPIO1_3 -input_data_destination_nodes \
58     GPS_1/gps_baseband_rtl1_controller_c1_edt_channels_in[1] \
59     -enable_dft_signal retargeting2_mode
60 add_dft_modal_connections -ports GPIO1_3 -input_data_destination_nodes \
61     GPS_2/gps_baseband_rtl1_controller_c1_edt_channels_in[1] \
62     -enable_dft_signal retargeting2_mode
63 add_dft_modal_connections -ports GPIO1_0 -output_data_source_nodes \
64     GPS_1/gps_baseband_rtl1_controller_c1_edt_channels_out[0] \
```

```

65     -enable_dft_signal  retargeting2_mode -pipeline_stages 1
66 add_dft_modal_connections -ports GPIO1_1 -output_data_source_nodes \
67     GPS_1/gps_baseband_rtl1_controller_c1_edt_channels_out[1] \
68     -enable_dft_signal  retargeting2_mode -pipeline_stages 1
69 add_dft_modal_connections -ports GPIO2_0 -output_data_source_nodes \
70     GPS_2/gps_baseband_rtl1_controller_c1_edt_channels_out[0] \
71     -enable_dft_signal  retargeting2_mode -pipeline_stages 1
72 add_dft_modal_connections -ports GPIO2_1 -output_data_source_nodes \
73     GPS_2/gps_baseband_rtl1_controller_c1_edt_channels_out[1]
74     -enable_dft_signal  retargeting2_mode -pipeline_stages 1
75
76 report_dft_modal_connections
77
78 set_dft_specification_requirements -logic_test on -memory_test on
79
80 set_system_mode analysis
81 report_dft_control_points
82
83 # Create DFT specification
84 set_spec [create_dft_specification -sri_sib_list {occ edt lbist} ]
85 report_config_data $spec
86 read_config_data -in $spec -from_string {
87     OCC {
88         ijtag_host_interface : Sib(occ);
89     }
90 }
91 set id_clk_list [list \
92     pll_clock_0 PLL_1/pll_clock_0 \
93     INCLK INCLK \
94 ]
95 foreach {id clk} $id_clk_list {
96 set occ [add_config_element OCC/Controller($id) -in $spec]
97 set_config_value clock_intercept_node -in $occ $clk
98 }
99 report_config_data $spec
100
101 read_config_data -in $spec -from_string {
102     EDT {
103         ijtag_host_interface : Sib(edt);
104         Controller (c1) {
105             ...
106 }
107 }
108 # Connect the EDT channels to the GPIO ports
109 set_config_value port_pin_name \
110     -in $spec/EDT/Controller(c1)/Connections/EdtChannelsIn(1) \
111     [get_auxiliary_pins GPIO1_0 -direction input]]
112 set_config_value port_pin_name \
113     -in $spec/EDT/Controller(c1)/Connections/EdtChannelsOut(1) \
114     [get_single_name [get_auxiliary_pins GPIO2_0 -direction output] ]
115
116 # Insert DMA IST controller
117 read_config_data -in $spec -from_string {
118     InSystemTest {
119         Controller(post) {
120             DesignInstance (.) {}
121             // host_interface: HostScanInterface(tap);
122             data_width : 8 ;

```

```

123     protocol : direct_memory_access ;
124     ControllerChain {
125         present : on ;
126         clock: tck;
127         segment_per_instrument: on;
128     }
129     DirectMemoryAccessOptions {
130         max_wait_cycles: 6400;
131         address_width: 12;
132         max_test_program_count: 2;
133     }
134     Connections {
135         reset: RESET_N;
136         DirectMemoryAccess {
137             clock: PLL_1/pll_clock_0_100;    // free-running clock
138             enable: istb/enable;
139             test_program_index: istb/program_index;
140             mem_data: IST_memory/Q[%d];
141             mem_address: DMA_A/Y[%d];
142             fail_flag: istb/fail_flag ;
143             test_program_done: istb/test_done;
144         }
145     }
146 }
147 }
148}
149
150process_dft_specification
151
152extract_icl
153
154write_design_import_script  for_dc_synthesis.tcl -replace
155
156# Include the lower-level physical instances for IJTAG retargeting
157set_defaults_value PatternsSpecification/SignOffOptions/
158     simulate_instruments_in_lower_physical_instances on
159
160set pat_spec [create_patterns_specification]
161
162# Create in-system test pattern for core-level memory
163add_core_instance -module [ get_name [get_icl_module *post] ]
164read_config_data -replace -in $pat_spec -from_string {
165     InSystemTest {
166         Controller(chip_top_rtl1_tessent_in_system_test_post_inst) {
167             TestProgram(0) { pattern : MemoryBist_P1 ; }
168         }
169     }
170}
171
172process_pattern_specification
173
174set_simulation_library_sources
175     -v ../../library/standard_cells/verilog/*.v
176-v ../rtl/noncore_blocks/pll.v \
177-v ../rtl/noncore_blocks/pad8_io_macro.v \
178-v ../rtl/noncore_blocks/iopad_sel.v \
179-v ../rtl/noncore_blocks/iopad.v \
180-v ../../library/memories/SYNC_1RW_8Kx16.v \

```


```
181-v ../../library/memories/SYNC_1RW_32x16_RC.v \  
182-v ../../library/memories/SYNC_1R1W_4096x8.v \  
183-v ../../library/standard_cells/verilog/adk.v \  
184-v fusebox/LVFuseBox.vb  
185  
186# Add three levels to the actual location because execution occurs inside  
187# three levels in the simulation_outdir/  
188run_testbench_simulations -simulator_option  
189     { +nowarnTSCALE -voptargs="+acc"  
190       +IST_INIT_MEM=../../../../../tsdb_outdir/patterns/  
191       chip_top_rtl2.patterns_signoff/InSystemTest.mem }  
192     -select InSystemTest_0_MemoryBist_P1  
193  
194run_testbench_simulations -exclude {InSystemTest_0_MemoryBist_P1 }  
195  
196exit
```

Scan Insertion for the Top Design


After synthesizing your design, stitch the scan logic at the top-level along with the wrapper chains (external mode) of the cores. During scan insertion, the non-scan instances such as EDT are automatically understood. In addition, the built-in OCC sub-chains are understood and stitched into scan chains. The TCD segments that were created for the hybrid IP and IST controller test logic must be stitched together within the controller chain mode so that you can test them during ATPG.

Tessent uses the DFT signals that you specified during the previous insertion passes, therefore you do not need to specify them again.

Note

 Tessent Shell works with any third-party scan insertion or other tools. However, because all Tessent products reside on the same code and database, you lose some automation with third-party tools.

Note

 The line numbers used in this procedure refer to the command flow dofile in [Example 6-14](#) on page 326 unless otherwise noted.

Procedure

1. Load the design. (See lines 1-12.)

Ensure that you specify a unique design ID, and that you open the TSDB for all the child cores.

2. Exclude some design objects from the scan insertion process. (See lines 17-19.)

Exclude the pipeline stages that you added using `add_dft_modal_connections` and the fusebox instance.

3. Constrain the BISR reset to constant 1 (c1). (See line 21.)
4. Create EDT and controller chain mode scan modes. (See lines 25-55.)

Create an EDT scan mode to connect the scan chains to the EDT signals and EDT hardware that you inserted during the second DFT insertion pass.

Create a top-level controller chain mode. To do this, you must first set the `active_child_scan_mode` attribute value, which enables you to select the active child scan mode of a multi-mode core and build a single chain for all controller chain mode IPs in all cores and also the top-level IST controller. The IST controller must be tested in controller chain mode.

When you specify `add_chain_mode` for the controller chain mode, include controller chain mode segments from the processor core, GPS baseband cores and the top elements. To reuse the I/O pins you created with `add_dft_modal_connections` during the second DFT insertion pass, specify the auxiliary pin SI/SO connections. For controller chain mode, you must specify the SI/SO-associated ports so the tool recognizes that the chains are uncompressed.

5. Perform scan insertion. (See lines 57-61.)

Examples

The following dofile example shows a command dofile for top-level scan insertion for the automotive flow.

Figure 6-14. Dofile Example for Top-Level Scan Chain Insertion, Automotive Flow


```
1 # Load the design
2 set_context dft -scan -design_id gate3
3 open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
4 open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir
5 read_cell_library ../../library/standard_cells/tessent/adk.tcelllib
6 read_verilog ../3.synthesize_rtl/chip_top_synthesized.vg
7
8 read_design chip_top -design_id rt12 -no_hdl -verbose
9 read_design processor_core -design_id gate1 -verbose
10 read_design gps_baseband -design_id gate1 -view graybox -verbose
11
12 set_current_design chip_top
13
14 report_clocks
15 report_static_dft_signal_settings
16
17 # Specify the non-scan instances
18 add_nonscan_instances *tessent_dft_modal_*
19 add_nonscan_instances -instances fbi_instance
20
21 add_input_constraints BISR_RSTN_PAD -c1
```

```
22
23 set_system_mode analysis
24
25 # Create a scan mode to connect scan chains to EDT
26 set_edt_instance [get_name_list [get_instance -of_module [get_name
27     [get_icl_module -of_instances chip_top* -filter
28     tessent_instrument_type==mentor::edt]]] ]
29
30 add_scan_mode edt_mode -edt_instance $edt_instance
31
32 # Start creating a scan mode for controller_chain_mode
33 set_attribute_value [get_instance -of_module *in_system_test*] -name
34     active_child_scan_mode -value controller_chain_mode
35 set_attribute_value [get_instance PROCESSOR_1] -name
36     active_child_scan_mode -value controller_chain_mode
37 set_attribute_value [get_instance GPS_1] -name active_child_scan_mode
38     -value controller_chain_mode
39 set_attribute_value [get_instance GPS_2] -name active_child_scan_mode
40     -value controller_chain_mode
41
42 # Create a scan chain family for elements at the top
43 create_scan_chain_family ccm_mm_top -include_elements
44     [get_scan_elements -of_child_scan_mode controller_chain_mode]
45
46 # Create the ccm scan mode and include elements at core and top
47 add_scan_mode controller_chain_mode -include_chain_families
48     {ccm_mm_top}
49     -include_elements{controller_chain_mode/PROCESSOR_1/ccm_scan_out
50     controller_chain_mode/GPS_1/ccm_scan_out
51     controller_chain_mode/GPS_2/ccm_scan_out} -si_connections
52     [get_single_name [get_auxiliary_pins GPIO1_2 -direction input]]
53     -so_connection [get_single_name [get_auxiliary_pins GPIO2_2
54     -direction output] ] si_associated_ports GPIO1_2
55     -so_associated_ports GPIO2_2 -enable_dft_signal controller_chain_mode
56
57 analyze_scan_chains
58 report_scan_chains
59
60 insert_test_logic
61 report_scan_cells > scan_cells.list
62
63 exit
```

ATPG Pattern Generation for the Top Design

For top-level ATPG pattern generation, use the scan-inserted design data for the chip. Tessent uses the graybox models for the wrapped cores so that you can use the external mode of the wrapper chains to run ATPG.

Note

 Top-level ATPG pattern generation for the automotive flow follows the typical hierarchical flow as described in “[Top-Level ATPG Pattern Generation Example](#)” on page 175.

If you used Tessent Scan to insert the scan chains, specify the `import_scan_mode` command for ATPG pattern generation. Tessent Shell passes the scan-inserted design data through for the EDT and OCC logic. This data includes the scan structures that are stored in the TSDB under the “gate” design ID. You can create ATPG patterns for any mode that you need, such as stuck-at, transition, and controller chain mode.

In addition, Tessent automatically creates and simulates the `test_setup` procedure cycles that are required to initialize the EDT and OCC static signals. You only need to specify non-default parameter values, if, for example, you run EDT with `bypass on` or set `int_ltest_en` to 1 to use the boundary scan as the source of the core values and isolate the ATPG test from the top-level IOs.

You can read in the stuck-at fault models for the wrapped cores to calculate the total fault coverage for the chip.

ATPG Pattern Retargeting for the Top Design

For each wrapped core, retarget the internal ATPG patterns for all the modes you specified and all the fault types.

Perform pattern retargeting as described in “[Performing Top-Level ATPG Pattern Retargeting](#)” on page 177.

Interconnect Bridge/Open UDFM Generation for the Top Design

Generate an interconnect bridge and open UDFM for the top design module in the same way as you did for the `processor_core` module.

Refer to “[Interconnect Bridge/Open UDFM Generation: processor_core](#)” on page 294 to see how to generate an interconnect bridge/open UDFM on a wrapped core.

Refer to the following directory, which has an example on the top design module:

```
tessent_automotive_reference_flow_<software_version>/top/9.extract_fault_sites
```

Cell-Neighborhood UDFM Generation for the Top Design

Generate a cell-neighborhood UDFM for the top design module in the same way as the `processor_core` module.

Refer to “[Cell-Neighborhood UDFM Generation: processor_core](#)” on page 296 to see how to generate a cell-neighborhood UDFM on the top design.

Refer to the following directory, which has an example on the top design module:

tessent_automotive_reference_flow_<software_version>/top/9.extract_fault_sites

Automotive-Grade ATPG Pattern Generation for the Top Design

Generating automotive-grade ATPG patterns on the top design is similar to regular ATPG pattern generation. It needs several UDFM files for cell-internal defects, interconnect bridge or open defects, and cell-neighborhood defects.

Refer to the [Interconnect Bridge/Open UDFM Generation: processor_core](#) for interconnect bridge/open UDFM generation on your design, and [Cell-Neighborhood UDFM Generation: processor_core](#) for cell-neighborhood UDFM generation for your design.

Refer to “[UDFM Generation for Cell-Aware ATPG](#)” on page 329 for details on how to generate the cell-aware UDFM for your technology library.

As you do for regular ATPG, you must use graybox models for wrapped cores. The only difference with the regular top-level ATPG, in terms of the flow, is that you must read UDFM files for Automotive-Grade ATPG on the top design. You can run ATPG with topoff runs by loading only the UDFM files you want to target during an ATPG run. Also, you can run ATPG all together by reading in all UDFM files at once.

Refer to “[Automotive-Grade ATPG Pattern Generation: processor_core](#)” on page 300 to see how to generate ATPG patterns on a wrapped core.

Refer to the following directory, which has an example on the top design module:


tessent_automotive_reference_flow_<software_version>/top/10.generate_AGA_patterns

UDFM Generation for Cell-Aware ATPG

To run cell-aware ATPG, you should generate cell-aware UDFMs for standard cells using Tessent CellModelGen. Ensure that all input requirements and required tools are available before running Tessent CellModelGen.

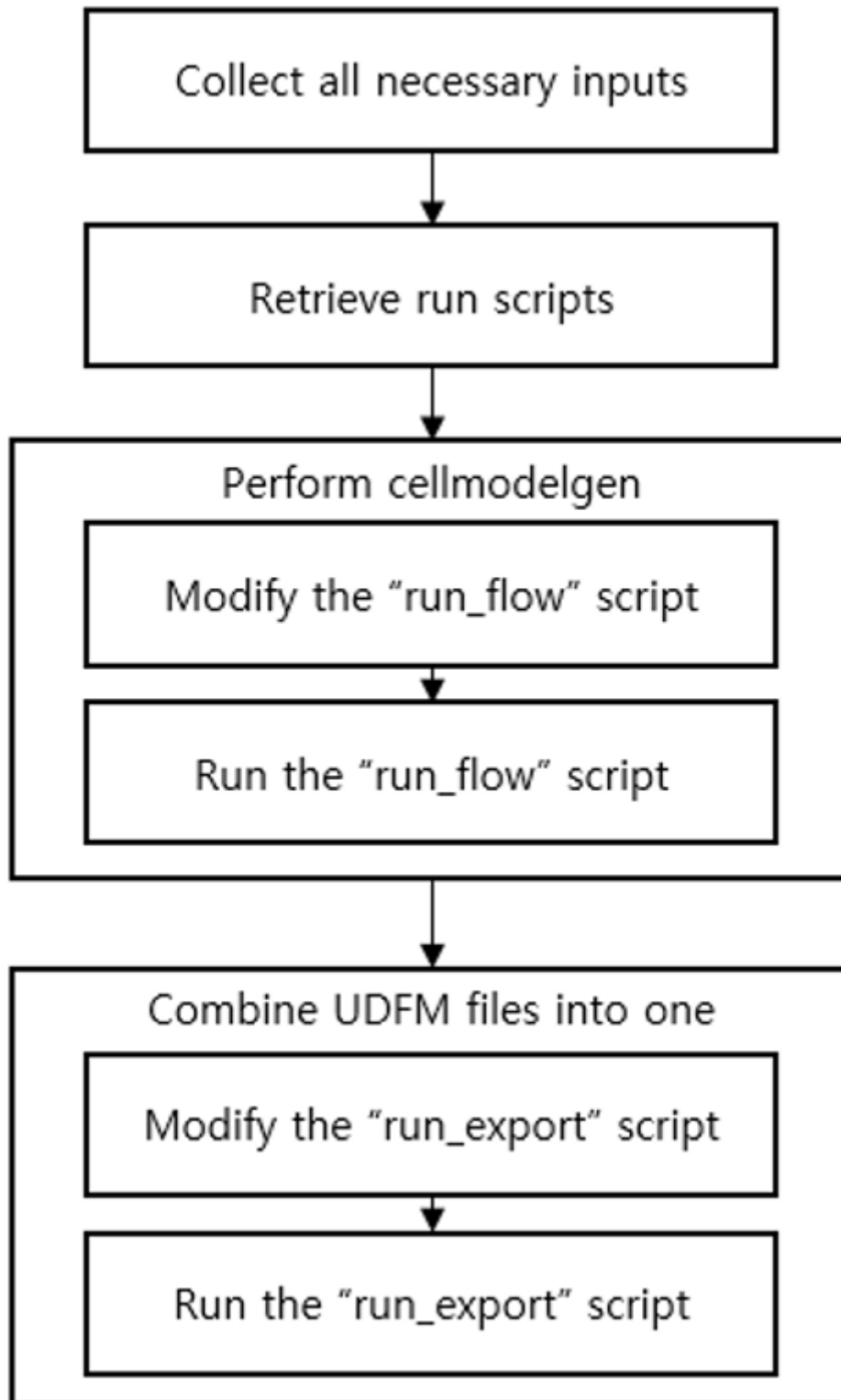
For information on Tessent CellModelGen or input requirements, refer to the *Tessent CellModelGen User’s Manual*.

Note

 Refer to the following directory, which has a usage example described in this section:

tessent_automotive_reference_flow_<software_version>/udfm_gen/stdlib

Figure 6-15. Cell-Aware UDFM Generation Flow



Procedure

1. Check all inputs and tools requirements are met.

For details, refer to the “[Required Licenses and Input Data](#)” chapter in the *Tessent CellModelGen Tool Reference*.

2. Create an empty directory, and retrieve run scripts:


```
cellmodelgen -get_script all
```

3. Look at the existing *run_flow* script in the directory. Modify the *run_flow* script for your design. Refer to the inline comments within the script for details.
4. Run the *run_flow* script to run Tessent CellModelGen on your standard cells.
5. Look at the existing *run_export* script in the directory. Modify the *run_export* script for your design. Refer to the inline comments within the script for details.
6. Run the *run_export* script to combine the individual standard cell UDFM files into a single UDFM file.

TCA Based Pattern Sorting

Tessent Shell provides a way to sort ATPG patterns based on the total critical area (TCA) of defects. The TCA defect data is saved in cell-aware, interconnect bridge/open, and cell-neighborhood UDFM files.

Note

 The line numbers used in this procedure refer to the command flow in “[Dofile Example for Running ATPG With All Types of UDFM in Internal Mode for gps_baseband](#)” on page 332. Refer to the following directory, which has a usage example described in this section:

```
tessent_automotive_reference_flow_<software_version>/wrapped_cores/gps_baseband/  
12.pattern_sorting
```

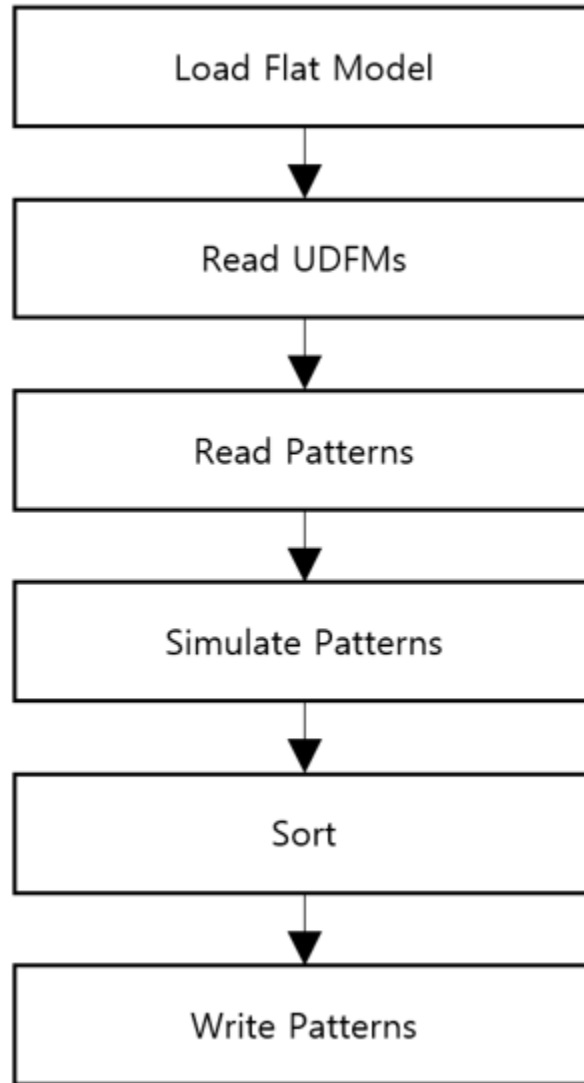
Procedure

1. Load the flat model generated from a post-layout design. (See lines 1–11.)
2. Set a fault type, read in the UDFM file for your standard cells, and add faults. (See lines 13-18.)
3. Read the patterns you want to sort. (See lines 20-26.)
4. Run the “[set_critical_area_options](#) -reporting on” command to enable the TCA coverage reporting feature. (See line 28.)
5. Simulate the patterns. (See line 30.)
6. Run the “[order_patterns](#) -critical_area” to sort the patterns based on TCA. (See line 31.)

7. Write out the sorted patterns (See line 33.)

Examples

Figure 6-16. TCA Based Pattern Sorting Flow



The following dofile shows a command flow for generating a bridge/open UDFM for use in the Automotive-Grade ATPG flow.

Example 6-14. Dofile Example for Running ATPG With All Types of UDFM in Internal Mode for gps_baseband

```
1 set_context patterns -scan -design_id pnr
2 set design_name "gps_baseband"
3
4 # Specify where the tsdb_outdir is to be located, default is the current
  working directory
5 set_tsdb_output_directory ../tsdb_outdir
```

```
6
7 # Add more processors to run ATPG
8 add_processors localhost:4
9
10 # Read flat model
11 read_flat_model ../tsdb_outdir/logic_test_cores/
   ${design_name}_pnr.logic_test_core/${design_name}.atpg_mode_ATPG_int/
   ${design_name}_ATPG_int.flat.gz
12
13 # Read UDFMs
14 set_fault_type udfm -static_fault
15 read_fault_sites ../../../../udfm_gen/stdlib/udfm/
   NangateOpenCellLibrary.udfm
16 read_fault_sites ../10.extract_fault_sites/udfm/
   ${design_name}_brdg_opn.udfm
17 read_fault_sites ../10.extract_fault_sites/udfm/CELLS_neighbor.udfm
18 add_faults -all
19
20 # Read patterns
21 #read_patterns ../tsdb_outdir/logic_test_cores/
   ${design_name}_pnr.logic_test_core/${design_name}.atpg_mode_ATPG_int/
   ${design_name}_ATPG_int_stuck.patdb
22 read_patterns ../11.generate_AGA_patterns/patterns/
   ${design_name}_int_stuck.stil.gz
23 read_patterns ../11.generate_AGA_patterns/patterns/
   ${design_name}_int_CAT_static_topoff.stil.gz -append
24 read_patterns ../11.generate_AGA_patterns/patterns/
   ${design_name}_int_BRDG_static_topoff.stil.gz -append
25 read_patterns ../11.generate_AGA_patterns/patterns/
   ${design_name}_int_OPN_static_topoff.stil.gz -append
26 read_patterns ../11.generate_AGA_patterns/patterns/
   ${design_name}_int_intercell_static_topoff.stil.gz -append
27
28 set_critical_area_options -reporting on
29
30 simulate_patterns -source external -store_patterns all
31 order_patterns -critical_area
32
33 write_patterns patterns/${design_name}_int_static_topoff_sorting.stil.gz
   -stil -rep
34 exit
```

Functional Mode Fault Tolerance for Static JTAG Signals

Tessent Shell provides a way to implement gating logic to ensure that a single-event upset (SEU) does not affect the mission mode operation of a user design.

TDR outputs are ORed in a hardware fault tolerant (HFT) module. The result is available as gated DFT signals and an alarm signal that you monitor during functional operation mode. The alarm indicates that a test signal register value has been altered because of an SEU.

This gating and monitoring logic is available for static DFT signals implemented with the `-create_with_tdr` option. You can control the implementation of this logic for individual signals.

Procedure

1. Add the `dft_signal_disable` DFT signal to gate the TDR logic:

```
add_dft_signals dft_signal_disable
```

2. Enable the creation of the HFT module for all static DFT signals interacting with functional logic:

```
set_dft_signals_options -disable_for_functional_safety static
```

3. Add the individual DFT signals that you want to monitor:

```
add_dft_signals dft_signal -create_with_tdr \  
-disable_for_functional_safety on
```

4. Add any DFT signals you want to exclude from monitoring:

```
add_dft_signals dft_signal -create_with_tdr \  
-disable_for_functional_safety off
```

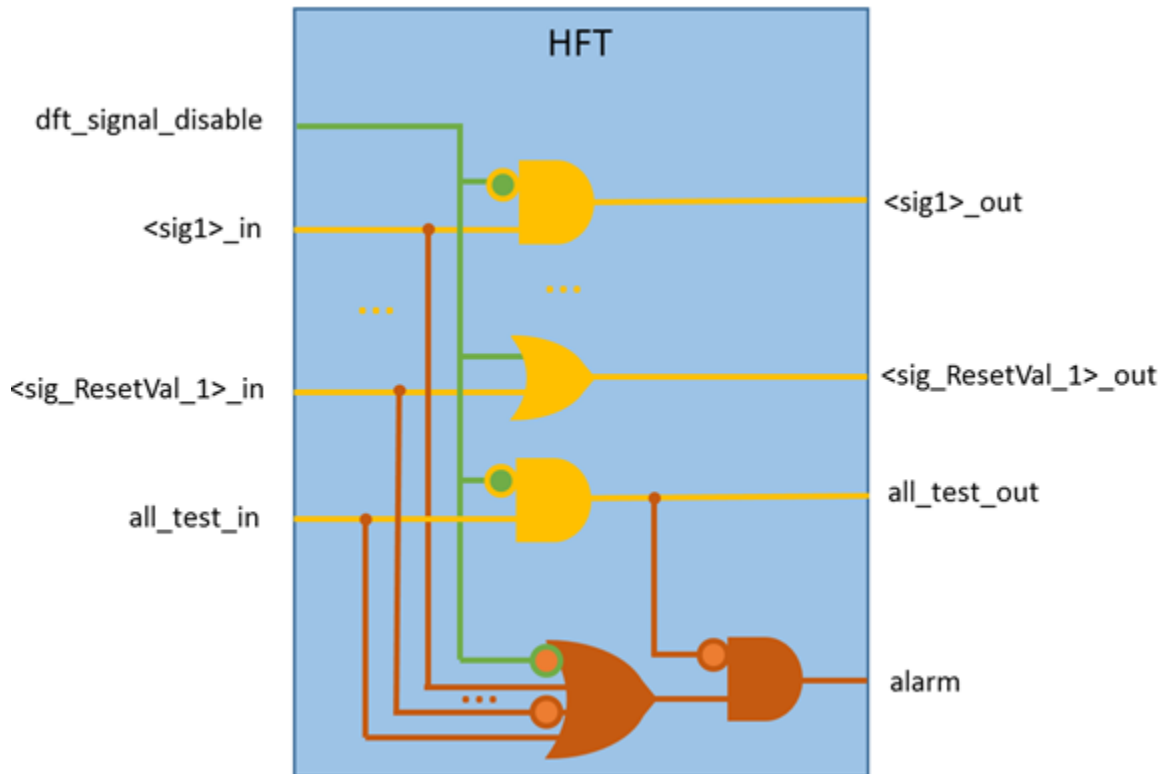
5. Create and process the `DftSpecification`:

```
check_design_rules  
create_dft_specification  
process_dft_specification
```

Results

An instance of an HFT module is created in a TDR hosting the static DFT signals. This module provides the gating of the specified static DFT signals in addition to SEU monitoring logic. See [Figure 6-17](#) for an example.

Figure 6-17. Hardware Fault Tolerant Module Example



The alarm signal is raised only in functional mode, and is gated with the all_test_out signal (the all_test signal gated with fault tolerant logic). It is accessible on a persistent buffer regardless of the DftSpecification add_persistent_buffers_in_scan_resource_instruments setting. The buffer instance name is `<tessent_persistent_cell_prefix>_hft_alarm_buf`.

Use the following command to introspect the alarm signal:

```
get_icl_pins -filter \
    {tessent_dft_function=="functional_mode_alarm"} \
    -of_instance [get_icl_instance -filter \
    {tessent_instrument_type=="mentor::ijtag_node"}]
```

Note

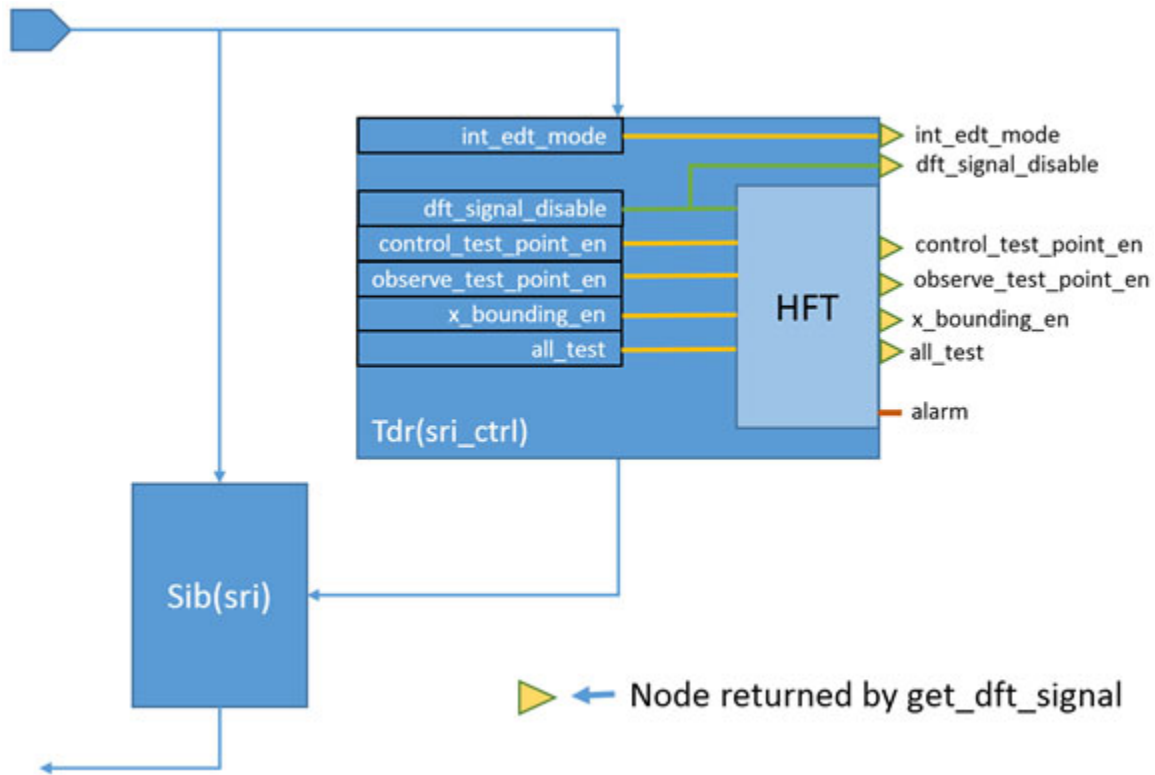
If you do not add the DFT signals explicitly, the tool creates all_test and dft_signal_disable automatically.

Note

Typically, the reset value for DFT signals is 0. When the reset value is 1, the gating logic is inverted, as shown with `<sig_ResetVal_1>` in Figure 6-17. Each DFT signal holds its reset value in the functional mode of circuit operation.

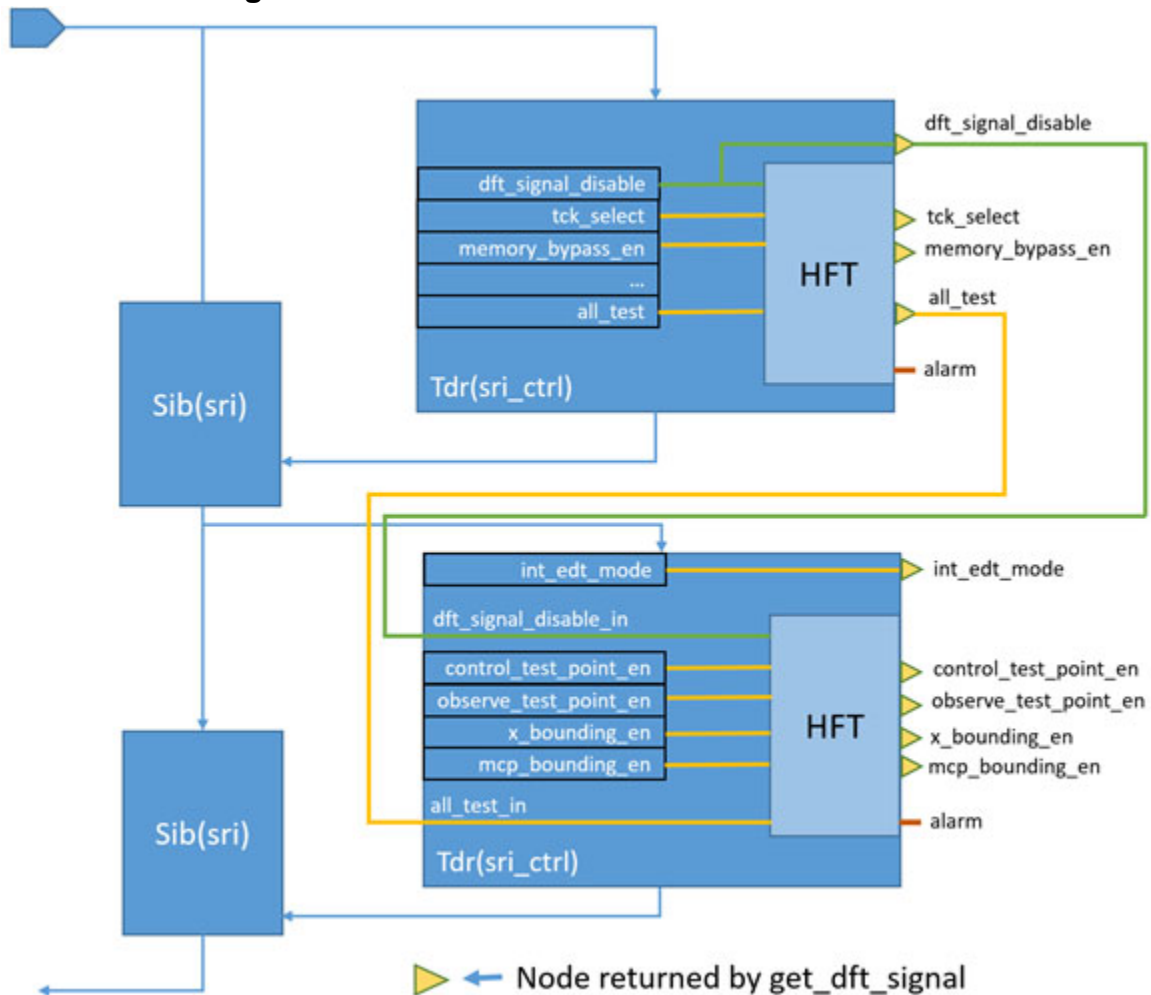
In the single-pass DFT insertion flow, the dft_signal_disable signal is created in a Scan Resource Instrument (SRI) TDR. See Figure 6-18 for an example.

Figure 6-18. HFT Module in the Single-Pass Flow



In the two-pass DFT insertion flow, the `dft_signal_disable` and `all_test` signals are defined in a `Tdr(sri_ctrl)` module created during the first insertion pass. The TDR with logic test signals is created in the second DFT insertion pass, and takes the `all_test` and `dft_signal_disable` signals as primary inputs. Each TDR instance contains its own HFT logic, and two alarm signals are available on the TDR output ports. See [Figure 6-19](#) for an example.

Figure 6-19. HFT Modules in the Two-Pass Flow



If you implement signal monitoring with the global `set_dft_signals_options` command, only those signals that affect the functional mode of operation can be monitored. If you implement signal monitoring and gating on a per-signal basis, any signal can be monitored.

By default, the `capture_per_cycle_static_en` and `se_pipeline_en` logic test control signals are not monitored. Also, no scan mode signals are monitored by default. Monitoring and gating is possible only for those static DFT signals you create with the `-create_with_tdr` option.

If you create the `dft_signal_disable` signal using the `-source_node` option, the tool implements the source node as part of an ICL network regardless of design level.

Related Topics

[add_dft_signals](#)

[set_dft_signals_options](#)

[get_dft_signals_option](#)

Chapter 7

TSDB Data Flow for the Tessent Shell Flow

The Tessent Shell Database (TSDB) is a repository for all the files and directories that Tessent Shell generates. The TSDB enhances flow automation by acting as the central location where Tessent can access the data it requires for the current task, whether that task be reading in a design, performing DRC, inserting logic test hardware, or performing ATPG pattern generation.

The TSDB structure aids data management between steps in a process even if you are not performing these steps within the Tessent Shell platform. If the steps are performed within Tessent Shell, then specifying the correct design ID automatically ensures that Tessent uses the correct file inputs for the current task.

Refer to “[Tessent Shell Database \(TSDB\)](#)” in the *Tessent Shell Reference Manual* for details about the TSDB directory structure and contents.

The data flow content builds on the material described in “[Tessent Shell Flow for Flat Designs](#)” and “[Tessent Shell Flow for Hierarchical Designs](#)” regarding the RTL and scan DFT insertion flow. During the RTL and scan DFT insertion process, Tessent Shell generates many output files and directories that it accesses later in the flow as data inputs. This chapter illustrates the data flow through each step of the RTL and scan DFT insertion flow.

Core-Level or Flat TSDB Data Flow	339
Top-Level TSDB Data Flow	344

Core-Level or Flat TSDB Data Flow

The core-level TSDB data flow applies to wrapped cores in a hierarchical DFT insertion flow. In the bottom-up insertion process, you process the wrapped cores before the top-level chip.

Refer to “[Tessent Shell Flow for Hierarchical Designs](#)” for details.

With the addition of boundary scan in the first DFT insertion pass and the exclusion of graybox modeling, this data flow also applies to flat and DFT-inserted designs unless the core includes embedded pad I/O macros that need boundary scan to be inserted as well. Refer to “[Tessent Shell Flow for Flat Designs](#)” for details.

Table 7-1. Core-Level TSDB Data Flow Inputs and Outputs

Task	Input	Output
DFT insertion: first pass Design ID for output: rtl1	<ul style="list-style-type: none"> • RTL netlist • Libraries • Required DFT signals • MemoryBIST requirements • For flat: boundary scan requirements also 	<ul style="list-style-type: none"> • Modified RTL + new RTL • ICL for IJTAG network • TCD: clocks, DFT signals • ICL for Memory + PDL • For flat: boundary scan also
DFT insertion: second pass Design ID for output: rtl2	<ul style="list-style-type: none"> • rtl1 design data • libraries • required DFT signals • EDT and OCC requirements 	<ul style="list-style-type: none"> • modified RTL + new RTL • ICL for IJTAG network • TCD: clocks, DFT signals • ICL for OCC and EDT + PDL
Synthesis Third-party tool	<ul style="list-style-type: none"> • output from write_design_import_script command (use rtl2 design data) 	<ul style="list-style-type: none"> • synthesized gate-level netlist
Scan chain insertion Design ID for output: gate	<ul style="list-style-type: none"> • synthesized gate-level netlist • scan modes • TCD, ICL, PDL from rtl2 	<ul style="list-style-type: none"> • scan-stitched gate-level netlist • ICL, PDL • TCD with scan modes • graybox model (not flat)
ATPG pattern generation	<ul style="list-style-type: none"> • gate scan-inserted design data • ATPG run name • scan mode 	<ul style="list-style-type: none"> • flat model • fault list • patterns database • TCD

During the first DFT insertion pass, you provide the required files for your DFT implementation. These files can include the RTL netlist, libraries for MemoryBIST insertion and boundary scan, and gate-level cells that require a Tessent cell library.

Tessent Shell generates the `dft_inserted_designs`, `instruments`, and `patterns` directories within the TSDB you specified. By default, Tessent Shell generates the TSDB in the current working directory if you do not specify a location. For details about these directories and the TSDB, refer to “[Tessent Shell Database \(TSDB\)](#)” in the *Tessent Shell Reference Manual*.

Tessent modifies the RTL netlist for the design into which the first-pass instrument hardware needs to be inserted. This hardware may include a MemoryBIST controller, BAP interface, and IJTAG network. In the flat DFT implementation, it may also include boundary scan and a TAP controller. Tessent Shell generates new RTL for the newly inserted DFT instruments.

In addition, Tessent Shell produces the TCD, ICL, and PDL for the design and the inserted instruments. As shown in [Figure 7-1](#), the design-level files and modified RTL are stored within the `dft_inserted_designs` subdirectory for this insertion pass (design ID “rtl1”). However, the

new RTL, TCD, ICL, and PDL files for each inserted instrument are stored in subdirectories within the instruments directory.

The patterns directory stores the patterns associated with the rtl1 design ID in an associated subdirectory.

Tip

i To facilitate data management, you can save each design (whether flat, core, sub-block, or chip) in its own TSDB. This is the recommended practice when using Tessent Shell for DFT insertion.

Figure 7-1. TSDB Data Flow, Core Level, First Insertion Pass

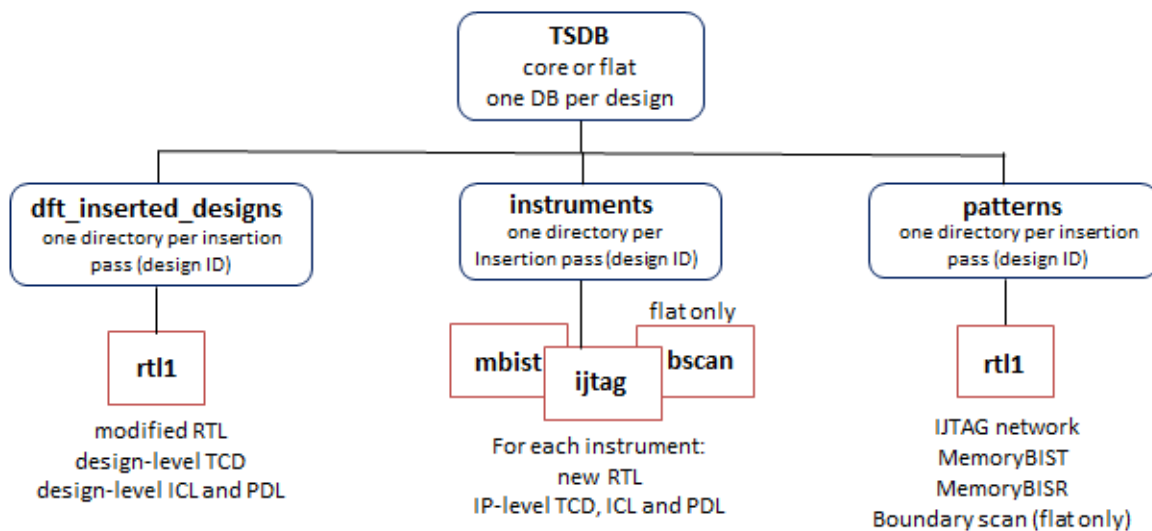


Figure 7-2 shows the data flow for the second DFT insertion pass. Tessent uses the design data that was saved as rtl1 in the first pass as input for the second pass.

The relevant design RTL, TCD, ICL, and PDL files from the first DFT insertion pass are automatically read in when you specify the read_design command as described in “Loading the Design”. You only need to supply a library, if required, and any DFT input requirement for the DFT instruments you are inserting during this pass.

The hardware you insert in this pass, such as EDT and OCC, is stored in the instruments directory. Separate directories are created for each type of hardware inserted in each insertion pass.

The patterns directory stores the patterns associated with the rtl2 design ID in an associated subdirectory.

Figure 7-2. TSDB Data Flow, Core Level, Second Insertion Pass

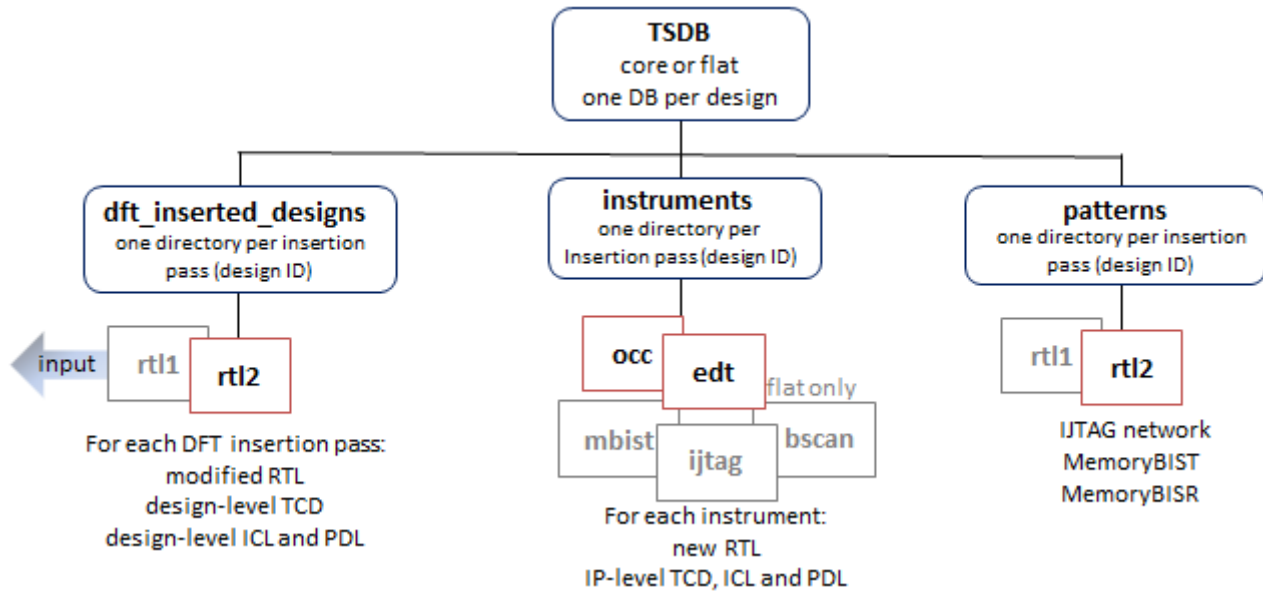


Figure 7-3 shows the data flow for scan chain insertion. After synthesis, which you perform using a third-party tool, you have a synthesized gate-level netlist. This netlist is the input for scan chain insertion along with the design-level TCD, ICL, and PDL from the rtl2 design generated during the second DFT insertion pass.

For wrapped cores, Tessent performs wrapper analysis along with scan chain insertion, whereas for flat designs, Tessent performs scan chain replacement and stitching. For information about using Tessent Scan for scan insertion, refer to “[Internal Scan and Test Circuitry Insertion](#)” in the *Tessent Scan and ATPG User’s Manual*.

Scan insertion does not insert instruments, so the instruments and patterns directories are not utilized in this step.

Figure 7-3. TSDB Data Flow, Core Level, Scan Insertion

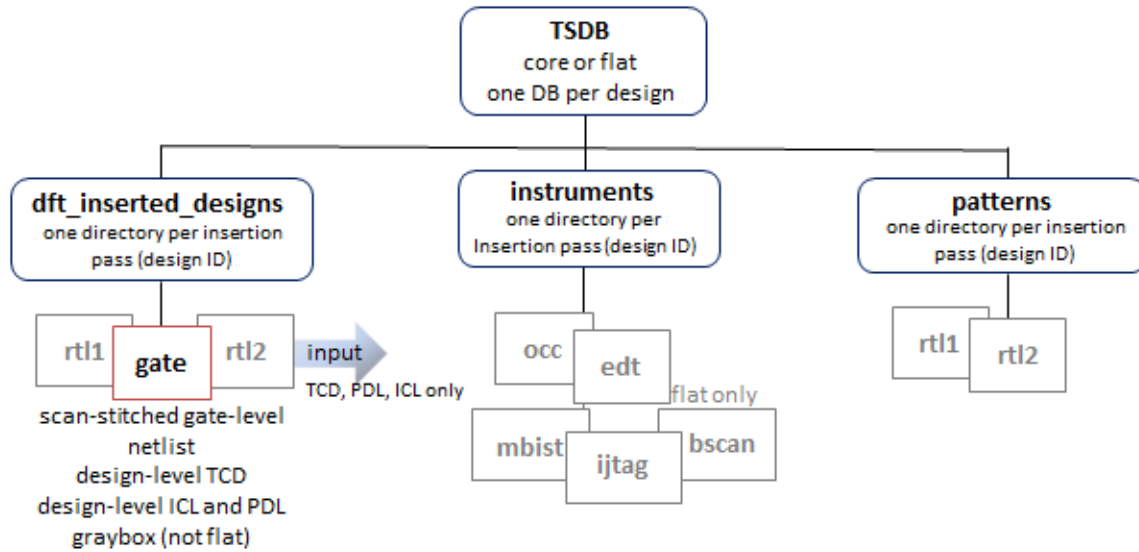
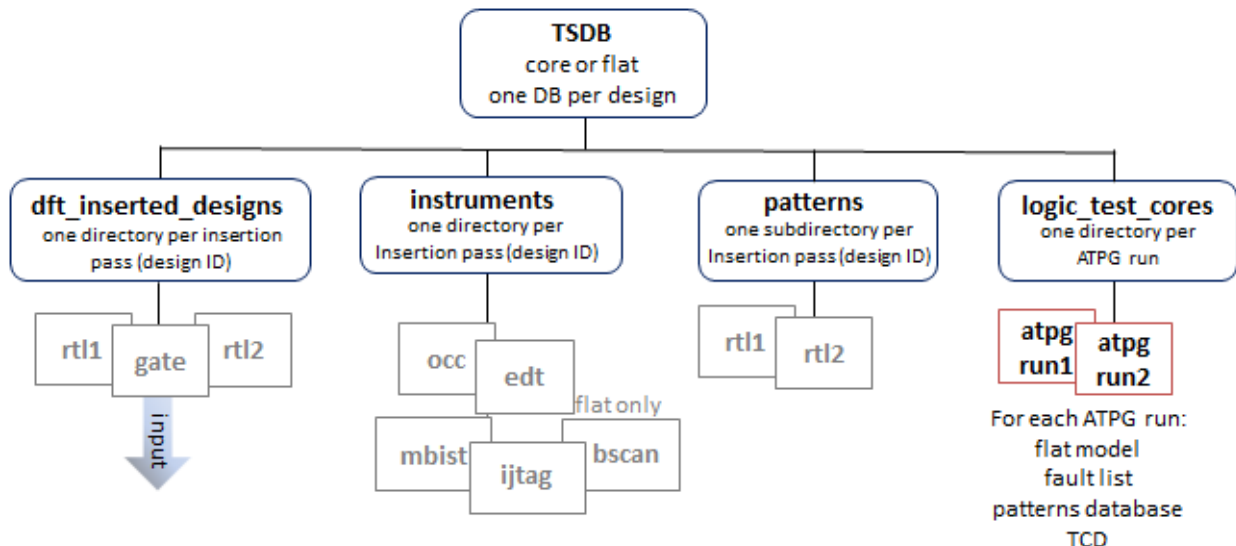


Figure 7-4 shows that the input to ATPG is the scan-inserted netlist and all the supporting files such as TCD, ICL, and PDL files that were stored in the TSDB during the scan insertion pass (design_id “gate”). Tessent creates the logic_test_cores directory, which stores the output pattern data for each ATPG run, which can include runs for various test types and associated fault models as described in “[Fault Modeling Overview](#)” in the *Tessent Scan and ATPG User’s Manual*.

Before generating patterns for wrapped cores, Tessent creates a graybox model of the core. This model is stored using the same design ID as the one created during scan insertion (design ID “gate”), so that at the top-level you can either use the full design view or graybox view of the wrapped core. Refer to “[Performing ATPG Pattern Generation: Wrapped Core](#)” for more information.

Figure 7-4. TSDB Data Flow, Core Level, Pattern Generation



Top-Level TSDB Data Flow

The top-level TSDB data flow applies to the hierarchical RTL and scan DFT insertion flow. In the hierarchical DFT insertion flow, you insert boundary scan and MemoryBIST, if present, at the top level of a chip during the first DFT insertion pass.

At the chip level, the core-level design data that was stored in the core-level TSDBs is used for ATPG pattern retargeting. Refer to “[RTL and Scan DFT Insertion Flow for the Top Chip](#)” for details.

Table 7-2. Top-Level TSDB Data Flow Inputs and Outputs

Task	Input	Output
DFT insertion: first pass design ID for output: rtl1	<ul style="list-style-type: none"> • RTL netlist • libraries • required DFT signals • boundary scan requirements • TSDBs, lower core design data 	<ul style="list-style-type: none"> • modified RTL + new RTL • ICL for IJTAG network • TCD: clocks, DFT signals • ICL for boundary scan + PDL
DFT Insertion: second pass design ID for output: rtl2	<ul style="list-style-type: none"> • rtl1 design data • libraries • required DFT signals • EDT and OCC requirements 	<ul style="list-style-type: none"> • modified RTL + new RTL • ICL for IJTAG network • TCD: clocks, DFT signals • ICL for OCC and EDT + PDL
Synthesis third-party tool	<ul style="list-style-type: none"> • output from write_design_import_script command (use rtl2 design data) 	<ul style="list-style-type: none"> • synthesized gate-level netlist
Scan chain insertion design ID for output: gate	<ul style="list-style-type: none"> • synthesized gate-level netlist • scan modes • TCD, ICL, PDL from rtl2 	<ul style="list-style-type: none"> • scan-stitched gate-level netlist • ICL, PDL • TCD with scan modes
ATPG pattern generation (flat)	<ul style="list-style-type: none"> • gate scan-inserted design data • ATPG run name • scan mode 	<ul style="list-style-type: none"> • flat model • fault list • patterns database • TCD
ATPG pattern generation (core level)	<ul style="list-style-type: none"> • gate scan-inserted design data • ATPG run name • scan mode • wrapped core ATPG pattern data, retargeted 	<ul style="list-style-type: none"> • flat model • fault list • patterns database • TCD

Figure 7-5 shows the data flow for the first DFT insertion pass. Tessent modifies the RTL netlist for the design and generates new RTL for the boundary scan and MemoryBIST (if inserted)

hardware. In addition, it produces the TCD, ICL, and PDL for the design and the inserted instruments.

For integration at the top level, the scan-inserted design data and the interface views from the wrapped cores are used. This is done by opening the core TSDB directories and using the read_design command to read in the graybox model and the TCD, ICL, and PDL files.

The patterns directory stores the patterns associated with the rtl1 design ID in an associated subdirectory.

Tip

i To facilitate data management, you can save each design (whether flat, core, sub-block, or chip) in its own TSDB. This is the recommended practice when using Tessent Shell for DFT insertion. You should have one TSDB per design.

Figure 7-5. TSDB Data Flow, Top Level, First Insertion Pass

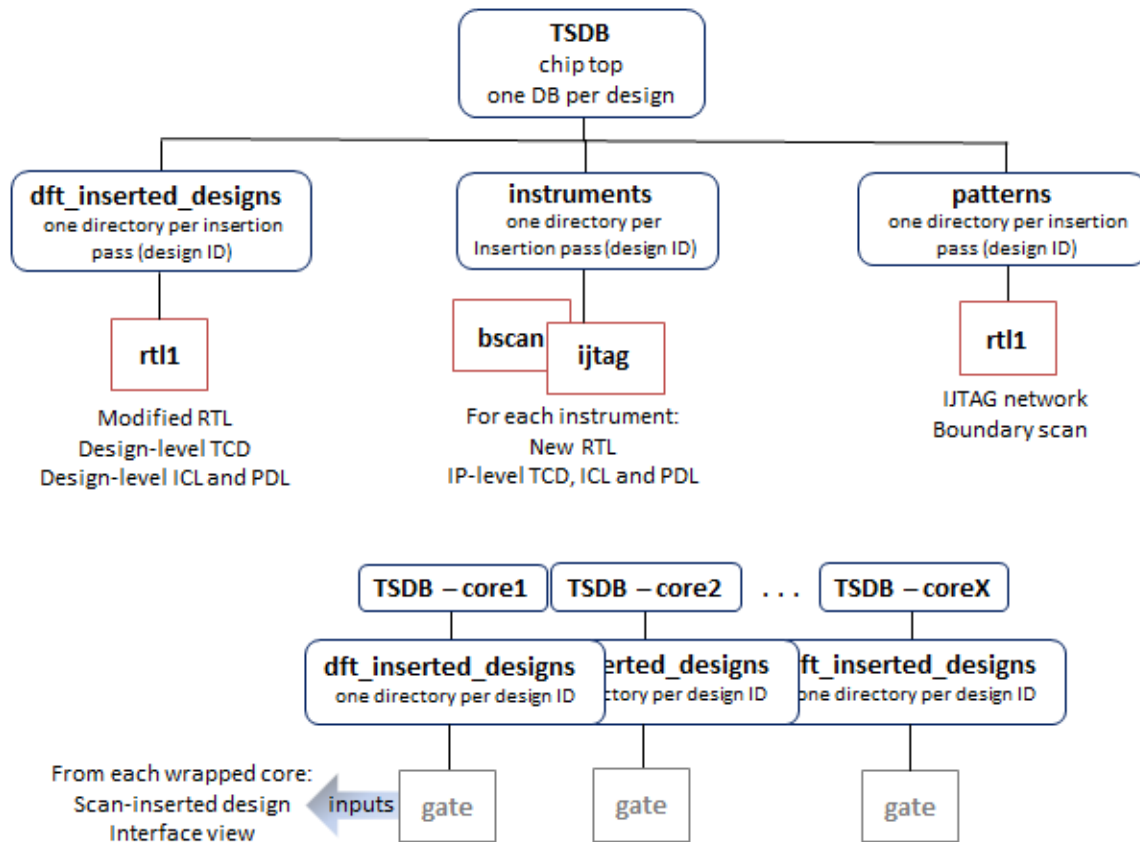


Figure 7-6 shows the data flow for the second DFT insertion pass. In addition to the other input requirements that you provide as shown in Table 7-2, Tessent uses the design data that was saved as rtl1 in the first pass, the gate scan-inserted design data, and graybox models from the wrapped cores.

Tessent saves the output design data for the EDT and OCC hardware in their applicable instruments subdirectories. The design-level TCD, ICL, PDL, and modified RTL that includes the EDT, OCC, and IJTAG network is placed in the `dft_inserted_designs` subdirectory for this insertion pass (`rtl2`).

The `patterns` directory stores the patterns associated with the `rtl2` design ID in an associated subdirectory.

Figure 7-6. TSDB Data Flow, Top Level, Second Insertion Pass

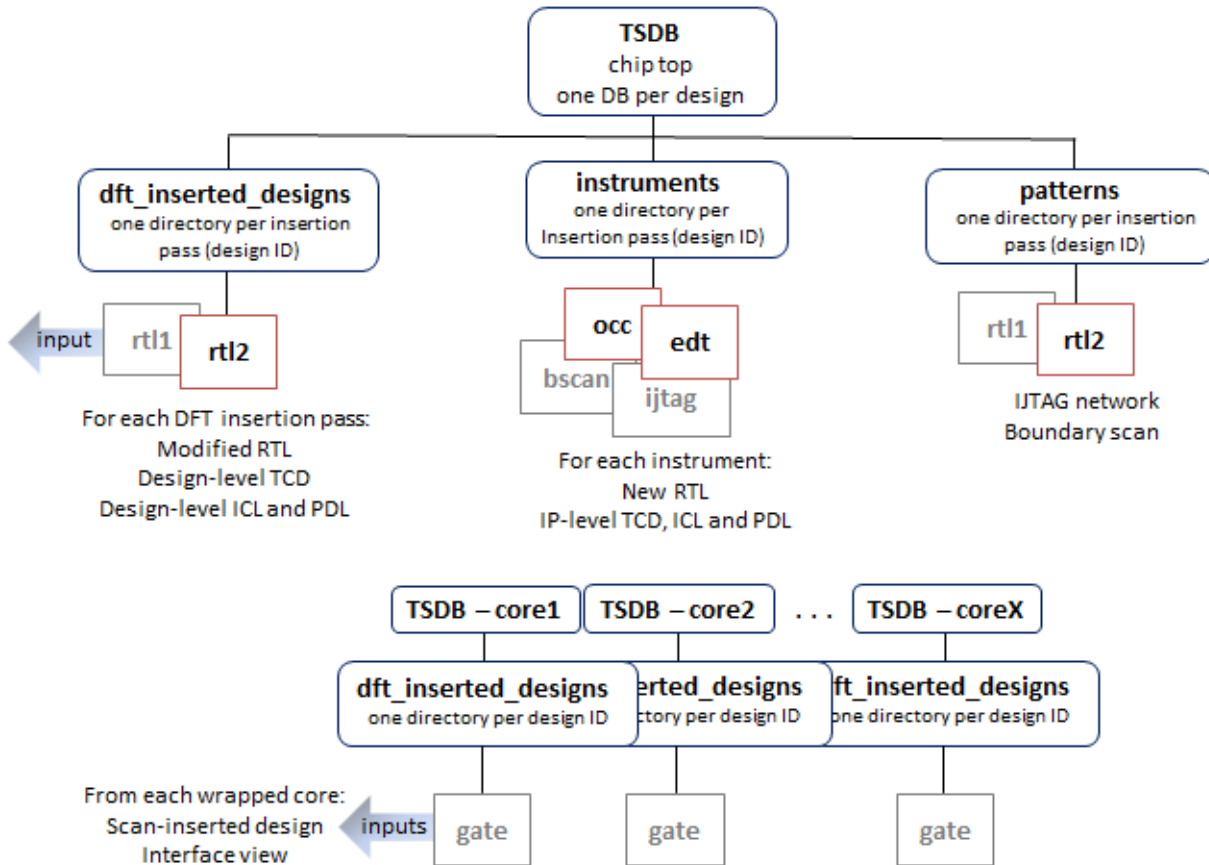


Figure 7-7 shows the data flow for scan chain insertion. Scan insertion does not insert instruments, so the `instruments` and `patterns` directories are not utilized in this step.

Figure 7-7. TSDB Data Flow, Top Level, Scan Insertion

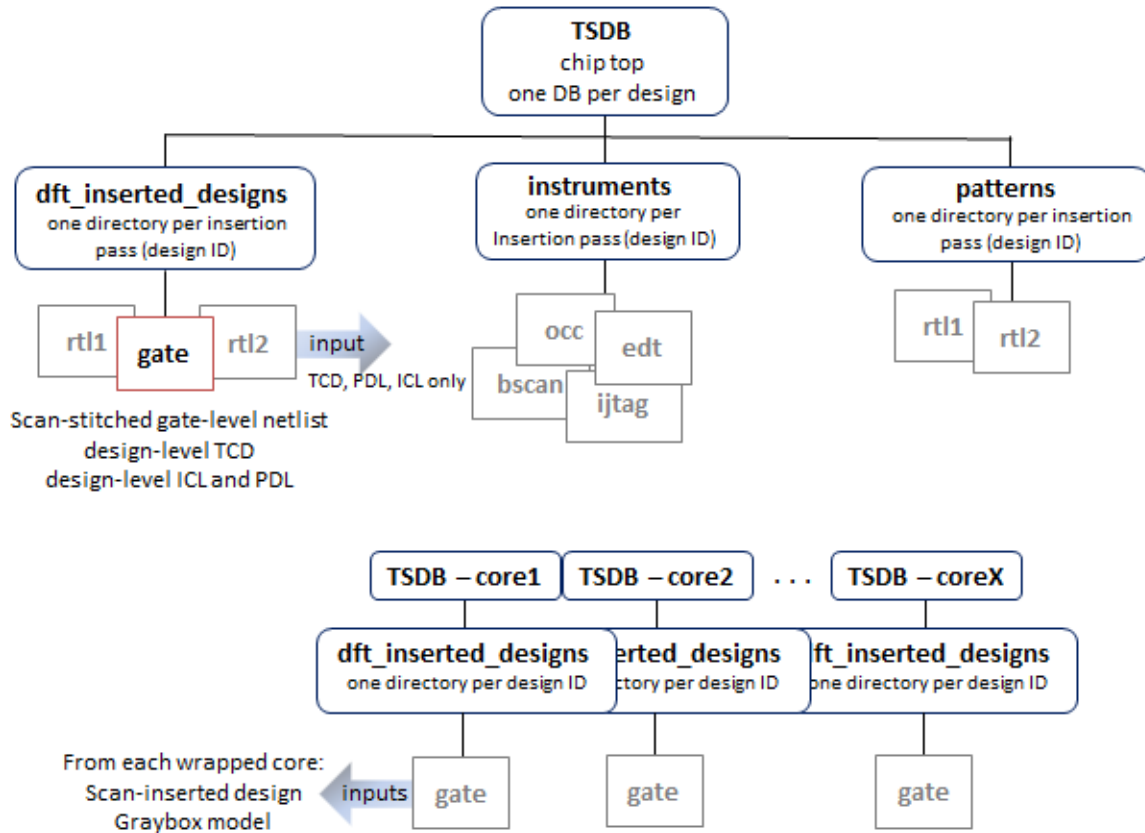
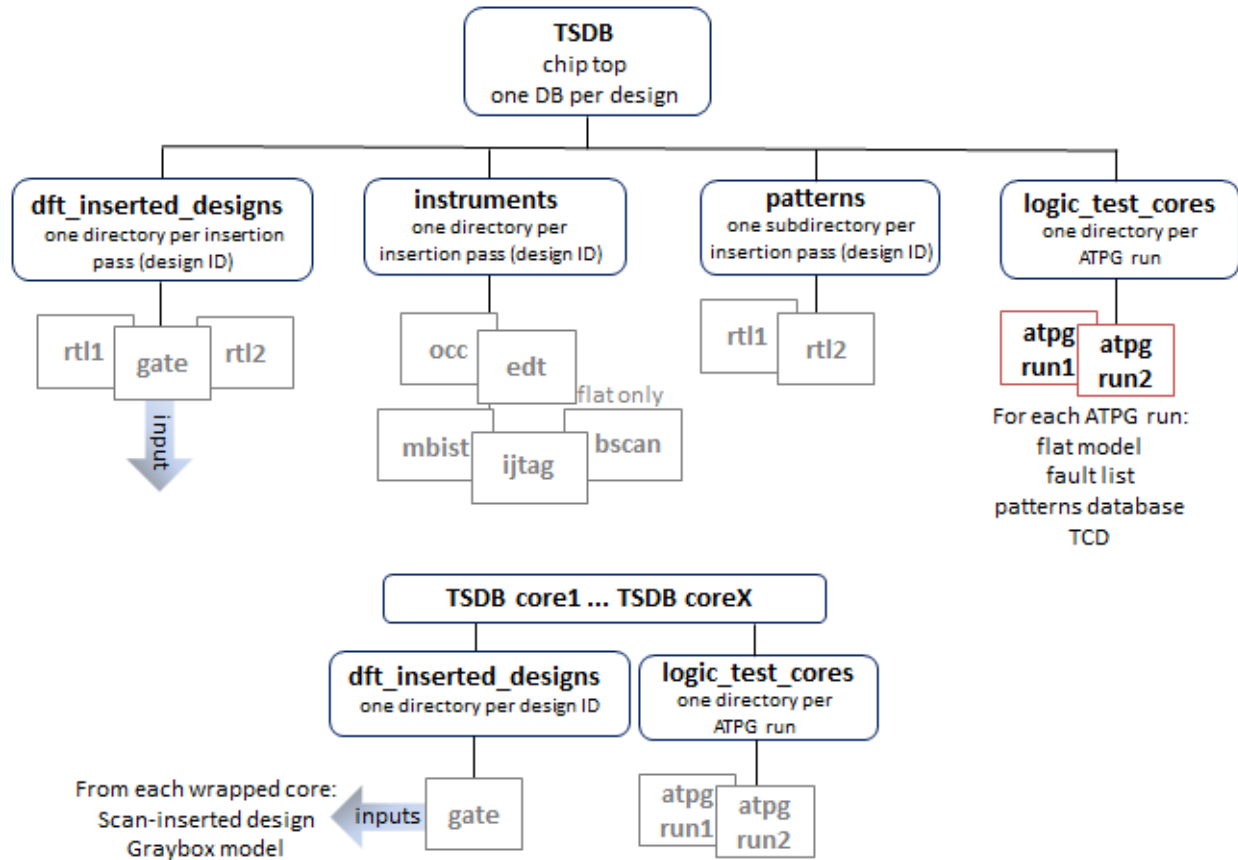


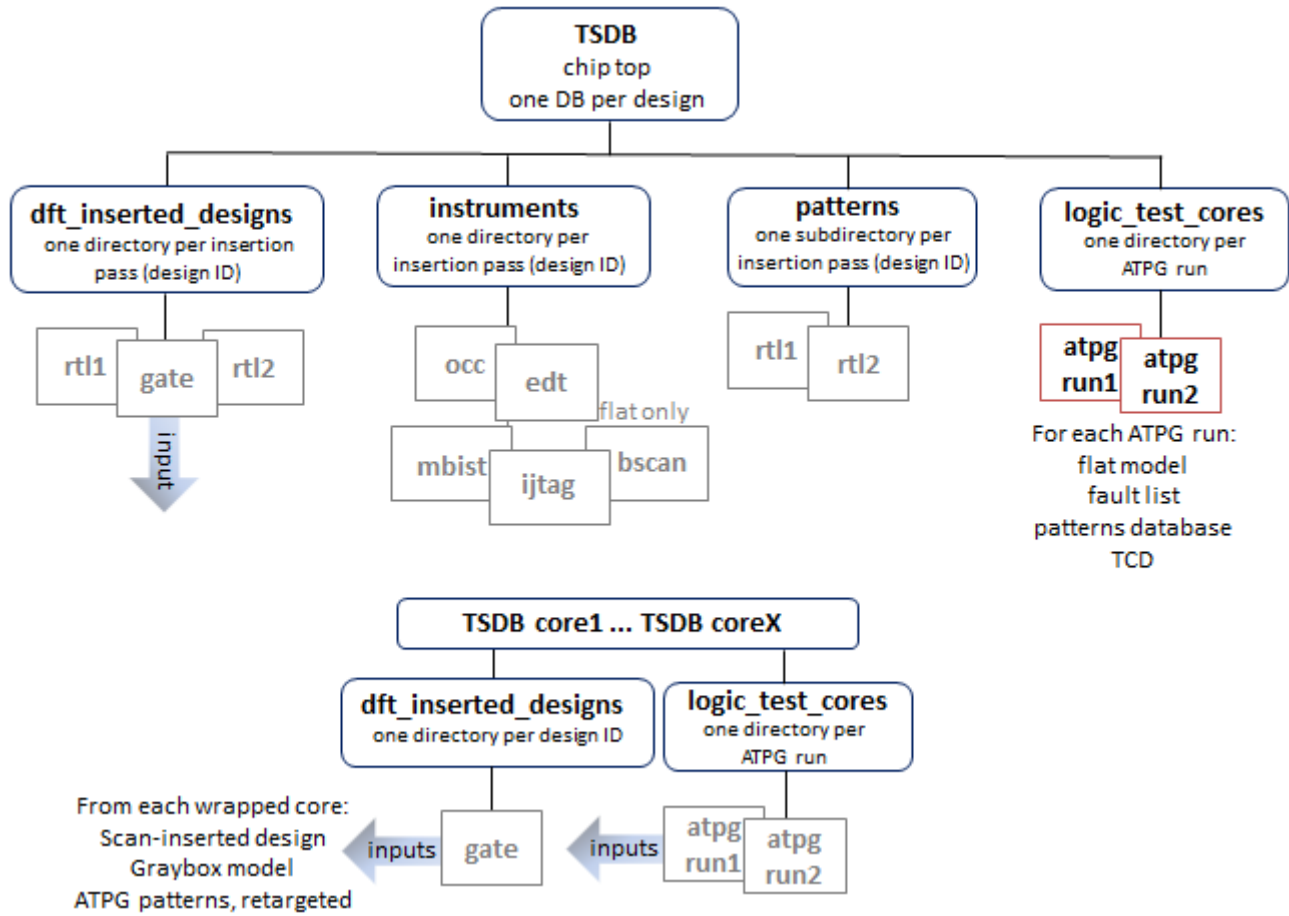
Figure 7-8 shows that output from ATPG pattern generation gets stored in the logic_test_cores directory. As inputs, Tessent uses the scan-inserted design data for the chip and for the cores.

Figure 7-8. TSDB Data Flow, Top Level, ATPG Pattern Generation



As the final step for the top level in a hierarchical design, you perform ATPG pattern retargeting of the core ATPG patterns as shown in “[Top-Level ATPG Pattern Generation Example](#)”. [Figure 7-9](#) shows that you read in the ATPG patterns from the logic_test_cores directory from each of the core TSDB directories and the scan-inserted design data for the chip and for the cores.

Figure 7-9. TSDB Data Flow, Top Level, ATPG Pattern Generation With Pattern Retargeting



Chapter 8

Streaming Scan Network (SSN)

Tessent Streaming Scan Network (SSN) is a bus-based scan data distribution architecture for use with Tessent TestKompress. The design of SSN addresses common DFT challenges in testing system-on-chip (SoC) designs, such as planning effort, tiled designs with abutment, test cost, and routing and timing closure.

SSN decouples test delivery and core-level DFT requirements. This means that instance core-level compression can be defined completely independently of chip I/O limitations. It enables programmatic decisions, such as selecting which cores will be tested concurrently. In a traditional pin-muxed approach, these options would be hard-wired.

This section describes the recommended SSN work flows, including how to use SSN to effectively test designs with identical cores and how to use different clock networks with SSN.

Tessent SSN Workflows	352
Block-Level SSN Insertion and Verification	355
Top-Level SSN Insertion and Verification	380
Advanced Topics	405
Streaming-Through-IJTAG Scan Data	405
On-Chip Compare With SSN	408
Types of Clock Networks To Use With SSN	417
Broadcast to Identical SSN Datapaths	421
Yield Statistics on ATE With SSN	422
Manufacturing Patterns With SSN	427
Signoff Patterns With SSN	444
SSN SDC Constraints in the Design Flow	453
Tessent SSN Examples and Solutions	484
Third-Party OCCs With SSN	484
SSN Frequently Asked Questions	486
SSN Limitations	488

Tessent SSN Workflows

There are several SSN Workflows, which are based on the Tessent Shell Flow for Hierarchical Designs.

In the general SSN Workflow, you perform a bottom-up DFT insertion for each physical block followed by DFT insertion at the parent level. You continue this bottom-up flow until you have reached the top level of the design.

The SSN Workflow description focuses on the differences from the base flows described in the [Tessent Shell Workflows](#) section of this manual. The descriptions of the Workflows assume you understand the information contained in the following sections. It is recommended that you read them first if you are new to the Tessent Workflows.

- [“Tessent Shell Flow for Hierarchical Designs”](#)
- [“How the DFT Insertion Flow Applies to Hierarchical Designs”](#)
- [“Hierarchical DFT Terminology”](#)

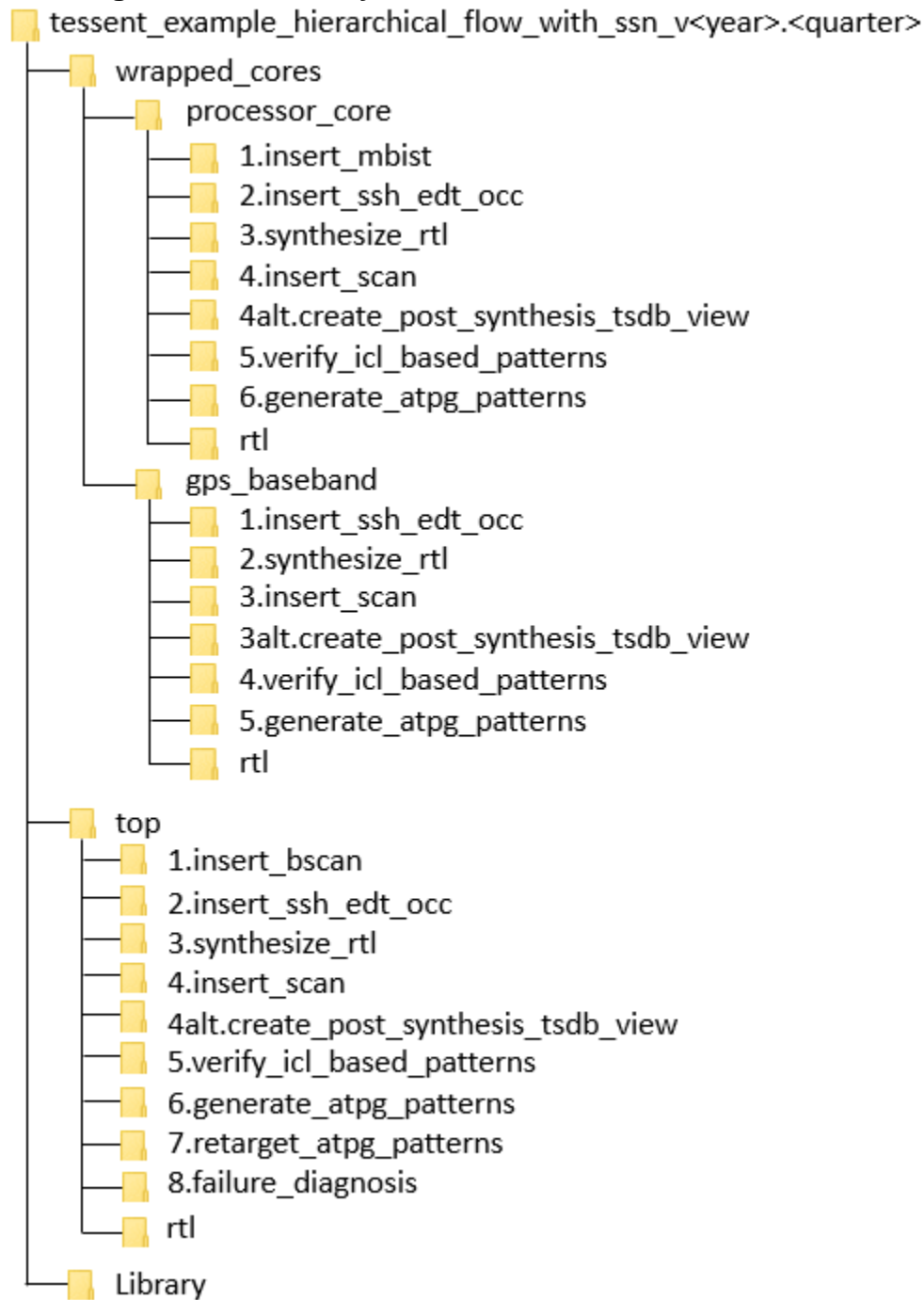
Reference Testcase

The Tessent Shell release tree includes a testcase to accompany the block- and top-level workflows described in the following sections. From within Tessent Shell, use the **getenv TESSENT_HOME** command to find the installation tree. The testcase is in *\$TESSENT_HOME/share/UsageExamples* in the following file:

tessent_example_hierarchical_flow_with_ssn_v<year>.<quarter>.tgz

[Figure 8-1](#) shows the structure of the directories in the testcase. You can run each step of the workflow in one of these directories.

Figure 8-1. Directory Structure of Reference Testcase



Block-Level SSN Insertion and Verification	355
First DFT Insertion Pass: Performing Block-Level MemoryBIST	357
Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN	357
Synthesis for Block-Level Insertion	364
Creating the Post-Synthesis TSDB View (Block-Level)	365
Performing Scan Chain Insertion With Tessent Scan (Block-Level).....	367


Verifying the ICL-Based Patterns After Synthesis (Block-Level)	370
Generating Block-Level ATPG Patterns	373
Top-Level SSN Insertion and Verification	380
First DFT Insertion Pass: Performing Top-Level MemoryBIST	382
Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN	385
Synthesis for Top-Level Insertion	393
Creating the Post-Synthesis TSDB View (Top-Level).	394
Performing Scan Chain Insertion With Tessent Scan (Top-Level)	394
Verifying the ICL-Based Patterns After Synthesis (Top-Level)	395
Generating Top-Level ATPG Patterns	395
Retargeting ATPG Patterns	397
Performing Reverse Failure Mapping for SSN Pattern Diagnosis	400

Block-Level SSN Insertion and Verification

This section presents the block-level Tessent Workflow for SSN by highlighting the minor differences from the normal workflow where SSN is not present.

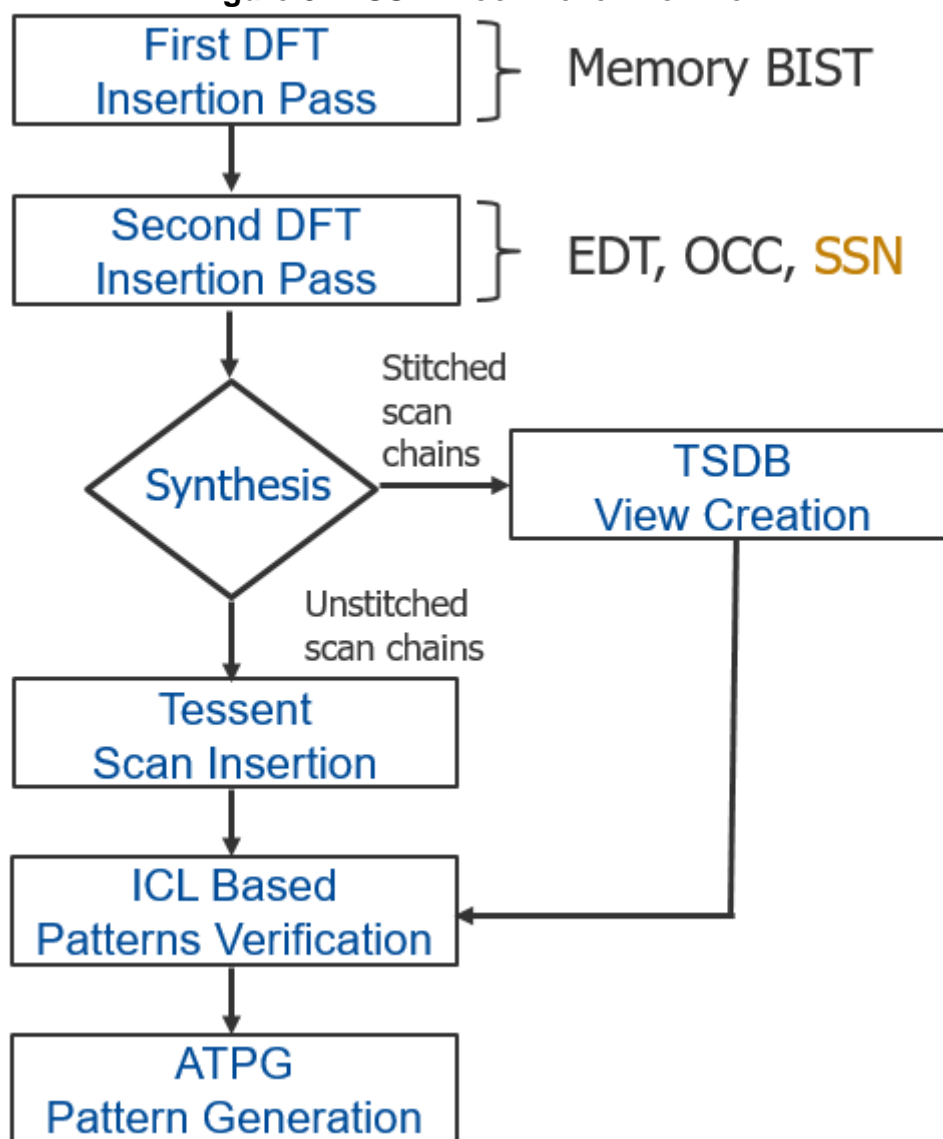
The SSN Workflow for block-level insertion contains minor differences in the dofiles relative to the flow in the “[RTL and Scan DFT Insertion Flow for Physical Blocks](#)” section of this manual. These differences include minor additions and deletions required in the dofiles specific to each step. In [Figure 8-1](#) on page 353 in the section “[Tessent SSN Workflows](#)”, the *wrapped_cores* subdirectories contain the block-level SSN flow. The text preceding the figure explains how to find the testcase within the release directory.

Note

 This procedure assumes your design meets the prerequisites of the hierarchical flow, as described in “[DFT Architecture Guidelines for Hierarchical Designs](#)”. Additionally, each physical block must have a full OCC with a shift clock injection mux and [shift_only_mode](#).

The following figure shows an identical block-level workflow used to process a child physical block to when you are not using SSN. As highlighted in the figure, you insert SSN into the design during the second DFT insertion pass. In this same step, you add a second smaller EDT to the design for the wrapper chains during external test. Click the boxes in the flow diagram to see the section describing each step.

Figure 8-2. SSN Block-Level Workflow

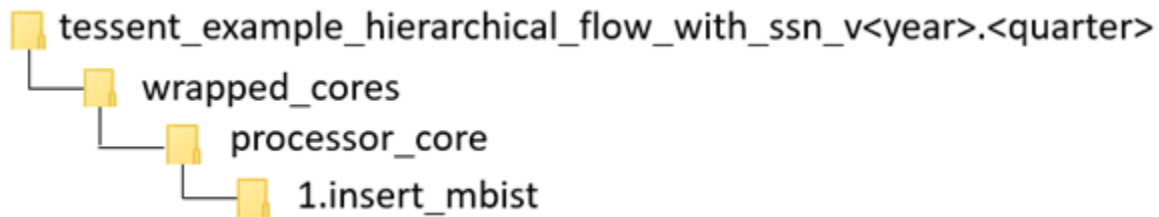


First DFT Insertion Pass: Performing Block-Level MemoryBIST.....	357
Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN.....	357
Synthesis for Block-Level Insertion	364
Creating the Post-Synthesis TSDB View (Block-Level).....	365
Performing Scan Chain Insertion With Tessent Scan (Block-Level)	367
Verifying the ICL-Based Patterns After Synthesis (Block-Level).....	370
Generating Block-Level ATPG Patterns.....	373

First DFT Insertion Pass: Performing Block-Level MemoryBIST

The first DFT insertion pass is identical to the Tessent Shell Flow for Hierarchical designs.

You can perform this step in the following directory of the [Reference Testcase](#):



Procedure

The first DFT insertion pass inserts the non-scan DFT elements, such as memory BIST and IJTAG. It happens before the second insertion pass inserts the logic test elements. Using SSN in the second DFT insertion pass does not affect the first DFT insertion pass. At the top level, you must equip the boundary scan cells with auxiliary input and output pins to connect the SSN bus. This change to the first DFT insertion pass for the top level is described in “[First DFT Insertion Pass: Performing MemoryBIST and Boundary Scan](#)” on page 116. For complete information about the current step at the block level, see “[First DFT Insertion Pass: Performing Block-Level MemoryBIST](#)” on page 147.

Examples

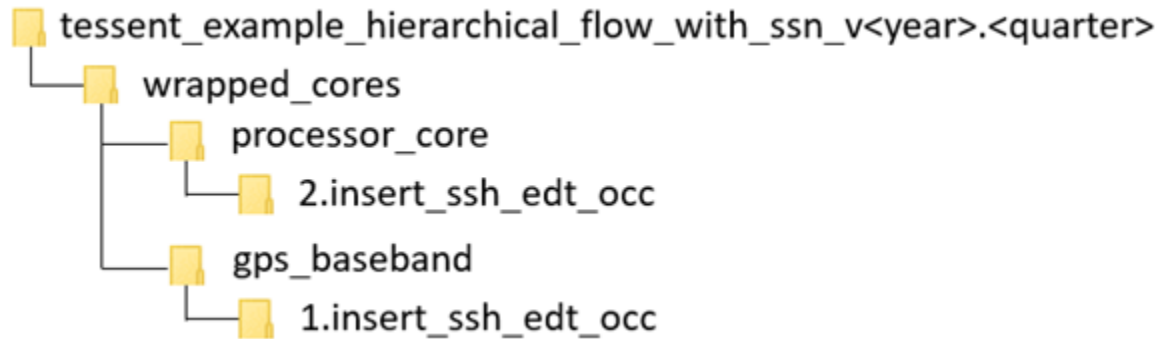
Refer to the example dofile *run_mbist_insertion* in the directory shown in the “Referenced Testcase Step” section earlier in this topic.

Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN

In the second DFT insertion pass, you insert the block-level EDT, OCC, and SSN elements.

This section explains the slight modifications that you do to the standard flow when you insert the SSN, EDT, and OCC into the physical block level. For a complete description of this step in the standard flow, refer to “[Second DFT Insertion Pass: Inserting Block-Level EDT and OCC](#)” on page 149.

You can perform this step in the following directories of the [Reference Testcase](#):



Notes About This Procedure

- Similar to [EDT](#) and [OCC](#), the tool creates the SSN hardware based on the SSN wrapper of the DftSpecification. For a description of the SSN wrapper, see the “[SSN](#)” wrapper description in the *Tessent Shell Reference Manual*.
- When the tool creates the SSH, EDT, and OCC elements at the same time with a single invocation of the [process_dft_specification](#) command, the tool automates the connections between the SSH, EDT, and OCC instances.
- As with the normal flow, after the insertion of the SSN and the other logic test elements, you must simulate the ICLNetwork pattern to verify the correct IJTAG access to all inserted logic test elements, including the SSN elements. Following a successful simulation of the ICLNetwork pattern, you must simulate the SSN Continuity pattern to verify that the SSN datapath is correctly integrated into the design. The tool automates the ICL verification patterns and the SSN continuity patterns as part of the [create_patterns_specification](#) and [process_patterns_specification](#) commands. During validation of the SSN datapath, you are responsible to use the iProc command to write to the [Multiplexer](#) node to configure the datapath for which you want to verify continuity, as explained in Step 7 in the following procedure. You must complete simulation of both the ICLNetwork pattern and SSN continuity pattern before moving to the next step of the DFT flow.
- To use third-party OCC with SSN, refer to the topic “[Third-Party OCCs With SSN](#)” on page 484 in the “[Tessent SSN Examples and Solutions](#)” section.

To insert the logic test elements with SSN, use the following procedure to modify the dofile described in “[Second DFT Insertion Pass: Inserting Block-Level EDT and OCC](#)” on page 149.

Lint Checks for the SSN Hardware

RTL output for SSN includes waivers for SpyGlass® to avoid lint errors. These waivers include comments detailing why the rules are waived and use the following format:

```
// <explanatory comment>
// spyglass disable_block <rule list>
// <waived code>
// spyglass enable_block <rule list>
```

The *<rule list>* uses SpyGlass naming for the rules that are being waived (for example, “W116 W164a”).

Prerequisites

- SSN requires that each physical block have a full OCC with a clock injection multiplexer and support of the [shift_only_mode](#) feature.

Procedure

1. Remove the [add_dft_signals](#) commands for the scan signals `edt_clock`, `edt_update`, `scan_en`, `shift_capture_clock`, and `test_clock`.

These signals are not needed because they are sourced by the [ScanHost](#) node during scan test. If you want to have your legacy channel access mechanism coexist with SSN for your first few designs, see [Example 2 in the ScanHost](#) reference page.

- a. Delete this line:

```
add_dft_signals scan_en edt_update test_clock -source_node \
{ scan_enable edt_update test_clock_u }
```

- b. Delete this line:

```
add_dft_signals edt_clock shift_capture_clock \
-create_from_other_signals
```

2. Update the “`create_dft_specification -sri_sib_list`” option to include SSN:

```
set spec [create_dft_specification -sri_sib_list {edt occ ssn} ]
```

This creates a dedicated IJTAG [Sib](#) node to serve as the host for the IJTAG interfaces of the [ScanHost](#) and [Multiplexer](#) SSN nodes.

3. Update the comment for the [report_config_syntax](#) command to include SSN as an option to view the syntax of the SSN DftSpecification:

```
// Use report_config_syntax DftSpecification/edt|occ|ssn to see full syntax.
```

4. Update the DftSpecification to include the [SSN](#) wrapper to create one or more SSN [Datapaths](#) with the nodes you want to insert in them. The following example creates a

32-bit wide Datapath with a [ScanHost](#) node preceded and followed by a [Pipeline](#) node. The SSN reference page contains several example of different SSN DftSpecifications:

```
SSN {
  ijtag_host_interface : Sib(ssn);
  DataPath(main) {
    output_bus_width      : 32;
    Pipeline(out) {
    }
  }
  ScanHost(1) {
  }
  Pipeline(in) {
  }
}
```

5. Update the EDT wrapper of the DftSpecification to include mode enables for the EDT, and add a second EDT for the wrapper chains to be used during external test mode.
 - a. Change the name of the EDT Controller from “Controller (c1)” to “Controller (c1_int)” to indicate that this EDT controller is used for internal test mode and is different from the second EDT that is used in external test mode.
 - b. Add a Connections wrapper within the EDT “Controller (c1_int)” wrapper with the [mode_enables](#) property to define the DFT Signal “int_edt_mode” as the mode enable for this EDT. This DFT Signal was added previously in the dofile and is used in the muxing logic that selects this EDT’s channel outputs to the SSH during the internal test mode.

```
EDT {
  ijtag_host_interface : Sib(edt);
  Controller (c1_int) {
    longest_chain_range      : 70, 80;
    scan_chain_count         : 20;
    input_channel_count      : 3;
    output_channel_count     : 3;
    Connections {
      mode_enables           : DftSignal(int_edt_mode);
    }
  }
}
```

- c. Add a second EDT controller for the wrapper chains to be used during external test mode. The “ext_edt_mode” DFT signal is used as the mode enable for this EDT. This DFT signal was added previously in the dofile and is used in the muxing logic that selects this EDT’s channel outputs to the SSH during the external test mode.



```

Controller (cl_ext) {
    longest_chain_range      : 70, 80;
    scan_chain_count        : 2;
    input_channel_count     : 1;
    output_channel_count    : 1;
    Connections {
        mode_enables        : DftSignal(ext_edt_mode);
    }
}

```

6. The SSN network that the specification just inserted is checked and extracted within the ICL model using the same [extract_icl](#) command that already checks and extracts the IJTAG network description. These design rule checks run automatically, and there is generally no need for additional input, except that you must specify the “-create_ijtag_graybox on” switch to create the IJTAG graybox in the TSDB.
7. The `create_patterns_specification` command creates a pattern specification for the ICL network that includes the SSN logic that was just inserted and all other element types, such as the Memory BIST logic (if present). The SSN continuity pattern is also created as part of the `create_patterns_specification` process and verifies the SSN datapath is properly connected. This is a mandatory verification step. The following example contains no [Multiplexer](#) node to configure at the block level. If you have such nodes at the block level, see how they are handled at the bottom of the dofile example in the section “[Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN](#)” on page 385. Because you created an IJTAG graybox, the tool creates the ICL verification patterns wrapper and SSN continuity patterns wrapper using the IJTAG graybox view for both simulations.

Note

 You can also create the SSN continuity pattern with the low-level [create_ssn_continuity_patterns](#) command, as shown at the bottom of the example dofile found at the end of this section.

Examples

The following is a dofile for the second DFT pass. You can also find it in the `run_ssh_edt_occ_insertion` scripts in the testcase directories shown in the “Referenced Testcase Step” section earlier in this topic. The testcase also includes the `run_continuity_sims` script that shows how to simulate the SSN continuity patterns.

```

# Set the context to insert DFT into RTL-level design
# Define a new design ID
set_context dft -rtl -design_id rtl2

# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_outdir

# Read in the Tessent cell library
read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
# Read in memory verilog model
read_verilog ../../../../library/memories/*.v -exclude_from_file_dictionary

```

Streaming Scan Network (SSN) Block-Level SSN Insertion and Verification

```
# Use read_design with the design ID from the first DFT insertion pass
read_design processor_core -design_id rtl1 -verbose
set_current_design processor_core

# DFT Signal used for logic test
add_dft_signals ltest_en
# DFT Signal used to test memories with multi-load ATPG patterns
add_dft_signals memory_bypass_en
# DFT Signal required to test STI network during logic test
add_dft_signals tck_occ_en
# DFT Signals required for hierarchical DFT and used during scan insertion
add_dft_signals int_ltest_en ext_ltest_en int_mode, ext_mode, \
    int_edt_mode ext_edt_mode

report_dft_signals

# Run pre-DFT DRC rules
set_dft_specification_requirements -logic_test on
check_design_rules

# Create and report a DFT Specification
set_spec [create_dft_specification -sri_sib_list {edt occ ssn} ]
# Use report_config_syntax DftSpecification/edt|occ|ssn to see full syntax
report_config_data $spec

# Specify the logic to be created with the EDT, OCC, and SSN wrappers.
# The connection between the EDT, OCC, and SSH created by the too.
# Modify the below specification to your specific design requirements.
```

```

read_config_data -in_wrapper $spec -from_string {
  Occ {
    ijtag_host_interface : Sib(occ);
    include_clocks_in_icl_model : on;
    Controller(dco_clk) {
      clock_intercept_node      : dco_clk;
    }
  }
  SSN {
    ijtag_host_interface : Sib(ssn);
    DataPath(1) {
      output_bus_width      : 32;
      Pipeline(2) {
      }
      ScanHost(1) {
      }
      Pipeline(1) {
      }
    }
  }
  EDT {
    ijtag_host_interface : Sib(edt);
    Controller (c1_int) {
      longest_chain_range    : 70, 80;
      scan_chain_count       : 20;
      input_channel_count    : 3;
      output_channel_count   : 3;
      Connections {
        mode_enables         : DftSignal(int_edt_mode);
      }
    }
    Controller (c1_ext) {
      longest_chain_range    : 70, 80;
      scan_chain_count       : 2;
      input_channel_count    : 1;
      output_channel_count   : 1;
      Connections {
        mode_enables         : DftSignal(ext_edt_mode);
      }
    }
  }
}

# Display the content of the DftSpecification just defined above
report_config_data $spec

# Generate the hardware
process_dft_specification

# Extract IJTAG network and create ICL file for core level
extract_icl -create_ijtag_graybox on

# Write updated RTL into this new file to elaborate and synthesize later
write_design_import_script -use_relative_path_to . for_dc_synthesis.tcl -
replace

```

```

# Generate test bench for ICLNetwork and Continuity patterns
# Generate ICL Verify patterns. Use the variable to update it if needed.
set spec [create_patterns_specification]
process_patterns_specification

# Point to simulation libraries and run the Verilog simulation
set_simulation_library_sources \
  -v ../../../../library/standard_cells/verilog/adk.v \
  -v ../../../../library/memories/*.v
run_testbench_simulations

exit

```

The following is an alternate way to create the SSN continuity pattern. Refer to Step 7 in the Procedure section of this topic for an explanation of how to create the SSN continuity pattern.

```

#
# Create the continuity pattern
#
# Define SSN bus clock. If you are using time multiplexing
# you must specify the -freq_multiplier switch
add_clocks ssn_bus_clock

# check design rules and transition to analysis mode
check_design_rules

# set ijtag period
set_ijtag_retargeting_options -tck_period 10

# set ssn bus clock period
set_load_unload_timing_options -usage ssn -ssn_bus_clock_period 5

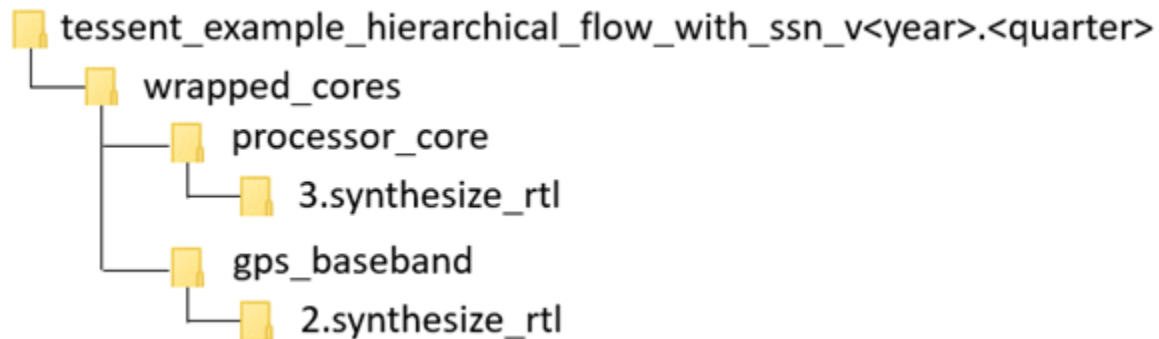
# Create the SSN continuity pattern
open_pattern_set ssn_continuity -usage ssn
  create_ssn_continuity_patterns
close_pattern_set
# Write simulation testbench
write_patterns patterns/ssn_continuity.v -verilog -replace \
  -parameter_list {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1}

```

Synthesis for Block-Level Insertion

Use the following dofile changes to synthesize the DFT-inserted RTL block.

You can perform this step in the following directories of the [Reference Testcase](#):



In the dofile of the section “[Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN](#)” on page 357, you used the `write_design_import_script` command to export a design load script for use in your synthesis tool.

Refer to the section “[Synthesis Guidelines for RTL Designs with Tessent Inserted DFT](#)” on page 779 for guidelines on how to perform the synthesis step. Also, see the section “[Example Scripts using Tessent Tool-Generated SDC](#)” on page 760 for example scripts specific to various synthesis tools.

When synthesis completes, it produces a file containing the concatenated netlist of the physical block you just synthesized.

- If you performed scan insertion within the synthesis tool, continue with the steps described in the section “[Creating the Post-Synthesis TSDB View \(Block-Level\)](#)” on page 365.
- If you are inserting your scan chains within Tessent, continue with the steps described in the section “[Performing Scan Chain Insertion With Tessent Scan \(Block-Level\)](#)” on page 367.

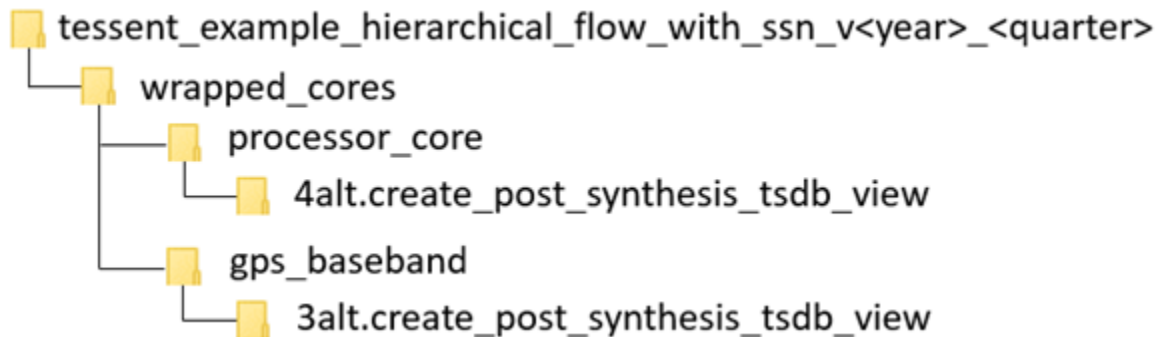
Example

Example dofiles are available as the `<module_name>.dc_synth_script` files found in the testcase directories shown in the “Referenced Testcase Step” section earlier in this topic. These files specify the TCK period and the `set_load_unload_timing_options` values using a separate file called `<design_name>.timing_options`. When you source this file at this point, it configures the SDC during synthesis. A step later in the flow also sources this file from ATPG dofiles to configure the patterns with the exact timing options that the tool used during timing closure.

Creating the Post-Synthesis TSDB View (Block-Level)

When you are using third-party scan insertion, use the following procedure to create the post-synthesis TSDB, separate from scan insertion. Creation of the post-synthesis TSDB occurs automatically as part of scan insertion when you use Tessent Scan.

You can perform this step in the following directories of the [Reference Testcase](#):



Notes About This Procedure

This step combines a third-party scan-inserted netlist and the collateral from the RTL insertion step into a new *dft_inserted_design* directory. By consolidating the netlist and the collateral into the *dft_inserted_design* directory, you can use the tool automation in future steps to load the design using the [read_design](#) command.

The TSDB consolidation flow works by using the `read_verilog` command to read the third-party scan-inserted netlist into the tool and then using the “`read_design -no_hdl -design_id rtl2`” command to read the collateral from the last RTL insertion step. Write the design into the *tsdb_outdir* directory using the “`write_design -tsdb -softlink_netlist -create_ijtag_graybox on`” command.

The following procedure and example dofile show how to create a *dft_inserted_design* directory with the post-synthesis TSDB flow.

Procedure

1. Set the context to `-no_rtl` and define the `design_id` as “gate” using the [set_context](#) command.
2. Use the [read_verilog](#) command to read the scan-inserted design. This is the output of the third-party scan-insertion step.
3. Use the “`read_design -no_hdl -design_id rtl2`” command to load the collateral from the last DFT RTL insertion step.

The design collateral includes the block level ICL, PDL, SDC, TCD, and other files.

4. Use the “`write_design -tsdb -softlink_netlist -create_ijtag_graybox_on`” command to populate the *dft_inserted_design* directory with the collateral and a symbolic link to your scan-inserted netlist. This also creates a gate-level IJTAG graybox view for the design that you can use during the ICL-based pattern verification step, and also later during SSN scan retargeting at the top level.

The tool also updates design instance names in the ICL file. This happens when the ICL objects were below generated blocks in the RTL.

Examples

The following is an example dofile for the TSDB consolidation step:

```
# Set context for this step
set_context dft -no_rtl -design_id gate

# Specify the TSDB output directory to write into.
set_tsdb_output_directory ../tsdb_outdir

# Read the scan-inserted design
read_verilog ../3.synthesize_rtl/processor_core_synthesized.vg

# Read only the collateral from the last DFT insertion step
read_design processor_core -design_identifier rtl2 -no_hdl -verbose
set_current_design processor_core

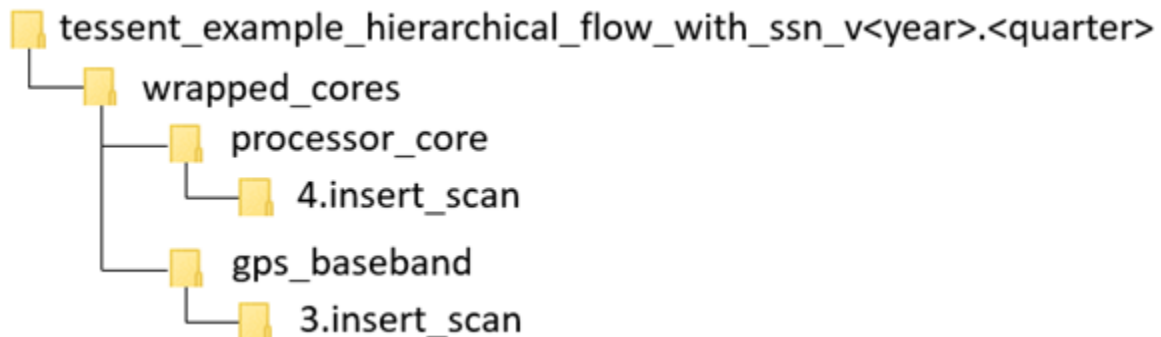
# Write the netlist and collateral to the tsdb_outdir
write_design -tsdb -softlink_netlist -create_ijtag_graybox on -verbose

exit
```

Performing Scan Chain Insertion With Tessent Scan (Block-Level)

The Tessent Scan insertion step in the SSN flow is similar to the scan chain insertion step in the Tessent Shell Flow for hierarchical designs.

You can perform this step in the following directories of the [Reference Testcase](#):



The complete details for this step appear in the topic “[Performing Scan Chain Insertion: Wrapped Core](#)” on page 154.

When SSN is present, the tool connects wrapper chains to the second, smaller EDT controller added during the step “[Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN](#)” on page 357.

The following procedure shows how to modify the dofile from the topic “[Performing Scan Chain Insertion: Wrapped Core](#)” when you are using SSN.

Procedure

1. Remove the commands that exclude the `*_edt_channels_*` from wrapper analysis. For example, remove:

```
set_wrapper_analysis_options \  
-exclude_ports [ get_ports {*_edt_channels_*}]
```

In the SSN flow, the EDT channels connect directly to the [ScanHost](#) node instead of being brought to the boundary of the block.

2. Add the following code to find each EDT and assign it to its intended scan mode:

```
foreach_in_collection edt $edt_instance {  
  if {[regexp {.*int.*} [get_single_name $edt] int_edt_instance]}{  
    puts "Found instance $int_edt_instance. \  
        Will use this for internal scan mode."  
  } elseif {[regexp {.*ext.*} [get_single_name $edt] \  
    ext_edt_instance]} {  
    puts "Found instance $ext_edt_instance. \  
        Will use this for external scan mode."  
  }  
}
```

During the second DFT insertion pass, the tool gave each added EDT controller a name according to how it was intended to be used (internal mode EDT is “c1_int” and external mode EDT is “c1_ext”).

3. Add the following commands to assign each EDT to the intended scan mode. In this example, the scan mode is named using the name of the DFT signal that activates that mode:

```
add_scan_mode int_edt_mode -edt_instances $int_edt_instance  
add_scan_mode ext_edt_mode -edt_instances $ext_edt_instance
```

If you wanted to use different scan mode names, you would need to use the `-dft_signal_name` switch of the [add_scan_mode](#) command to associate the correct DFT signal to that mode.

4. After you implement all the scan modes using the [insert_test_logic](#) command, run the code block highlighted in gold at the end of the following example dofile to perform DRC on each scan mode and extract the graybox model for the external mode.
5. Create the gate-level IJTAG graybox view by changing the context to `patterns-ijtag` and then running the “analyze_graybox -ijtag” command followed by the “write_design -tsdb -graybox” command.

Examples

The following is an example dofile for the scan insertion step:

```
# Set context to perform scan insertion and stitching  
set_context dft -scan
```



```

# Open the previous TSDB directories if not in the current working
# directory
set_tsdb_output_directory ../tsdb_outdir

# Read Tessent Library
read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib

# Read the synthesized netlist
read_verilog ../3.synthesize_rtl/processor_core_synthesized.vg

# Read the design data for the second RTL insertion pass
read_design processor_core -design_identifier rtl2 -no_hdl -verbose
set_current_design processor_core
add_black_boxes -modules { \
                    SYNC_1RW_8Kx16 \
                    }
check_design_rules
report_clocks

# To force insertion of dedicated wrapper cell use the
# set_dedicated_wrapper_cell_options command
# set_dedicated_wrapper_cell_options on -ports {.... }
# Perform and Report wrapper cell analysis
analyze_wrapper_cells
report_wrapper_cells -Verbose

# Specify different modes (int/ext) how the chains need to be stitched
# The type internal/external and enable_dft_signal are inferred from
# registered DFT Signals(int_edt_mode and ext_edt_mode)
set edt_instance [get_instances -of_icl_instances [get_icl_instances -
filter tessent_instrument_type==mentor::edt]]
foreach_in_collection edt $edt_instance {
    if {[regexp {.*int.*} [get_single_name $edt] int_edt_instance]} {
        puts "Found EDT instance $int_edt_instance. Will use for int scan_mode."
    } elseif {[regexp {.*ext.*} [get_single_name $edt] ext_edt_instance]} {
        puts "Found EDT instance $ext_edt_instance. Will use for ext scan_mode"
    }
}
add_scan_mode int_edt_mode -edt_instances $int_edt_instance
add_scan_mode ext_edt_mode -edt_instances $ext_edt_instance

# Analyze the scan chains and review the different scan modes and chains
# before stitching the chains
analyze_scan_chains
report_scan_chains

```

```
# Insert scan chains and write the scan-inserted design into tsdb_outdir
insert_test_logic
report_scan_chains
report_scan_cells > scan_cells.list

# DRC check the scan chains in each mode and create graybox for
# external mode

set_context pattern -scan

foreach_in_collection mode_wrapper [get_config_elements \
    Core(processor_core)/Scan/Mode -part tcd -silent] {
    set mode_name [get_config_value $mode_wrapper -id <0>]
    set mode_type [get_config_value type -in $mode_wrapper]
    import_scan_mode $mode_name
    #In setup mode
    report_clocks
    check_design_rules
    #In Analysis mode
    if {$mode_type eq "external"} {
        report_scan_cells
        analyze_graybox -scan
        write_design -tsdb -graybox -verbose
    }
    set_system_mode setup
}

# Create the IJTAG graybox

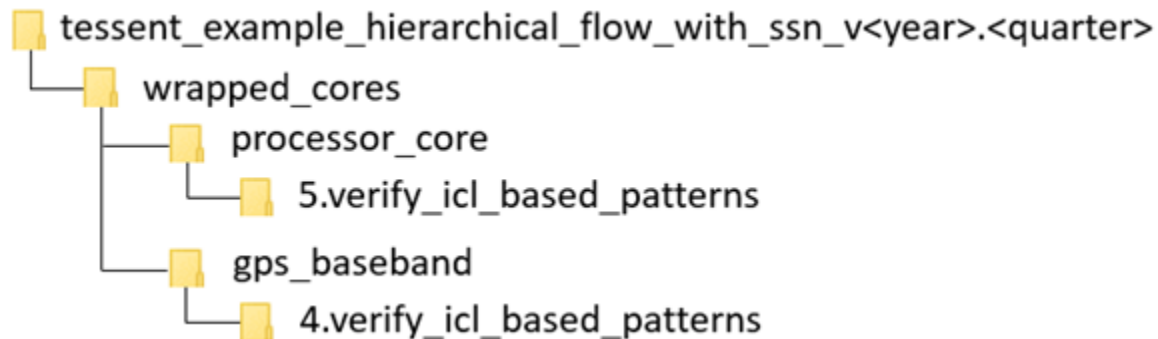
set_context patterns -ijtag
set_system_mode analysis
analyze_graybox -ijtag
write_design -tsdb -graybox -verbose

exit
```

Verifying the ICL-Based Patterns After Synthesis (Block-Level)

Resimulating the MemoryBIST and ICL verification patterns after synthesis ensures the ICL network is fully functional and able to be reliably used during ATPG and Scan retargeting setup.

You can perform this step in the following directories of the [Reference Testcase](#):



Notes About This Procedure

Verifying the ICL-based patterns after synthesis is especially critical when using SSN because the bulk of the test_setup relies on IJTAG. Your MemoryBIST simulations are also redone in this step. For the complete details of this step, see “[Verifying the ICL Model](#)” on page 157.


When you use SSN, it is important to also reverify the SSN continuity after synthesis before you try to use it for ATPG and scan retargeting.

The following procedure and example dofile show how to verify an ICL model including SSN.

Procedure

1. The top part of the dofile is identical to the normal flow. You can examine the PatternsSpecification generated by the [create_patterns_specification](#) command to see how it uses the IJTAG graybox view that you created in the previous step to simulate the ICLVerify patterns and the SSN continuity patterns.
2. As discussed earlier in this flow, you are responsible for configuring the SSN datapath you want to verify the continuity for by using iProcs containing iCall commands to the Tessent-generated pdl to write to the [Multiplexer](#) nodes when present. You must complete simulating both the ICLNetwork pattern and SSN continuity pattern before moving to the next step of the DFT flow.

Note

 You can also create the SSN continuity pattern with the low-level [create_ssn_continuity_patterns](#) command, as shown at the bottom of the example dofile found at the end of this section.

Examples

This example dofile shows how to verify the ICL model for SSN.

The [create_patterns_specification](#) command creates a pattern specification for the SSN continuity pattern and the ICL network that includes the IJTAG logic, SSN logic, and the MemoryBIST logic, if present.

This example contains no [Multiplexer](#) node to configure. If you have such nodes at the block level, see how they are handled at the bottom of the dofile example in the section “[Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN](#)” on page 385.

```
# Set context to patterns
set_context patterns

# Open the previous TSDB directories if it is not in the current

# working directory
set_tsdb_output_directory ../tsdb_outdir

# Read Tessent Library
read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib
read_verilog ../../../../library/memories/SYNC_1RW_8Kx16.v -interface_only

# Read the design files from the scan insertion pass
read_design processor_core -design_identifier gate -verbose
set_current_design processor_core

# Generate patterns to verify the inserted DFT logic
set_spec [create_patterns_specification]
report_config_data $spec
process_patterns_specification

# Point to the libraries and run the simulation
set_simulation_library_sources -v ../../../../library/standard_cells/ \
    verilog/adk.v -v ../../../../library/memories/*.v
run_testbench_simulations
check_testbench_simulations -report_status

exit
```

The following is an alternate way to create the SSN continuity pattern. Refer to [Step 2](#) in the [Procedure](#) section of this topic for an explanation of how to create the SSN continuity pattern.

```
#
# Create the continuity pattern
#
# Define SSN bus clock. If you are using time multiplexing
# you must specify the -freq_multiplier switch
add_clocks ssn_bus_clock

# check design rules and transition to analysis mode
check_design_rules

# set ijtag period
set_ijtag_retargeting_options -tck_period 10

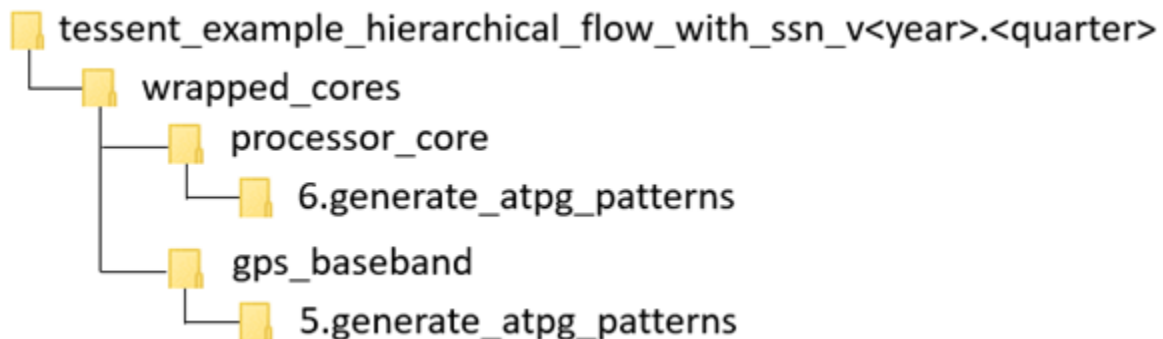
# set ssn bus clock period
set_load_unload_timing_options -usage ssn -ssn_bus_clock_period 5

# Create the SSN continuity pattern
open_pattern_set ssn_continuity -usage ssn
  create_ssn_continuity_patterns
close_pattern_set
# Write simulation testbench
write_patterns patterns/ssn_continuity.v -verilog -replace \
  -parameter_list {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1}
```

Generating Block-Level ATPG Patterns

The process for generating ATPG patterns in the SSN flow is similar to the process used without SSN.

You can perform this step in the following directories of the [Reference Testcase](#):



Notes About This Procedure

The complete details for this step appear in the “[Performing ATPG Pattern Generation: Wrapped Core](#)” on page 158. The SSN flow generates the scan graybox model as part of the “[Performing Scan Chain Insertion With Tessent Scan \(Block-Level\)](#)” step. Therefore, this step follows most of the standard flow, skipping the graybox generation because the insertion step has already completed that part. If you are performing scan insertion as part of synthesis, you create the graybox as part of creating ATPG patterns for the external mode, as shown in the

[Sample Dofile](#) in this section and illustrated in the `0opt.run_scan_graybox_generation` script in the reference testcase directories mentioned previously.

When SSN is present, the tool automatically generates all load and unload procedures and their timing based on the timing specified with the [set_load_unload_timing_options](#), [set_ijtag_retargeting_options](#), and [set_capture_timing_options](#) commands. The default bus_clock frequency and shift_clock frequency are 400 MHz and 100 MHz, respectively. Furthermore, you can improve the setup and hold timing around the edges of the scan_en and edt_updated signals. Use the `-scan_en*_extra_cycles` and `-edt_update*_extra_cycles` arguments to the [set_load_unload_timing_options](#) command to add extra cycles around the edges of scan_en and edt_update, respectively. If at any time during scan pattern retargeting or top-level ATPG you need to change the timing of a physical block, use the [set_current_physical_block](#) command to set the scope of all subsequent load/unload and capture timing settings. The reference page for the [set_load_unload_timing_options](#) command contains an explanation on how to use a single file that you source in both Tessent and the synthesis tool to configure the pattern generation and SDC consistently. The `1.run_internal_stuck` and `2.run_internal_transition` scripts in the reference testcase directories mentioned previously provide clear illustrations of this process. The `<module_name>.timing_options` file configures the SDC during synthesis, and you reuse it later to configure the patterns so that they are generated consistently.

If you did not use Tessent Scan, you cannot use the [import_scan_mode](#) command, and you use the [add_core_instances](#) command to select your active OCC and EDT controllers. You never need to use the `add_core_instances` command on the ScanHost instance. The tool automatically adds ScanHost core instances during the transition to analysis mode. When the tool performs the [check_design_rules](#) command, it automatically adds the ScanHost instance to which a scan chain or an active EDT traced. The `0opt.run_scan_graybox_generation` script in the reference testcase mentioned previously provides a clear illustration of this process.

After you run [create_patterns](#), the tool maps the SSN data stream to the SSN bus. It calculates SSH parameters when you write patterns with the [write_patterns](#) command. To see the final calculated parameters of the SSH, use the [report_core_instance_parameters](#) command after having called the `write_patterns` command.

Use the SSH loopback pattern to verify that the SSN network can successfully deliver packetized data to each active SSH. During the SSH loopback pattern, the ScanHost node does not send scan data out to the scan chains and EDT but instead loops it back internally.

- SSN patterns written in the serial test bench format deliver packetized scan data over the SSN data bus to each active SSH. Each active SSH transfers scan data to the scan chains and EDT and locally generates the scan signals. The scan out compares are mapped to the SSN bus_out.
- SSN patterns written in the parallel test bench format apply scan data similarly to a non-SSN parallel test bench. The tool adds cut points to the boundary of the SSH to drive the scan signals.

During Streaming-Through-IJTAG, the tool replaces the parallel SSN bus by the serial JTAG interface. Select the serial JTAG interface as the streaming interface with the `set_ssn_options` command. The tool delivers packetized scan data synchronously through the TDI port using TCK as the clock. You can stream any SSN pattern set through the serial JTAG interface without modification. For more information about Streaming-Through-IJTAG, refer to “[Streaming-Through-IJTAG Scan Data](#)” on page 405.

You can create SSN patterns using a scaled-down SSN bus width. Scale the SSN bus width with the `set_ssn_datapath_options` command. You can reapply any SSN pattern using a scaled-down SSN bus.

When simulating patterns at the core level, the SSN bus clock typically runs slower because it is limited by shift clock frequency. To run the bus clock at maximum frequency, set the “`SIM_SSN_MAXIMUM_BUS_SPEED 1`” parameter value pair with the “`write_patterns -parameter_list`” command and switch to add padding to the packet.

SSN supports only the `.patdb` file format for scan pattern retargeting.

Prerequisites

- Running ATPG with SSN requires a validated ICL model of the current design.

Procedure

1. Run ATPG in internal mode on the wrapped core.

This step in the SSN flow is similar to the step “Run ATPG on the internal mode of the wrapped core” in the topic “[Performing ATPG Pattern Generation: Wrapped Core](#)” on page 158, in the hierarchical flow section of this manual. Refer to this section for a detailed description of this step.

Use the following steps to update the dofile from the hierarchical flow procedure referenced previously:

- a. Set the `ijtag_tck` period for this core to the required frequency using the `set_ijtag_retargeting_options` command. This should be the `ijtag_tck` clock frequency you closed timing at.
- b. Define the SSN bus_clock and shift_clock periods for the core using the `set_load_unload_timing_options` command. The bus_clock period should be the maximum bus_clock frequency you plan to use for the chip. If you specify a slower frequency, it will limit the bus clock frequency of the entire datapath when this ATPG pattern is retargeted. The default bus_clock period is 2.5 ns, and the default shift_clock is 10 ns. The `1.run_internal_stuck` and `2.run_internal_transition` scripts in the reference testcase directories mentioned previously show you how to do this with the synthesis process used.
- c. Define the clock frequency used during the capture process. The SSH is capable of having a different frequency for the `shift_capture_clock` during the shift and the

capture cycles. The default slow capture clock frequency is 40 MHz to enable reliable capture staggering.

- d. Report core instances after you run [check_design_rules](#). As part of `check_design_rules`, the tool automatically adds the SSH core instance that was traced to from the active scan chains and EDT controllers.
- e. Write the SSH loopback pattern. Simulate this pattern to confirm that the SSN network can successfully deliver packetized data to the SSH. Start by simulating the parallel load scan patterns. Once those are clean, simulate the serial SSH loopback pattern, followed by one Serial Chain pattern and one Serial Scan pattern.

The sample dofile in the following Examples section also creates the on-chip comparator self-test pattern. This pattern set is only relevant for blocks having a ScanHost that supports the on-chip compare mode. This pattern set is not required for sign-off simulation but is critical during manufacturing test to ensure that no fault within the comparator logic can be present, which could mask faults within the scanned logic to be detected. Refer to the section “[On-Chip Compare With SSN](#)” on page 408 for more information about this pattern set.


- f. Run the “`set_chain_test -type nomask`” command. This enables writing the chain test patterns without masking. Simulating all chain test patterns is not practical for most designs. When you write the chain test patterns with the “`-end 0`” option, the tool writes one chain test pattern that simulates all the chains more efficiently.
- g. Write a single serial scan pattern. One serial scan pattern combined with a full set of parallel patterns provides sufficient coverage of the SSN and the scan chains.

Refer to the Examples section for a sample dofile updated using these steps. Also, see the *1.run_internal_stuck* and *2.run_internal_transition* scripts in the reference testcase directories mentioned previously.

2. Run ATPG in external mode on the wrapped core.

This step in the SSN flow is similar to the step “Run ATPG on the external mode of the wrapped core” in the topic “[Performing ATPG Pattern Generation: Wrapped Core](#)” on page 158, in the hierarchical flow section of this manual. During the external mode run, the core-level SSH sources the scan signals to the small wrapper chain EDT.

Note

 This ATPG pattern is used only to verify the proper behavior of the wrapper chains. It cannot be reused from the chip level.

The procedure for modifying the dofile presented in the hierarchical flow section is the same as the procedure for modifying the internal mode dofile. Refer to Substeps “a” through “g” in Step 1 for the instructions on updating your dofile. See the Examples section for a sample dofile updated using these steps.

Examples

Sample Dofile

The following dofile shows the result of following the instructions for internal and external mode ATPG. Because the dofile for the two modes are very similar, the following dofile shows the full internal mode dofile. The differences for an external mode dofile appear as comments in green text marked “EXTERNAL MODE:”.

All changes from the base hierarchical flow for SSN appear in gold text.

```
set_context patterns -scan

# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_outdir

# Read Tessent Library
read_cell_library ../../../../library/standard_cells/tessent/adk.tcelllib

# Read in the scan inserted netlist/design
read_design processor_core -design_id gate -verbose

set_current_design processor_core

# Specify the current mode using a unique name (different from
# add_scan_mode).
# Then bring in required scan mode configuration
set_current_mode int_edt_mode_stuck -type internal
import_scan_mode int_edt_mode -fast_capture off
# EXTERNAL MODE:
# set_current_mode ext_edt_mode_stuck -type external
# import_scan_mode ext_edt_mode -fast_capture off

# Define the tck period and ijtag retargeting options
set_ijtag_retargeting_options -tck_period 10

# Define the shift and bus clock timing
set_load_unload_timing_options -usage ssn \
                                -shift_clock_period 10ns \
                                -ssn_bus_clock_period 5ns

# Define the slow clock capture timing
set_capture_timing_options -usage internal \
                            -capture_clock_period 40ns

# Report core instances that were added as part of importing scan mode
report_core_instances

set_core_instance_parameters -instrument_type occ -parameter_values \
    {capture_window_size 3}

report_dft_signals
report_core_instances
report_static_dft_signal_settings

check_design_rules
```

Streaming Scan Network (SSN) Block-Level SSN Insertion and Verification

```
# EXTERNAL MODE:
# If you inserted your scan chains during synthesis, extract the graybox
# model here.
# analyze_graybox
# write_design -graybox -tsdb -verbose

report_clocks
report_core_instances

# Confirm the SSN core instance was added
report_core_instance_parameters
set_fault_type stuck

add_fault -all
report_statistics -detail

# Generate patterns
create_patterns
report_statistics -detail

# Store TCD, flat_model, fault list and patDB format files in the TSDB
# directory
write_tsdb_data -replace
```

```

# Write Verilog patterns for simulation
# Write parallel load testbench
write_patterns patterns/processor_core_int_edt_mode_stuck_parallel.v \
  -verilog -replace -parameter_list \
    {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set scan

# Write SSH loopback pattern
write_patterns patterns/processor_core_int_edt_mode_stuck_loopback.v \
  -verilog -serial -replace -parameter_list \
    {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set ssh_loopback

# Use this option for signoff simulations to verify all chains with a
# single chain pattern
set_chain_test -type nomask
write_patterns patterns/processor_core_int_edt_mode_stuck_serial_chain.v \
  -verilog -serial -replace -end 0 -parameter_list \
    {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set chain

write_patterns patterns/processor_core_int_edt_mode_stuck_serial_scan.v \
  -verilog -serial -replace -end 0 -parameter_list \
    {SIM_COMPARE_SUMMARY 1 ALL_EXCLUDE_UNUSED 0 SIM_KEEP_PATH 1} \
  -pattern_set scan

# EXTERNAL MODE:
# write_patterns patterns/processor_core_ext_edt_mode_stuck_parallel.v \
# -verilog -replace -parameter_list \
# {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set scan

# write_patterns patterns/processor_core_ext_edt_mode_stuck_loopback.v \
# -verilog -serial -replace -parameter_list \
# {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set ssn_loopback

# Use this option for signoff simulations to verify all chains with a
# single chain pattern
# set_chain_test -type nomask
# write_patterns patterns/processor_core_ext_edt_mode_stuck_serial_chain.v \
# -verilog -serial -replace -end 0 -parameter_list \
# {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set chain
# write_patterns patterns/processor_core_ext_edt_mode_stuck_serial_scan.v \
# -verilog -serial -replace -end 0 -parameter_list \
# {SIM_COMPARE_SUMMARY 1 ALL_EXCLUDE_UNUSED 0 SIM_KEEP_PATH 1} \
# -pattern_set scan

# Write SSH on-chip comparator self test patterns
write_patterns patterns/processor_core_int_ssh_occomp_self_test.v \
  -verilog -serial -replace -end 8 -parameter_list \
    {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set \
    ssh_on_chip_compare
# report fault coverage
report_statistics -detail

exit


```

Top-Level SSN Insertion and Verification

During top-level SSN insertion, the tool inserts other logic test elements into the design. This DFT insertion flow is similar to the one where SSN is not present.

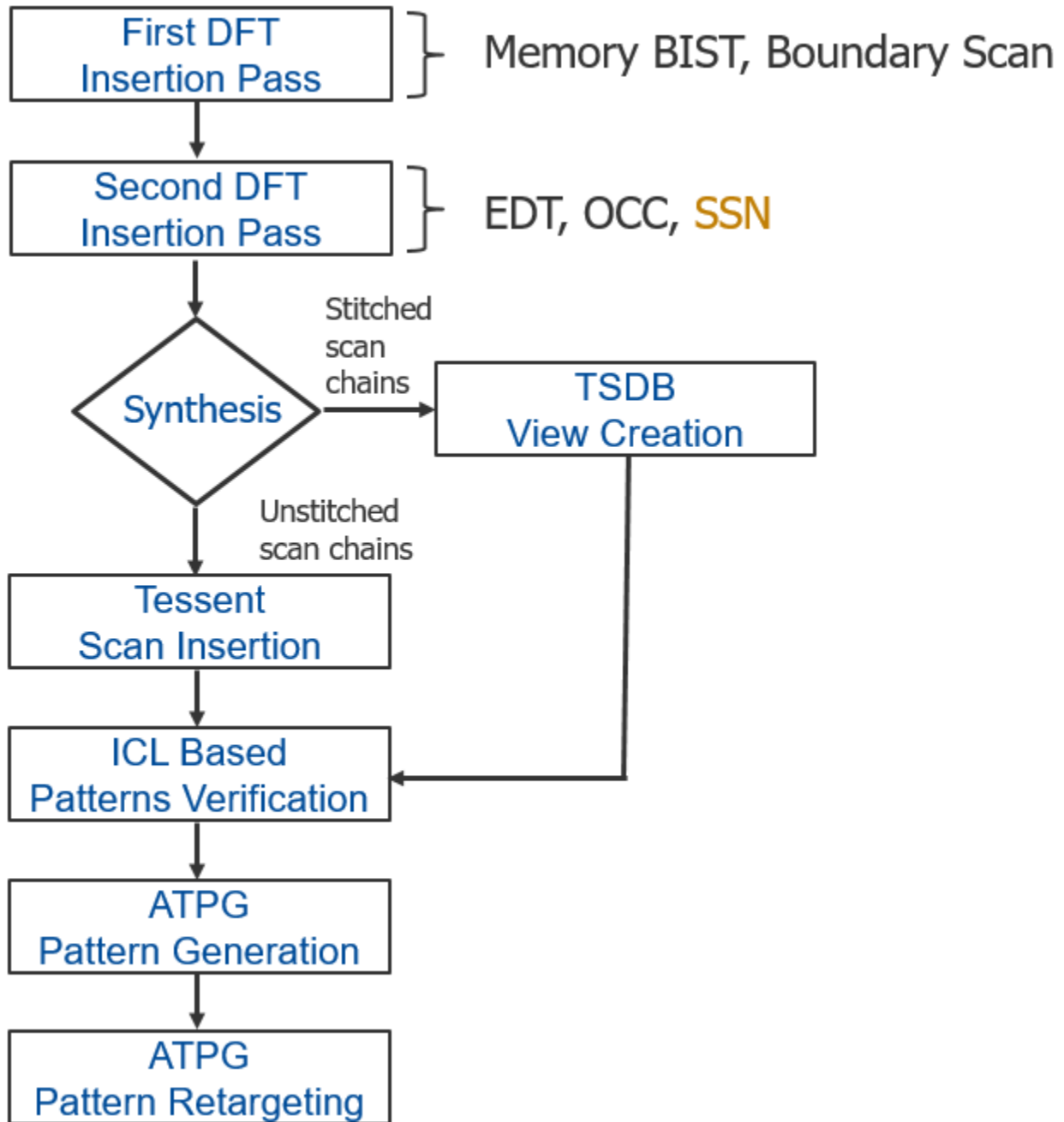
This section explains the differences in the dofiles relative to the standard hierarchical flow (described in the section “[RTL and Scan DFT Insertion Flow for the Top Chip](#)” on page 165). These differences include modifications that specific dofiles require.

Note

 This procedure assumes your design meets the prerequisites of the hierarchical flow, as described in “[DFT Architecture Guidelines for Hierarchical Designs](#)”. Additionally, each physical block must have a full OCC with a shift clock injection mux and [shift_only_mode](#).

[Figure 8-3](#) shows the same top-level workflow you follow when not using SSN. It has been updated to indicate that you insert SSN into the design during the second DFT insertion pass. This step adds a single EDT if there is top-level logic. You can also add the boundary scan chains to this top-level EDT using the [BoundaryScan/max_segment_length_for_logictest](#) property. Unlike at the core-level SSN insertion flow, this process does not add a second, smaller EDT, because the top-level boundary scan chains can provide isolation for the top level during top-level ATPG. Click the boxes in the flow diagram to see the section describing each step. The steps described in this section are implemented in the “top” directory of the [Reference Testcase](#).

Figure 8-3. SSN Top-Level Workflow



First DFT Insertion Pass: Performing Top-Level MemoryBIST 382

Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN 385

Synthesis for Top-Level Insertion 393

Creating the Post-Synthesis TSDB View (Top-Level) 394

Performing Scan Chain Insertion With Tessent Scan (Top-Level) 394

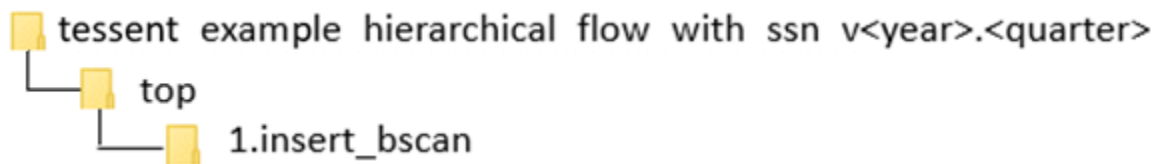
Verifying the ICL-Based Patterns After Synthesis (Top-Level) 395

Generating Top-Level ATPG Patterns	395
Retargeting ATPG Patterns	397
Performing Reverse Failure Mapping for SSN Pattern Diagnosis	400

First DFT Insertion Pass: Performing Top-Level MemoryBIST

The first DFT insertion pass at the top level adds non-scan DFT elements to the design. This step is similar to the first DFT insertion pass of the standard hierarchical flow. However, it uses the auxiliary input and output pins to connect the SSN bus and the bus_clock instead of using them for the EDT channels.

You can perform this step in the following directory of the [Reference Testcase](#):



Notes About This Procedure

- For a complete description of this step in the standard hierarchical flow, refer to the section “[First DFT Insertion Pass: Performing Top-Level MemoryBIST and Boundary Scan](#)” on page 168. To define the auxiliary logic pins for connecting the SSN bus and bus_clock, modify the dofile described in this referenced procedure.
- Identify the top-level input and output ports in your design that you plan to use for the SSN bus and bus_clock. The bus should have the same number of input and output ports (for example, 16 inputs and 16 outputs).
- The “[set_dft_specification_requirements -memory_test](#)” switch in the reference testcase is set to “On” even though this testcase has no memories in the top level. This requests that the memory clocks at the boundary of the child blocks receive design rule checks (DRCs) all the way to the top. Although issues would be detected during the [extract_icl](#) process, but it would require you to redo the DFT insertion to fix any detected issues. By performing the DRC before DFT, you can detect and correct those issues sooner.

Prerequisites

- SSN requires that the top-level design has an IEEE 1149.1 interface available during logic test.
- The lower-level physical blocks should have SSN inserted prior to inserting SSN into the top-level design. If the SSN is incomplete for any lower-level physical blocks, use the [ijtag_greybox](#) view to model the SSN datapath through the physical block during top-level SSN insertion. Alternatively, use an SSN multiplexer to force the SSN datapath around the incomplete physical block.

Procedure

Modify the `auxiliary_input_ports` and `auxiliary_output_ports` wrappers with the top-level ports you plan to use for the SSN bus and `bus_clock`:

```
# Add auxiliary mux on the inputs and outputs used for SSN bus and
# bus_clock
# bus_in {GPIO3_0 GPIO3_1} bus_out {GPIO4_0 GPIO4_1} bus_clock
{GPIO3_2}
read_config_data -in ${spec}/BoundaryScan -from_string {
  AuxiliaryInputOutputPorts {
    auxiliary_input_ports    : GPIO3_0, GPIO3_1, GPIO3_2;
    auxiliary_output_ports  : GPIO4_0, GPIO4_1;
  }
}
```

Examples

The following dofile is for the first DFT insertion pass. It contains changes to support SSN. These changes appear in gold text.

```
# Set the context to insert DFT into top-level design
set_context dft -rtl -design_id rtl1

# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_outdir

# Open the TSDB of all the child cores
open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir

# Read the tesseract cell library
read_cell_library ../../library/standard_cells/tesseract/adk.tcelllib

# Read the design
read_verilog ../rtl/noncore_blocks/pll.v -blackbox

set_design_sources -format verilog -v ../rtl/noncore_blocks/iopad_sel.v
read_verilog ../rtl/chip_top.v
read_verilog ../rtl/rds_process.v
set_current_design chip_top

set_design_level chip

# Define set_dft_specification_requirements to insert boundary scan at
# chip level
set_dft_specification_requirements -boundary_scan on -memory_Ttest on

# Specify the TAP pins using set_attribute_value
set_attribute_value TCK -name function -value tck
set_attribute_value TDI -name function -value tdi
set_attribute_value TMS -name function -value tms
set_attribute_value TRST -name function -value trst
set_attribute_value TDO -name function -value tdo
```

Streaming Scan Network (SSN) Top-Level SSN Insertion and Verification

```
# Specify all clocks so that the proper BSCAN cells gets inserted
automatically for them

add_clocks PLL_1/pll_clock_0 -reference REF_CLK -freq_multiplier 16
add_clock REF_CLK -period 48ns
add_clock INCLK -period 10ns

check_design_rules

# Create and report a DFT Specification
set_spec [create_dft_specification]

report_config_data $spec

# Segment the boundary scan to be used during logic test
set_config_value $spec/BoundaryScan/max_segment_length_for_logictest 80

# Add auxiliary mux on the inputs and outputs used for SSN bus and
# bus_clock
# bus_in {GPIO3_0 GPIO3_1} bus_out {GPIO4_0 GPIO4_1} bus_clock {GPIO3_2}
read_config_data -in ${spec}/BoundaryScan -from_string {
  AuxiliaryInputOutputPorts {
    auxiliary_input_ports : GPIO3_0, GPIO3_1, GPIO3_2;
    auxiliary_output_ports : GPIO4_0, GPIO4_1 ;
  }
}

report_config_data $spec

# Generate and insert the hardware
process_dft_specification -transcript_insertion_commands

# Extract IJTAG network and create ICL file for the design
extract_icl

# Create patterns(testbenches) to verify the inserted DFT logic
set_spec [create_patterns_specification]
process_patterns_specification

# Point to the libraries and run simulation
set_simulation_library_sources -v ../../library/standard_cells/verilog/*.v \
                               -v ../rtl/noncore_blocks/pll.v \
                               -v ../rtl/noncore_blocks/iopad_sel.v \
                               -v ../../library/memories/SYNC_1RW_8Kx16.v

run_testbench_simulations

exit
```


Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN

In the second DFT insertion pass at the top level, you insert any EDT, OCC, and SSN elements that exist at the top level. In addition, you use the `DftSpecification` to specify the physical order of lower-level cores from the SSN bus_in to SSN bus_out.

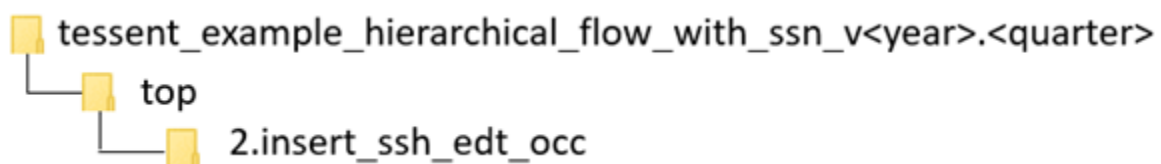
Specify this physical order using the following elements. SSN nodes that appear at the top of the `DftSpecification` are closest to SSN data_out.

- [DesignInstance](#) wrapper
- Any of the following SSN nodes along the datapath:
 - [Pipeline](#)
 - [Receiver1xPipeline](#)
 - [OutputPipeline](#)
 - [Multiplexer](#)
 - [BusFrequencyDivider](#)
 - [BusFrequencyMultiplier](#)

This section explains the slight modifications that you do to the standard flow when you insert the SSN, EDT, and OCC at the top level. For a complete description of this step in the standard flow, refer to “[Second DFT Insertion Pass: Inserting Block-Level EDT and OCC](#)” on page 149.

Reference Testcase Step

You can perform this step in the following directory of the [Reference Testcase](#):




Notes About This Procedure

- Similar to the logic test insertion passes at the block level, the tool creates the SSN hardware based on the SSN wrapper of the `DftSpecification`. For a description of the SSN wrapper, see the “[SSN](#)” wrapper description in the *Tessent Shell Reference Manual*.
- When the tool creates the SSH, EDT, and OCC elements at the same time with a single `process_dft_specification` command, the tool automates the connections between the SSH, EDT, and OCC instances.

- You must complete an ICL-based verification step at the end of the second DFT insertion pass in order to verify the DFT elements you just added to the IJTAG network. The ICL-based patterns are created as part of running the [create_patterns_specification](#) and [process_patterns_specification](#) commands, and you simulate them using the [run_testbench_simulations](#) command. The following are the ICL-based pattern verifications:
 - **ICLNetwork Pattern** — Verifies IJTAG access to all new DFT elements added to the IJTAG network and to the child blocks.
 - **SSN Continuity Pattern** — Verifies you have correctly integrated the SSN datapath into your design and detects potential problems along the datapath. If your datapath includes SSN multiplexers, you must configure the multiplexer select to test each leg of the multiplexer, as demonstrated in step 5 and at the bottom of the *l.run_ssh_edt_occ_insertion* script in the directory of the reference testcase mentioned previously.

Note

 You can also create the SSN continuity pattern with the low-level [create_ssn_continuity_patterns](#) command, as shown at the bottom of the example dofile found at the end of this section.

- To use a third-party OCC with SSN, refer to the topic “[Third-Party OCCs With SSN](#)” on page 484 in the “[Tessent SSN Examples and Solutions](#)” section.

To insert the logic test elements with SSN, use the following procedure to modify the dofile described in “[Second DFT Insertion Pass: Inserting Block-Level EDT and OCC](#)” on page 149.

Prerequisites

- SSN requires that each physical block have a full [OCC](#) with a clock injection multiplexer and support of the [shift_only_mode](#) feature.
- It is recommended that you have terminated the wrapper chains of the child blocks on a local small EDT and that EDT is driven by the ScanHost node local to the block, as it was done in the flow description found in the section “[Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN](#)” on page 357.

Procedure

1. Remove the [add_dft_signals](#) commands for the scan and retargeting signals `edt_clock`, `edt_update`, `shift_capture_clock`, `test_clock`, `retargeting1_mode`, `retargeting2_mode`, `retargeting3_mode`, and `retargeting4_mode`.

The scan signals (`edt_clock`, `edt_update`, `shift_capture_clock`, and `test_clock`) are not needed because they are sourced by the SSH located in the top level. The scan signals to the lower-level EDT and wrapper chain are sourced by the SSH inside the lower-level child core. When the lower-level child cores are in external test mode, the internal mode EDT is inactive.

The retargeting signals (retargeting<n>_mode) are not needed because each core-level EDT is accessible through the [ScanHost](#) node, effectively decoupling the dependency between core-level EDT channels and top-level I/O. With SSN, you can retarget each core to the top level individually or concurrently with any combination of the other cores.

To remove these commands, delete these lines:

```
add_dft_signals test_clock edt_update -source_nodes \
    {TEST_CLOCK_top EDT_UPDATE_top}

add_dft_signals shift_capture_clock edt_clock \
    -create_from_other_signals

add_dft_signals retargeting1_mode retargeting2_mode \
    retargeting3_mode retargeting4_mode
```

2. Add the `ssn_en` DFT signal:

```
# DFT Signal used for logic test
add_dft_signals ltest_en ssn_en
```

3. Remove all of the [add_dft_modal_connections](#) commands that add multiplexer logic to support scan pattern retargeting.
4. Add the SSN wrapper to the `DftSpecification`, as shown later in this step. The Datapath connections point to the top-level ports that the previous step equipped with auxiliary input and output logic during Boundary Scan. The tool automatically maps the specified port connections to the corresponding auxiliary data pins and automatically connects the auxiliary en pins to the `ssn_en` DFT signal.

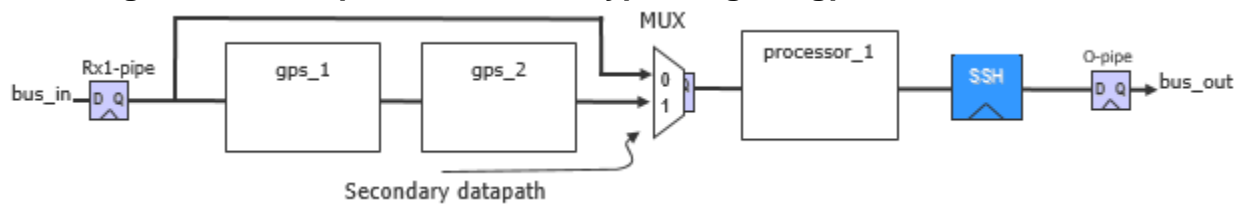
A Multiplexer node wraps the two `DesignInstance` wrappers associated with the GPS instances. You may need to create a similar setup in your design if the datapath through the those two cores can be powered down. One way to avoid those multiplexers is to make sure the SSN datapath remains in the always on domain. In this example, it is used to illustrate the way you create an `iProc` to setup the datapath so that you can use it during SSN continuity and Scan retargeting.

```
read_config_data -in_wrapper $spec -from_string {
  SSN {
    ijttag_host_interface : Sib(ssn);
    Datapath(1) {
      output_bus_width : 2;
      Connections {
        bus_clock_in : GPIO3_2;
        bus_data_in : GPIO3_%d;
        bus_data_out : GPIO4_%d;
      }
      OutputPipeline(1) {
      }
      ScanHost(1) {
      }
      DesignInstance(PROCESSOR_1) {
      }
      Multiplexer(gps_byp) {
        DesignInstance(GPS_2) {
        }
        DesignInstance(GPS_1) {
        }
      }
      Receiver1xPipeline(1) {
      }
    }
  }
}
```

5. Create the SSN continuity pattern to test the primary and secondary paths through the SSN multiplexer.
 - a. Identify the different paths from SSN bus_in to bus_out through each leg of the SSN multiplexers. In the example in substep b, a single multiplexer is in the datapath. Some complex datapaths may contain multiple SSN multiplexers.
 - b. Create an **iProc** to program the secondary datapath through the multiplexer. Reuse the iProc during ATPG pattern generation to ensure that the datapath during ATPG has been checked. Some complex datapaths may have multiple iProc procedures to program the different multiplexer combinations.

```
#
# PDL for configuring top-level mux.
# When mux select=1'b1, gps_baseband physical blocks are included
#
iProcsForModule chip_top
iProc include_gps_blocks_in_ssn_datapath {} {
  iCall chip_top_rtl2_tessent_ssn_mux_gps_byp_inst.setup \
    select_secondary_bus 0b1
}
```

Figure 8-4. Multiplexer Node for Bypassing the gps_baseband Blocks



- c. For each configuration of the multiplexers, use the ProcedureStep wrapper to create the continuity pattern.

```
# Add path through gps_baseband physical blocks to continuity pattern.
read_config_data -last -in wrapper $spec/Patterns(SSN) -from_string {
  ProcedureStep(cfg_datapath) {
    iCall(include_gps_blocks_in_ssn_datapath) {
    }
  }
  SSNContinuityVerify(include_gps_baseband) {
  }
}
```

6. Add the “-create_ijtag_graybox on” switch to the extract_icl command. The ICL-based patterns verification step also makes use of the IJTAG graybox views of the top and child levels to optimize the simulations.

Examples

The following is a dofile for the second DFT pass.

```
# Set the context to insert DFT
set_context dft -rtl -design_id rtl2

# Use dft_cell_selection that is part of the library
read_cell_library ../../library/standard_cells/tessent/adk.tcelllib

# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_outdir

# Open the TSDB of all the child cores
open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir

# Read the tessent cell library
read_cell_library ../../library/standard_cells/tessent/adk.tcelllib

# Read the verilog
read_verilog ../rtl/noncore_blocks/pll.v -blackbox
set_design_sources -format verilog -v ../rtl/noncore_blocks/iopad_sel.v

# Read the synthesized netlist
read_design chip_top -design_id rtl1 -verbose
read_design processor_core -design_id gate -view graybox -verbose
read_design gps_baseband -design_id gate -view graybox -verbose

set_current_design chip_top
```

Streaming Scan Network (SSN) Top-Level SSN Insertion and Verification

```
# The design level is already specified in the first pass; no need to specify it
# again

# Add DFT Signals
# DFT Signal used for BoundaryScan to be used with logic test without
contacting inputs
add_dft_signals int_ltest_en output_pad_disable
# DFT Signals used for Scan Tested Network with Instruments like
# memorybist/boundary scan
add_dft_signals tck_occ_en
# DFT Signal used for logic test
add_dft_signals ltest_en
# DFT Signal used by top-level EDT
add_dft_signals edt_mode ssn_en

# Specify pre-DFT DRC rules
set_dft_specification_requirements -logic_test On

check_design_rules
report_dft_control_points

# Create and report a DFT Specification
set_spec [create_dft_specification -sri_sib_list {occ edt ssn} ]
report_config_data $spec
# Use report_config_syntax DftSpecification/edt|occ to see full syntax

# The edt_clock and edt_update signals are automatically connected to EDT
instances
# The EDT controller is built with Bypass
# The shift_capture_clock is automatically connected to OCC instances
read_config_data -in_wrapper $spec -from_string {
  SSN {
    ijtag_host_interface : Sib(ssn);
    DataPath(1) {
      output_bus_width :2;
      Connections {
        bus_clock_in : GPIO3_2;
        bus_data_in  : GPIO3_%d;
        bus_data_out : GPIO4_%d;
      }
      OutputPipeline(1) {
      }
      ScanHost(1) {
      }
      DesignInstance(PROCESSOR_1) {
      }
      Multiplexer(gps_byp) {
        DesignInstance(GPS_2) {
        }
        DesignInstance(GPS_1) {
        }
      }
      Receiver1xPipeline(1) {
      }
    }
  }
}
```

```

Occ {
  ijtag_host_interface : Sib(occ);
  include_clocks_in_icl_model : yes;
  Controller(pll_clock_0) {
    clock_intercept_node      : PLL_1/pll_clock_0;
  }
  Controller(INCLK) {
    clock_intercept_node      : INCLK;
  }
}

EDT {
  ijtag_host_interface : Sib(edt);
  Controller (c1) {
    longest_chain_range : 60, 100;
    scan_chain_count : 17;
    input_channel_count : 2;
    output_channel_count : 2;
    Connections {
      EdtChannelsIn(1) {
      }
      EdtChannelsOut(1) {
      }
    }
  }
}

}

# Generate and insert the hardware
process_dft_specification

# Note the effective data and clock rate comments above each SSN node in the
# following command
report_config_data $spec

# Extract IJTAG network and create ICL file for the design
extract_icl -create_ijtag_graybox on

# Creates a synthesis script of ALL the RTL (original and newly-created)
# to be used in your synthesis tool in next Step
write_design_import_script -use_relative_path_to . for_dc_synthesis.tcl -replace

# Generate patterns to verify the inserted DFT logic
set_defaults_value /PatternsSpecification/SignoffOptions/
simulate_instruments_in_lower_physical_instances on

```

Streaming Scan Network (SSN) Top-Level SSN Insertion and Verification

```
# Generate patterns. Use the variable to update it if needed.
set spec [create_patterns_specification]
# Add path through gps_baseband physical blocks to continuity pattern.
read_config_data -last -in_wrapper $spec/Patterns(SSN) -from_string {
  ProcedureStep(cfg_datapath) {
    iCall(include_gps_blocks_in_ssn_datapath) {
    }
  }
  SSNContinuityVerify(include_gps_baseband) {
  }
}

# Validate pattern specification
process_patterns_specification

# Point to the libraries and run the simulation
set_simulation_library_sources -v ../../library/standard_cells/verilog/*.v \
                               -v ../rtl/noncore_blocks/pll.v \
                               -v ../rtl/noncore_blocks/iopad_sel.v \
                               -v ../../library/memories/SYNC_1RW_8Kx16.v

run_testbench_simulations -simulation_macro_definitions
TESSENT_DISABLE_CLOCK_MONITOR
check_testbench_simulations

exit
```

The following is an alternate way to create the SSN continuity pattern. Refer to Step 5 in the Procedure section of this topic for an explanation of how to create the SSN continuity pattern.

```
#
# Create the continuity pattern
#

# Define SSN bus clock. If you are using time multiplexing
# you must specify the -freq_multiplier switch.
add_clocks GPIO3_2

# Check design rules and transition to analysis mode
check_design_rules

# set ijtag period
set_ijtag_retargeting_options -tck_period 10

# set ssn bus clock period.
set_load_unload_timing_options -usage ssn -ssn_bus_clock_period 5

# chip level iProc to select secondary datapath through mux
source ../ssn_datapath_configuration.pdl
```



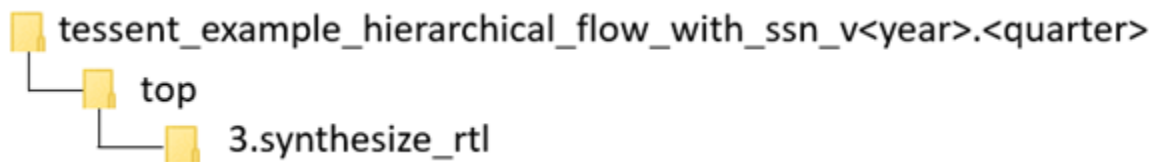
```
# Create the SSN continuity pattern
open_pattern_set ssn_continuity -usage ssn
# This is the default datapath configuration
create_ssn_continuity_patterns
# Select secondary datapath side of MUX
iCall include_gps_blocks_in_ssn_datapath
create_ssn_continuity_patterns
close_pattern_set

# Write simulation and STIL patterns
write_patterns patterns/ssn_continuity.v -verilog -replace -parameter_list \
  {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1}
write_patterns patterns/ssn_continuity.stil -stil -replace
```

Synthesis for Top-Level Insertion

Use the following dofile changes to synthesize the DFT-inserted RTL at the top level.

You can perform this step in the following directory of the [Reference Testcase](#):



In the dofile of the section “[Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN](#)” on page 385, you used the `write_design_import_script` command to export a design load script for use in your synthesis tool.

Refer to the section “[Synthesis Guidelines for RTL Designs with Tessent Inserted DFT](#)” on page 779 for guidelines on how to perform the synthesis step. Also, see the section “[Example Scripts using Tessent Tool-Generated SDC](#)” on page 760 for example scripts specific to various synthesis tools.

When synthesis completes, it produces a file containing the concatenated netlist of the physical block you just synthesized.

- If you performed scan insertion within the synthesis tool, continue with the steps described in the section “[Creating the Post-Synthesis TSDB View \(Block-Level\)](#)” on page 365.
- If you are inserting your scan chains within Tessent, continue with the steps described in the section “[Performing Scan Chain Insertion With Tessent Scan \(Block-Level\)](#)” on page 367.

Example

Example dofiles are available as the `<module_name>.dc_synth_script` files found in the testcase directories shown in the “Referenced Testcase Step” section earlier in this topic. These

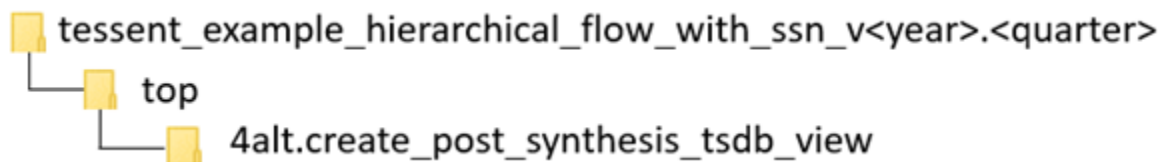
files specify the TCK period and the [set_load_unload_timing_options](#) values using a separate file called `<design_name>.timing_options`. When you source this file at this point, it configures the SDC during synthesis. A step later in the flow also sources this file from ATPG dofiles to configure the patterns with the exact timing options that the tool used during timing closure.

Creating the Post-Synthesis TSDB View (Top-Level)

When you are using third-party scan insertion, use the following procedure to create the post-synthesis TSDB, separate from scan insertion. Creation of the post-synthesis TSDB occurs automatically as part of scan insertion when you use Tessent Scan.

Reference Testcase Step

You can perform this step in the following directory of the [Reference Testcase](#):



Procedure

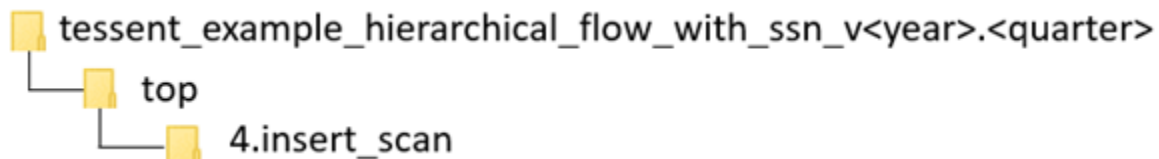
Creating the post-synthesis TSDB view at the top level is identical to the equivalent step at the block level. The only difference is that you also need to open the TSDB to the child blocks. Refer to the section “[Creating the Post-Synthesis TSDB View \(Block-Level\)](#)” on page 365 for the details. The `create_post_synthesis_tsdb_view` script in the directory shown previously in the reference testcase also illustrates this process.

Performing Scan Chain Insertion With Tessent Scan (Top-Level)

The Tessent Scan insertion step in the SSN flow is similar to the scan chain insertion step in the Tessent Shell Flow for hierarchical designs.

Reference Testcase Step

You can perform this step in the following directories of the [Reference Testcase](#):



Procedure

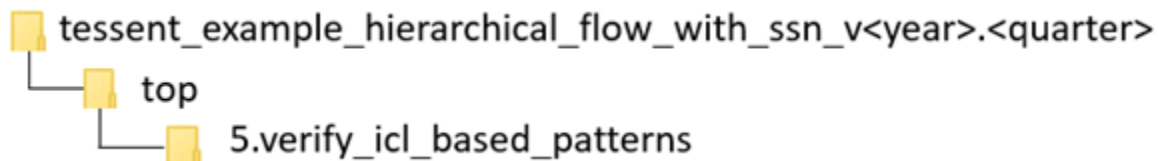
When you are using SSN, the scan insertion step at the top level is very similar to the flow you use at the block level. The only difference is that you do not need wrapper chains and, typically, have a single scan mode. The boundary scan chains implemented in the section “[First DFT Insertion Pass: Performing Top-Level MemoryBIST](#)” on page 382 were built so that the EDT controller can control them during scan test, and they can serve as isolation from the outside. Those chains were preconnected to the EDT ports and are left untouched during scan insertion. Refer to the section “[Performing Scan Chain Insertion With Tessent Scan \(Block-Level\)](#)” on page 367 for more details. It may also be helpful to look at the *run_scan_insertion* script in the directory shown previously in the reference testcase.

Verifying the ICL-Based Patterns After Synthesis (Top-Level)

Resimulating the MemoryBIST and ICL verification patterns after synthesis ensures the ICL network is fully functional and able to be reliably used during ATPG and Scan retargeting setup.

Reference Testcase Step

You can perform this step in the following directory of the [Reference Testcase](#):



Procedure

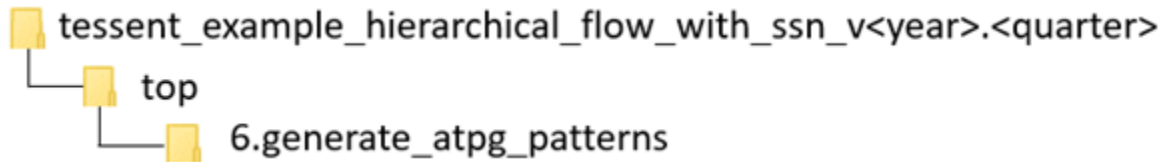
The verification of the ICL-based patterns at the top level is identical to the procedure at the block level. The only difference in the script is that you must open the TSDB for the child blocks. See the section “[Verifying the ICL-Based Patterns After Synthesis \(Block-Level\)](#)” on page 370 for the full details. It may also be helpful to review the *l.create_post_synthesis_icl_verification_patterns* script in the directory shown previously in the reference testcase.

Generating Top-Level ATPG Patterns

The process for generating ATPG patterns in the SSN flow is similar to the process used without SSN.

Reference Testcase Step

You can perform this step in the following directories of the [Reference Testcase](#):



Procedure

Generating the ATPG patterns at the top level is identical to generating the patterns at the block level. The block levels are in external mode when you create top-level ATPG patterns, so you only need to read in their scan graybox views using the “`read_design -view scan_graybox`” command.

You also need to import the `.timing_options` file used for all blocks so that you create the ATPG patterns to satisfy the maximum frequencies within all the blocks. The `1.run_tk_chip_stuck` and `2.run_tk_chip_transition` scripts in the directory shown previously for the reference testcase illustrate the loading of the timing options. Use the following Tcl code used to perform this import procedure:

```
# Import timing options used during synthesis
set tck_period 0
foreach {physical_block path} {
  processor_core ../../wrapped_cores/processor_core/3.synthesize_rtl \
  gps_baseband ../../wrapped_cores/gps_baseband/2.synthesize_rtl \
  chip_top ../3.synthesize_rtl/} {

  set_current_physical_block -module $physical_block
  source ${path}/${physical_block}.timing_options
  if {[set ${physical_block}_tck_period] > $tck_period} {
    puts "Imported tck_period from $physical_block"
    set tck_period [set ${physical_block}_tck_period]
  }
}

# Define the tck period and ijtag retargeting options
set_ijtag_retargeting_options -tck_period $tck_period
```

Refer to the section “[Generating Block-Level ATPG Patterns](#)” on page 373 for more details about the content of the dofile, or examine the `1.run_tk_chip_stuck` and `2.run_tk_chip_transition` scripts in the directory shown previously for the reference testcase. The file `3.run_sims` in same directory shows how to perform the simulation. It compiles the full view of the top level and the scan graybox view of the child blocks.

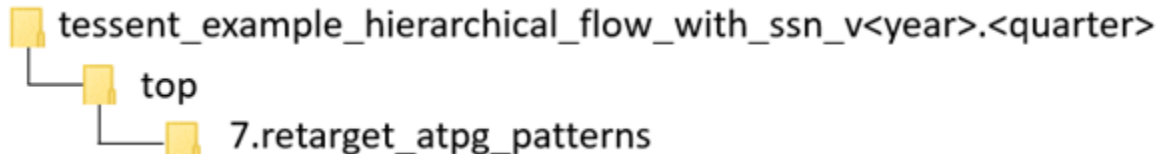
Similar to the block-level process, the parallel load scan patterns are simulated first, followed by the serial SSH loopback patterns and then the one Serial Chain and one Serial Scan patterns.

Retargeting ATPG Patterns

The steps you use to retarget the ATPG scan patterns from the block to the top level are very similar to the steps used when not using SSN. When SSN is present, the process is easier and more flexible.

Reference Testcase Step

You can perform this step in the following directories of the [Reference Testcase](#):



Notes About This Procedure

When you use SSN, the design does not have a `scan_mode` to set at the top level that configures the channel access to the required blocks. Instead, you simply use the [add_core_instances](#) command to specify which mode you want to retarget on a given set of block modules or instances. During the DFT signoff process, you typically retarget one block at a time so that you can speed up the simulation and use the IJTAG graybox views of all blocks other than the one you are retargeting. This is the usage model that the reference testcase demonstrates. The testcase retargets the stuck and transition patterns for `process_core` into a set of test benches independently from the stuck and transition patterns for the `gps_baseband` block. The `3.run_retargnet_processor_core_sims` and `6.run_retargnet_gps_baseband_sims` scripts in the reference testcase directory show how to compile the correct view for each simulation.

When you want to create the manufacturing patterns, you can retarget any number of blocks in parallel. The flow is identical to the one in the dofile in the Examples section, except that you use the `add_core_instances` command to specify all the blocks you want to include in the given retargeting pattern set. The SSN procedure has the advantage that you do not hard code the grouping of blocks at design time, so you can optimize them later, between power and test time.

The example dofile shows how you source the `.timing_options` files from each block again at this time to ensure you create the SSN patterns consistently with how timing was closed within each block.

Procedure

1. Load the design in Tessent Shell.

Regardless of which blocks you are retargeting, you always load the IJTAG graybox view of all blocks when doing scan retargeting with SSN. The three “`read_design -view ijtag_graybox`” commands used in the example dofile illustrate how to do this.

2. Import the timing options files for all blocks.

As shown in the dofile in the Examples section and in the description of the previous step, the tool automatically configures the SSN patterns to satisfy the requirements specified with the `set_load_unload_timing_options` command. You use the same command to configure the Tessent SSN SDC during timing closure. Specifying the command in a distinct file for each block enables you to share the exact same specification during timing closure and pattern generation to ensure they are always consistent.

3. Configure the SSN Multiplexer node to provide access to all active SSH blocks.

The third section of highlighted code in the example dofile is commented out because the iCall it contains configures the Multiplexer node when you want to include the `gps_baseband`. However, this dofile only retargets the patterns for the `processor_core` block. As illustrated in [Figure 8-4](#) on page 389 in the section “[Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN](#)”, an inserted [Multiplexer](#) node enables you to bypass the `gps_baseband` blocks. When you want to retarget the scan patterns of the `gps_baseband` block, you must configure the Multiplexer to include them in the active SSN datapath.

Examples

The following dofile example illustrates how to perform scan retargeting with SSN.

```
# Set the Context to retarget ATPG Patterns from lower level child cores
set_context pattern -scan_retargeting

# Point to the TSDB directory
set_tsdb_output_directory ../tsdb_outdir

# Open all the TSDB of the child cores
open_tsdb ../../wrapped_cores/processor_core/tsdb_outdir
open_tsdb ../../wrapped_cores/gps_baseband/tsdb_outdir

# Read the tessent cell library
read_cell_library ../../library/standard_cells/tessent/adk.tcelllib

# Read the hard macros
read_verilog ../../library/plls/pll.v -interface_only
read_verilog ../../library/memories/SYNC_1RW_8Kx16.v -interface_only
```

```

#For scan retargeting with SSN, only the ijtag_graybox view is needed.
read_design chip_top -design_id gate -view ijtag_graybox
read_design processor_core -design_id gate -view ijtag_graybox
read_design gps_baseband -design_id gate -view ijtag_graybox

set_current_design chip_top

# Import timing options used during synthesis
set tck_period 0
foreach {physical_block path} {processor_core ../../wrapped_cores/
processor_core/3.synthesize_rtl \
gps_baseband ../../wrapped_cores/
gps_baseband/2.synthesize_rtl \
chip_top ../../3.synthesize_rtl/} {
  set_current_physical_block -module $physical_block
  source ${path}/${physical_block}.timing_options
  if {[set ${physical_block}_tck_period] > $tck_period} {
    puts "Imported tck_period from $physical_block"
    set tck_period [set ${physical_block}_tck_period]
  }
}

# Define the tck period and ijtag retargeting options
set_ijtag_retargeting_options -tck_period $tck_period

# Define the slow clock capture timing
set_capture_timing_options -usage internal \
-capture_clock_period 40ns

# Retarget Stuck-at patterns from processor_core
set_current_mode retarget1_processor_stuck

add_core_instances -instances {PROCESSOR_1} -core processor_core -mode
int_edt_mode_stuck
import_clocks

# Configure SSN datapath to include all SSH
# This is not needed for the processor_core as it is always part of
# the active SSN datapath

# source ../ssn_datapath_configuration.pdl
# set_test_setup_icall include_gps_blocks_in_ssn_datapath -append

check_design_rules

# Write the TCD for the chip-level in the TSDB directory
write_tsdb_data -replace

# Read the patterns to be retargeted. The mode is specified with the
add_core_instances command.
read_patterns -module processor_core -fault_type stuck

```

```
# Write Verilog patterns for simulation
# Write parallel load testbench
write_patterns patterns/top_retarget_processor_core_stuck_parallel_scan.v \
  -parallel -v -replace -param_list {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} \
  -pattern_set scan

# ssh loopback pattern
write_patterns patterns/top_retarget_processor_core_stuck_ssh_loopback.v \
  -serial -v -replace -param_list {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} \
  -pattern_set ssh_loopback

# ssh on-chip comparator self test when OCComp mode present
# Not needed for Sign-off sims but critical as a manufacturing pattern
write_patterns patterns/top_retarget_gps_baseband_occomp_self_test.v \
  -serial -v -replace -param_list \
  {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set \
  ssh_on_chip_compare

# Write a single chain and single scan test pattern
write_patterns patterns/top_retarget_processor_core_stuck_serial_chain.v \
  -serial -v -replace -end 0 -param_list \
  {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set chain
write_patterns patterns/top_retarget_processor_core_stuck_serial_scan.v \
  -serial -v -replace -end 0 -param_list \
  {SIM_COMPARE_SUMMARY 1 SIM_KEEP_PATH 1} -pattern_set scan

exit
```

Performing Reverse Failure Mapping for SSN Pattern Diagnosis

SSN patterns include core-level patterns retargeted to the top level to create chip-level tester patterns. In this way, they are similar to retargeted patterns for a hierarchical DFT design. Tessent Diagnosis requires a core-level design flat model as an input. This means that, to diagnose failures, you must reverse-map the tester-reported failures at the top level to their corresponding core level, which enables you to map the reported transactions at the SSN bus level into transactions at the active SSH and EDT scan chain level. This failure mapping process must include top-level ATPG. Tessent Diagnosis then uses the mapped failure file, design flat model, and test pattern database (patdb) to perform the diagnosis.

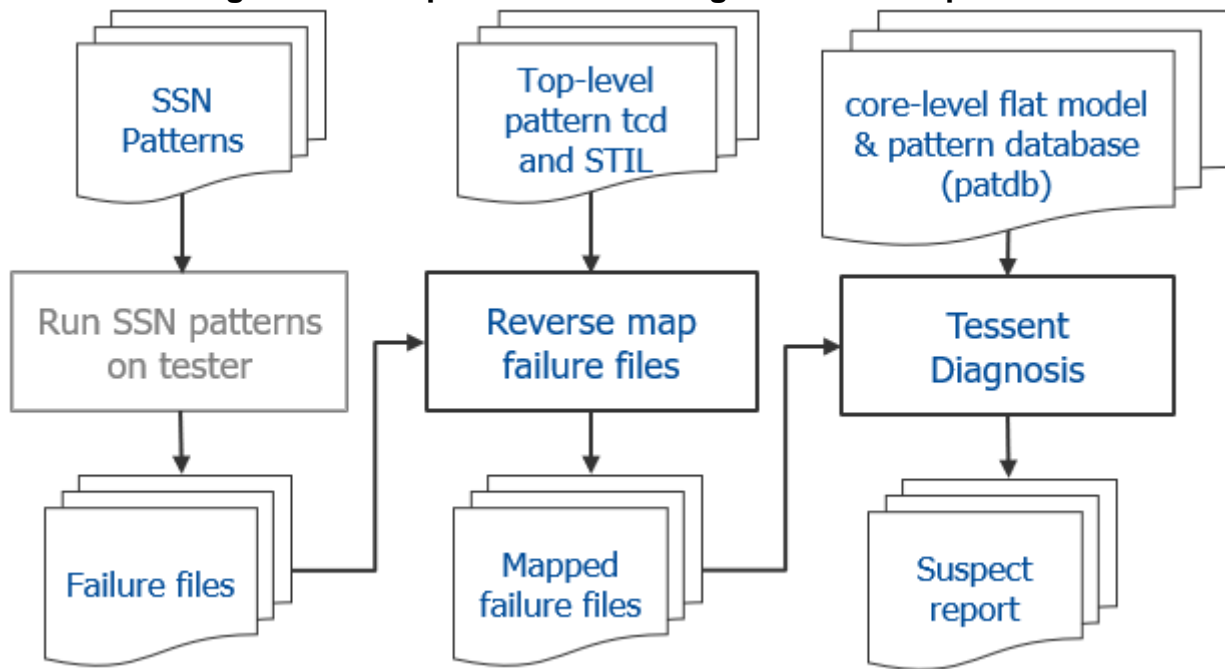
The following figure shows a simplified Tessent Diagnosis flow as a two-step flow. In the first step, you map tester-generated failures to core-level failures. Refer to the section “[Reverse Mapping Top-Level Failures to the Core](#)” in the *Tessent Diagnosis User’s Manual* for more details. In the second step, you run Tessent Diagnosis using the mapped failure files.

Note



This procedure validates that reverse mapping failures back to the core level is working properly. It does not validate diagnosis.

Figure 8-5. Simplified Tessent Diagnosis Two-Step Flow



Note

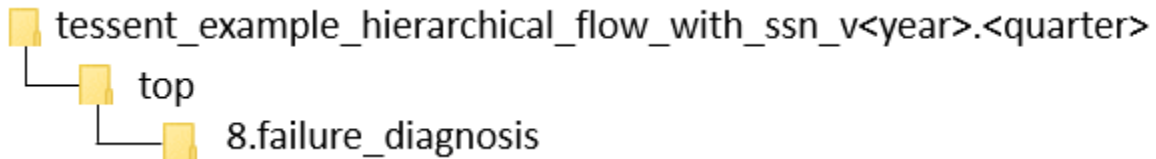
When you write SSN patterns with the maxloads option and exclude one of the pattern files from being applied to the DUT on the tester, you should also exclude that file from the reverse failure mapping process from reading that pattern file.

Tessent tools support a flow for validating the reverse failure mapping process that leads into the diagnosis flow in the preceding figure by enabling the Verilog test bench to log simulation failure information in a format similar to a tester fail file. To generate those failures, you must inject a fault on a design primitive instance (sequential or combinatorial logic). The simplest way to do this is to force a design node to a fixed value during the entire simulation, as illustrated in the testcase demonstration.

This demonstration injects a fault at one of the core scan cells during the event-driving simulation using Questa Advanced Simulator or a third-party simulator. You use the fail file generated during pattern test bench simulation to generate a core-level failure file; this does not change the diagnosis flow. Then you use the failure file, design flat model, and pattern database to perform the diagnosis and generate the list of suspects in an effort to find the root cause.

Reference Testcase Step

You can perform this step in the following directory of the [Reference Testcase](#):



Notes About This Procedure

- While the diagnosis flow requires post-layout design flat models and patterns, the testcase uses the post-synthesis flat model and patterns for the sole purpose of demonstrating the validation flow.
- This validation flow uses the Verilog test bench for serial patterns. You cannot use the parallel Verilog test bench.
- The Verilog test bench and STIL patterns should come from the same session, which means that the number of patterns written for Verilog simulation must match the STIL patterns.
- The number of Verilog test bench patterns affects the simulation results during fault injection. Choose a number that you can simulate in a reasonable amount of time while still being able to detect the injected fault. This is typically ten patterns if you inject faults on the D input of a scan flop.
- To demonstrate the flow, the testcase uses a Tessent-generated test bench of the serial scan patterns to simulate and inject a fault at a given scan cell.
- Tessent uses the provided name as a prefix. This results in the following generated filename:

```
<command-provided_filename>__<core_module_name>__<scan_test_mode>
```

Prerequisites

- When you create the serial scan test bench, you must set the parameter `SIM_DIAG_FILE` with a value of two to create the fail file during the fault-injected simulation:

```
write_patterns <test_bench_filename> -serial -parameter_list {SIM_DIAG_FILE 2}
```
- You have injected a fault at any design cell. Do this by directly forcing any cell instances pin in the design to a constant value to emulate a fault. The testcase does this as part of the simulation invocation, as shown at the end of this prerequisite. The testcase uses the following cells as a fault location:
 - **gps_baseband core** — `sw_rst_reg/D`
 - **processor_core core** — `watchdog_0.wdt_reset_reg/D`

The following example uses the Questa Advanced Simulator force command to inject a fault on the data input of a design sequential cell:

```
vsim -c -voptargs="+acc"
chip_top_top_retarget_processor_core_stuck_serial_scan_v_ctl -l
logfile/verilog.log_top_retarget_processor_core_stuck_serial_scan \
    -do " force sim:/
chip_top_top_retarget_processor_core_stuck_serial_scan_v_ctl/
chip_top_inst/PROCESSOR_1/PROCESSOR_1/watchdog_0/wdt_reset_reg/D 1 -f ; \
    if {0} {log -r /*};run -all"
```

Procedure

1. Perform reverse failure mapping:
 - a. Set the context for failure mapping.


```
set_context patterns -failure_mapping
```
 - b. Read in the retargeted pattern TCD file.


```
read_core_descriptions <pattern_tcd_file>.tcd.gz
```
 - c. Read in the retargeted STIL pattern.


```
read_patterns <pattern_stil_file>
```
 - d. Read in the fail file generated during pattern fault injection and simulation.


```
read_failures <fail_file>
```
 - e. Write out the failure file.


```
write_failures <fail_file> -replace
```
2. Perform failure diagnosis:
 - a. Set the context for scan diagnosis.


```
set_context patterns -scan_diagnosis
```
 - b. Read in the core-level design flattened model.


```
read_flat_model <flat_model_path>
```

The ATPG process for the targeted pattern creates this model. Typically, this is a compressed file with a .gz extension.
 - c. Read in the core-level pattern database.


```
read_patterns <pattern_database_path>.patdb
```
 - d. Run diagnosis on the failure file to produce a list of suspects.


```
diagnose_failures <failure_mapped_file>
```

Examples

Example 1

The following dofile is for the `gps_baseband` failure mapping. Find it in the `run_script` in the testcase directories for this step.

```
# Set the Context to pattern
set_context pattern -failure_mapping

# read retargeted ATPG tcd file
read_core_descriptions ../tsdb_outdir/logic_test_cores/ \
  chip_top_gate.logic_test_core/ \
  chip_top.atpg_retargeting_mode_retarget1_gps_baseband_stuck/ \
  chip_top_retarget1_gps_baseband_stuck.tcd.gz

# set the current design to the chip_top
set_current_design chip_top
set_system_mode analysis

# read the STIL pattern
read_patterns ../7.retarget_atpg_patterns/patterns/ \
  top_retarget_gps_baseband_stuck_serial_scan.stil

# read the fail file generated by the testbench
read_failures ../7.retarget_atpg_patterns/patterns/ \
  top_retarget_gps_baseband_stuck_serial_scan.v.fail

# write failure file for the diagnosis tool
write_failures
top_retarget_gps_baseband_stuck_serial_scan.failure_diagnosis -replace
```

Example 2

The following dofile is for the `gps_baseband` failure diagnosis. Find it in the `run_script` in the testcase directories for this step.

```
# Set the Context to pattern
set_context pattern -scan_diagnosis

# read flat model file
read_flat_model ../../wrapped_cores/gps_baseband/tsdb_outdir/ \
  logic_test_cores/gps_baseband_gate.logic_test_core/ \
  gps_baseband.atpg_mode_int_edt_mode_stuck/ \
  gps_baseband_int_edt_mode_stuck.flat.gz
read_patterns ../../wrapped_cores/gps_baseband/tsdb_outdir/ \
  logic_test_cores/gps_baseband_gate.logic_test_core/ \
  gps_baseband.atpg_mode_int_edt_mode_stuck/ \
  gps_baseband_int_edt_mode_stuck_stuck.patdb

# Diagnose failures
diagnose_failures top_retarget_gps_baseband_stuck_serial_scan. \
  failure_diagnosis___GPS_2__gps_baseband__int_edt_mode_stuck
```

Advanced Topics

The following topics provide additional information about working with SSN. This information can help you use other features and modes with SSN, configure your design for optimal use with SSN, and understand the underlying structures of SSN.

Streaming-Through-IJTAG Scan Data	405
On-Chip Compare With SSN	408
Types of Clock Networks To Use With SSN	417
Broadcast to Identical SSN Datapaths	421
Yield Statistics on ATE With SSN	422
Manufacturing Patterns With SSN	427
Manufacturing Pattern Quick Reference	428
SSN Pattern Structure	429
How To Write Complete SSN Patterns	430
How To Write <code>ssn_setup</code> and <code>ssn_end</code> Procedures to Separate Files	432
How To Write SSN On-Chip Compare Patterns	435
How To Write Streaming-Through-IJTAG Patterns	436
SSN Debug on the Tester	438
Signoff Patterns With SSN	444
Block-Level Signoff Patterns	445
Top-Level Signoff Patterns	449
Signoff Pattern Quick Reference	452
SSN SDC Constraints in the Design Flow	453
Overview of SSN-Related SDC Procs and Constraints	454
SSN/SSH SDC Constraint Descriptions	462

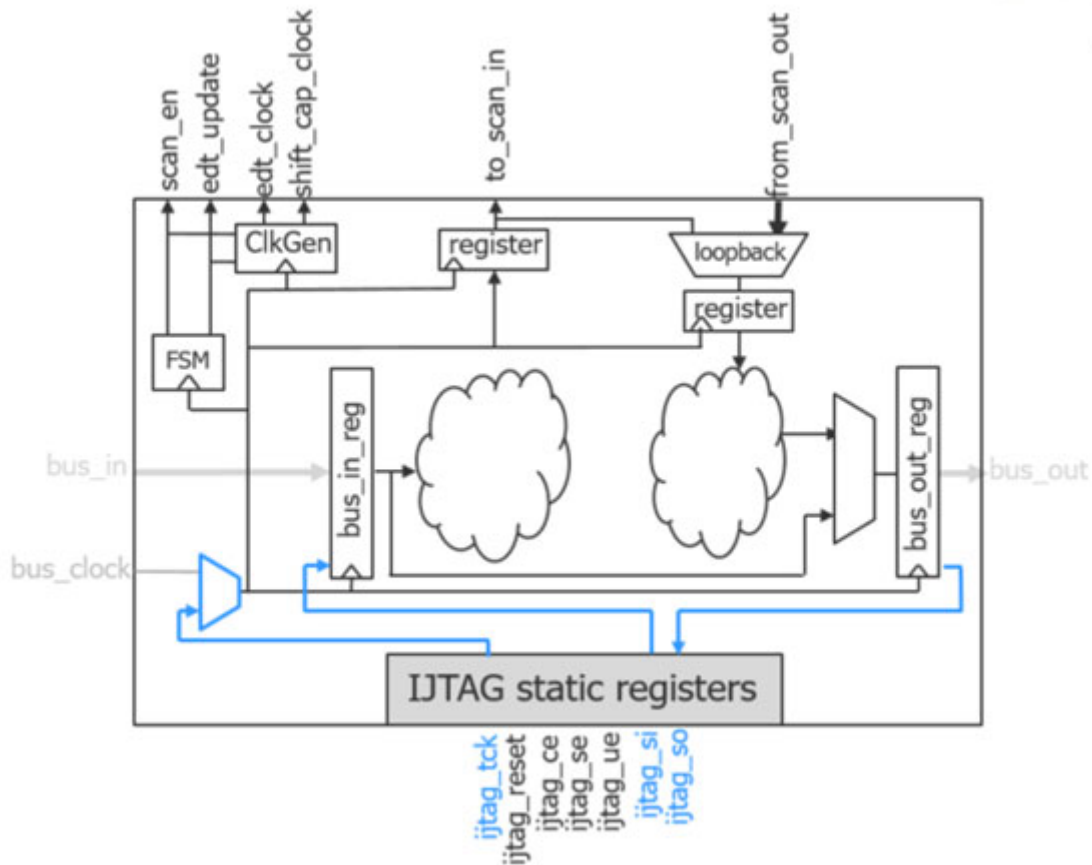
Streaming-Through-IJTAG Scan Data

The ScanHost node is designed to also enable streaming the SSN data through the IJTAG network instead of through the SSN data bus. This feature is always available with no additional overhead because the IJTAG network is already connected to each ScanHost node, and the ScanHost nodes can be programmed to use a single-bit bus.

For details on enabling this mode during ATPG or scan pattern retargeting, see the [set_ssn_options](#) command in the *Tessent Shell Reference Manual*.

When activated, the `ijtag_si` data reaching the ScanHost node is injected on bit 0 of the input data register, and bit 0 of the output data register is fed back to the `ijtag_so` pin, as the blue lines in [Figure 8-6](#) show. TCK is also injected as the SSN bus clock within the ScanHost node.

Figure 8-6. Streaming-Through-IJTAG Mode



Scan data delivery into the chip is significantly slower when using Streaming-Through-IJTAG than with the SSN data bus. It is only one bit wide and limited to the maximum operating frequency of TCK. However, the IJTAG network is very robust due to its two-edge timing and is always accessible in the system through the IEEE 1149.1 JTAG TAP.

In silicon, if you find you have a timing problem in a section of the SSN datapath, the Streaming-Through-IJTAG mode is available as an alternative method to deliver patterns to bring up your initial parts. For this reason, the Streaming-Through-IJTAG mode is also referred to as a fail-safe mechanism.

The Streaming-Through-IJTAG feature is also a more flexible replacement for LPCT-2. You are no longer limited to being able to drive only a single EDT controller having a single channel pair when using the IEEE 1149.1 protocol for delivering scan data.


If your IJTAG network architecture provides concurrent access to all ScanHost nodes you want to access through IJTAG (as is possible when using SIBs), you can test any number of blocks in parallel, with an unlimited number of channels. This can be useful in a 3D or 2.5D package if IEEE 1838 compliance is a requirement.

The IEEE 1838 standard mandates that you can perform the die interconnect test through the IEEE 1149.1 JTAG TAP, using only TCK as the scan and capture clock. To do this, create a scan mode within each die that collects the wrapper cells interacting with the pins of the die, and create ATPG patterns using this scan mode to target the interconnect faults. Apply these patterns through the normal SSN bus, which is effectively the FFP mode of the IEEE 1838 standard. Alternatively, retarget those ATPG patterns to apply through the IEEE 1149.1 TAP by using the Streaming-Through-IJTAG mode that comes with SSN, and satisfy the IEEE 1838 requirements.

Consider the following while using Streaming-Through-IJTAG:

- All ScanHost nodes you want to include must be in the active IJTAG scan path. Streaming-Through-IJTAG mode does not support IJTAG broadcast. Star network configurations restrict the number of ScanHost nodes that can be active in parallel. The IJTAG solver within the Tessent tool automatically opens up the IJTAG path if the network allows it.
- The Streaming-Through-IJTAG mode supports both internal and external capture, and all fault models are supported.

Tip

 At the top level, make the boundary scan chains available to the top-level EDT controller for best results. (See the [max_segment_length_for_logictest](#) property description in the [DftSpecification/BoundaryScan](#) wrapper reference page for more details.) Add the `int_ltest_en` DFT signal at the top level and set it to 1 using the [set_static_dft_signal_values](#) command during ATPG.


This enables full coverage of the chip logic without the need to drive the primary inputs and observe the primary outputs during ATPG.

- You can run any number of ScanHost nodes concurrently in this mode as long as your IJTAG network allows placing all the ScanHost nodes along the active IJTAG scan path.
- When a Streaming-Through-IJTAG session is complete, the binary states of all SIBs along the active scan path are restored to their states that were present before streaming began. This provides a predictable IJTAG network configuration between Streaming-Through-IJTAG sessions.

All scan testable instruments on the IJTAG network must be isolated from the following IJTAG interface signals:

- `ijtag_select`
- `ijtag_se`

Caution

-  If the state of these IJTAG interface signals is observable into scan cells, the enabling of the IJTAG streaming feature may invalidate the patterns.
-

You can accomplish this isolation by gating the signals with the `ltest_en` DFT signal. All scan tested instruments under the SIB STI are already isolated and do not require any hardware changes. Refer to [the figure “Sib\(sti\) With Scan Isolation Chain”](#) in the [Sib](#) reference page for mode details about the way scan tested instruments are isolated from the IJTAG network during scan test when you use the recommended Sib structure that is automatically inferred within the `create_dft_specification` command.

On-Chip Compare With SSN

The on-chip compare feature enables the ScanHost node to compare expected data against the response of the scan chains within the node itself.

The following subsections describe in more detail how to get the most effective use of the on-chip compare feature and the self-test patterns used to test faults with it:

- [Common Usage Scenarios](#) — Scenarios that the on-chip compare feature is designed to address.
- [On-Chip Compare Mode Operation](#) — The methods used by the on-chip compare feature and the packet format when the feature is enabled.
- [Costs, Benefits, and Recommendations](#) — Trade-offs between DFT area and test time efficiency, and recommendations for when the on-chip compare feature is most effective.
- [On-Chip Compare Mode Setup](#) — Instructions for building your ScanHost with the on-chip compare feature and enabling the feature during ATPG and scan pattern retargeting.
- [Diagnosis Using On-Chip Compare](#) — Instructions for enabling detailed diagnostics when using the on-chip compare feature.

Common Usage Scenarios

Comparing expected data to scan chain response in hardware within the ScanHost can be helpful in the following scenarios:

- You have one or more physical blocks that are instantiated multiple times within the design. With on-chip compare, the packet includes the chain input values and also the expected and mask values. The packet values are not altered by the [ScanHost](#) node because the chain input values are not replaced by the chain output values and thus can be reused by any number of instances of that physical block. The section [“Costs, Benefits, and Recommendations”](#) on page 409 describes the number of instances of the

physical block required for the on-chip compare feature to become beneficial for test time and test data volume considerations.

- You want to quickly identify the failing cores on the ATE during manufacturing because you have a Partial Good Die methodology that tolerates a subset of identical cores failing. In such a case, the failing cores must be identified on the tester, and the test flow must take actions to decommission them while testing the rest of the chip.

The on-chip compare mode provides a sticky status bit per ScanHost node that IJTAG scans out at the end of the pattern set to directly identify the failing blocks. You can also scan out a sticky status per channel output if you require. Refer to the [OnChipCompareMode/sticky_status_resolution](#) property description in the [ScanHost](#) reference page in the *Tessent Shell Reference Manual* for information about how to request usage of this feature. Refer to the [OnChipCompareMode/present](#) property description in the same topic for information about the special annotations added to the STIL file to quickly identify the comparisons that match the sticky status bits.

On-Chip Compare Mode Operation

When you add the on-chip compare mode (OCComp mode) to the ScanHost node, it can still operate in the normal mode, in which the input time slots of the packet corresponding to the channel input values are replaced by the channel output values. Then the ATE compares them at the end of the SSN data bus when they come off the chip. The packet format in this normal mode of operation is illustrated in [the figure “SSN Packet Formats When Not Using On-Chip Compare”](#) in the ScanHost reference page in the *Tessent Shell Reference Manual*.

When you enable the on-chip compare mode of operation, the packet is modified to include the channel input values followed by the expected values, the mask values, and the optional status values. The packet format in on-chip compare mode is illustrated in [the figure “SSN Packet Formats When Using On-Chip Compare”](#) in the ScanHost reference page in the *Tessent Shell Reference Manual*. When building a ScanHost node with support for on-chip compare, you should use as few output channels as possible to minimize the time slot count used to carry the expected and mask data.

For more details about the packet format in both modes, refer to the section [“SSN Packet Formats”](#) in the ScanHost reference page.

Costs, Benefits, and Recommendations

The [Common Usage Scenarios](#) section explains why it is beneficial to include the on-chip-compare mode within your [ScanHost](#) node when you reuse the physical block that contains it multiple times in the chip. Because the OCComp mode uses packet data time slots for the channel inputs, for the expected, mask, and status values, and for the status time slot, it becomes more efficient than the normal mode of operation when you have at least three identical instances of the physical block. With fewer instances, the test time and data volume reduction are negligible and do not justify the area overhead needed to implement the OCComp mode.

The typical area of a ScanHost node that does not support the OCComp mode is significantly smaller than a typical EDT controller. When the ScanHost node supports the OCComp mode, its area becomes comparable to a typical EDT controller. To minimize the area impact of OCComp on the ScanHost node, you must minimize the number of output channels you use with the EDT. In this case, you typically use a single output channel.

You can build the ScanHost node with the support of many status groups for diagnosis. Building support for many status groups increases the area of the ScanHost node, and using many status groups increases test time and data volumes. Refer to the section “[Diagnosis Using On-Chip Compare](#)” on page 410 for more information when and how to use multiple status groups.

On-Chip Compare Mode Setup

To set up OCComp mode, you need to equip the ScanHost hardware node with the logic for it and then enable the mode once the node is set up for it. The following subsections describe how to do this.

Equipping the ScanHost Hardware Node With On-Chip Compare Mode

The ScanHost node is not built with the on-chip compare mode by default. Building with OCComp mode doubles the area of the controller. Enable the creation of the OCComp mode by inserting the [OnChipCompareMode](#) wrapper within the [ScanHost](#) wrapper, typically when you have one of the conditions described in the section “[Common Usage Scenarios](#)” on page 408. Use the [OnChipCompareMode/sticky_status_resolution](#) property within the wrapper to specify whether you want one sticky status per output channel or only a single one for the entire ScanHost node.

Enabling On-Chip Compare Mode During ATPG and Scan Pattern Retargeting

By default, using the [write_patterns](#) command in the `patterns -scan` or `patterns -scan_retarting` context does not enable OCComp mode. Manually enable OCComp mode by using the “[set_core_instance_parameters -parameter_values {on_chip_compare_enable on}](#)” command. You can either specify a group of ScanHost instances by using the `-instances` switch or use the “`-instrument_type ssh`” switch to enable on-chip compare on all ScanHost nodes that support it.

Diagnosis Using On-Chip Compare

As described in the figure “[SSN Packet Formats When Not Using On-Chip Compare](#)” and its preceding text in the [ScanHost](#) reference page in the *Tessent Shell Reference Manual*, you can program the ScanHost node to report per-cycle pass/fail information into special Status time slots when using the OCComp mode. You can choose to have no status time slots and rely purely on the Go/NoGo sticky status bit scanned out with IJTAG at the end of the session. This sticky status provides failure resolution down to the failing EDT output channel when you have specified [OnChipCompareMode/sticky_status_resolution](#) as “`output_chain`”, but it does not provide diagnostic resolution down to the failing gate because there are no per-cycle compare statuses.

When several ScanHost nodes are programmed to use the same status time slots, which is the default behavior, diagnosis may require reapplication of the pattern set to collect all the data needed to perform detailed diagnostics of every failing core. Consider a case where all identical core instances are placed into a single status group, such that their per-cycle pass/fail information aggregates into the same status time slots. If any of those bits indicate failures, you have the cumulative per-pin, per-cycle fail data but may not be able to determine which core or cores the failures came from. The sticky status bits unloaded at the end of the pattern set through JTAG indicate which cores failed at least once. If only one core in this group failed, then you know the per-cycle pass/fail data came from this core alone; therefore, you have all the information needed for diagnosis. However, if multiple cores fail, you must then separately test and observe each failing core to get its individual fail data. If two cores fail, for example, then you reapply the same test set twice, with patching applied to the `disable_on_chip_compare_contribution` bit for all but one ScanHost node at a time. There is no need to store separate patterns for diagnosis on the tester. Refer to the description of the `disable_on_chip_compare_contribution` bit in the table “JTAG Registers in ScanHost Nodes” in the ScanHost reference page in the *Tessent Shell Reference Manual* for more information about the special iWriteVar annotation that identifies the location of such patchable bits.

If identical core instances are split into multiple groups, this slightly increases the test time, but decreases the probability of resorting to multiple test applications for collecting diagnosis data. In the example shown in the figure “SSN Packet Formats When Using On-Chip Compare” in the ScanHost reference page in the *Tessent Shell Reference Manual*, the six cores are split into two groups. If you find cores A1 and A4 to have failed, there is no need for test reapplication, because cores A1 through A3 accumulate their status bits separately from cores A4 through A6. However, if cores A1 and A3 fail, you must reapply the tests with patching to acquire the individual fail data. In an extreme case, you may choose to assign each core instance to its own group to observe each core individually. This mode of operation may be better suited for silicon debug than high-volume manufacturing.

To learn about the ATE failure log format, refer to the section “ATE Failure File Format Requirements” in the *Tessent Diagnosis User’s Manual*.

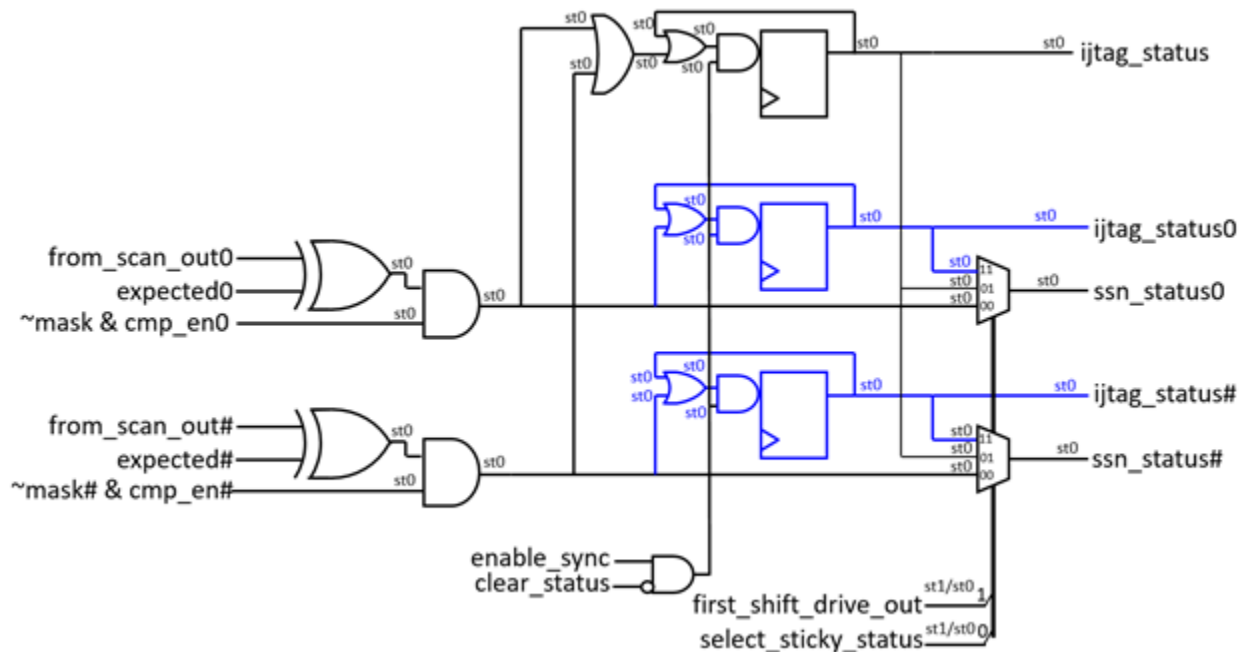
To learn about how to map the failures at the chip boundary to the core level and to enable performing detailed diagnostics, refer to the section “Reverse Mapping Top-Level Failures to the Core” in the *Tessent Diagnosis User’s Manual*. The failure mapping steps described in this section are required when using SSN, even for top-level ATPG patterns. This is unlike when you are not using SSN, where they are only required for retargeted scan patterns.

On-Chip Comparator Self-Test Patterns

As described in this section, you can equip the SSN ScanHost node with On-Chip compare logic. This logic is described in the “On-Chip Compare Logic” figure in the ScanHost reference page. There are several possible faults in the comparator logic that could prevent the detection of mismatches caused by defects in the scanned circuit. Figure 8-7 identifies those faults. The opposite fault polarity would result in systematic mismatches, and you can detect them with the normal On-Chip compare patterns. The faults listed in this figure would result in faults

within the scan circuitry that can go undetected. You can generate a special pattern using the ScanHost loopback mode with expected values set opposite to the scan-in values, to cause mismatches and enable you to observe the effects on the SSN bus outputs. The logic shown in blue in the following figure exists only when you have specified the ScanHost/OnChipCompareMode/sticky_status_resolution property as “output_chain”.

Figure 8-7. Faults in Comparator Logic That Could Mask Real Faults



Structure of the On-Chip Comparator Patterns

Generate the on-chip comparator self-test pattern set using the “[write_patterns](#) -pattern_set ssh_on_chip_compare” command in the patterns -scan or -scan_retargeting context. The tool performs the on-chip comparator self-test pattern on all active SSHs in the current atpg or scan_retargeting session. Use the [report_core_instances](#) command to see the active SSHs

The self-test pattern programs the ScanHost node with the following register values, so that each scan load comprises exactly five packets.

```

edt_update_falling_launch_word           0
edt_update_falling_transition_words      0
extra_shift_packets                       0
force_suppress_capture                   1
loop_back_en                             1
on_chip_compare_enable                   1
on_chip_compare_group                    1
on_chip_compare_group_count              1
scan_en_launch_packet                     0
scan_en_transition_packets                0
total_shift_cnt_minus_one                 1

```

With these settings, each scan load is exactly five packets long, and the packets have the following labels: `edt_update`, `shift0`, `shift1`, `post_shift0`, and `post_shift1`. The values of the `from_scan_out_bits` registers equal the number of output chains usable during on-chip compare for each chain group; refer to the [ScanHost](#) reference page for more details in the description of the `output_chain_count_in_on_chip_compare_mode` property. The values of the `to_scan_in_bits` registers match the exact value of the `from_scan_out_bits` register for each chain group. This ensures that there are exactly the same amount of scan-in values as there are comparators to test. The format of the five packets is shown in the following output for an example ScanHost having 3 comparators. The green values in the following output carry the scan payload in and out. The tool compares the `i0`, `i1`, and `i2` values provided in the `shift0` and `shift1` packets against the `e0`, `e1`, and `e2` values and masks them with the `m0`, `m1`, and `m2` values scanned in during the `post_shift0`, and `post_shift1` packets. The tool scans out the comparator statuses per output chain during the `post_shift0` and `post_shift1` packets.

packet_id	scanin			exp			mask			status		
<code>edt_update</code>	i0	i1	i2	e0	e1	e2	m0	m1	m2	s0	s1	s2
<code>shift0</code>	i0	i1	i2	e0	e1	e2	m0	m1	m2	s0	s1	s2
<code>shift1</code>	i0	i1	i2	e0	e1	e2	m0	m1	m2	s0	s1	s2
<code>post_shift0</code>	i0	i1	i2	e0	e1	e2	m0	m1	m2	s0	s1	s2
<code>post_shift1</code>	i0	i1	i2	e0	e1	e2	m0	m1	m2	s0	s1	s2

The following list describes three control bits whose time slots are identified in gold in the preceding scan load description.

- **select_sticky_status** — This control bit is scanned in during the `i0` time slot of the `edt_update` packet. When the value is “1”, the `select_sticky_status` signal, shown in [Figure 8-7](#) on page 412, goes high at the end of the `shift1` packet within the same scan load.

As that figure illustrates, the logic injects the sticky status results into the status time slots of the `post_shift0` and `post_shift1` within the same scan load. It also injects the global sticky status into the status time slots of the `shift0` packet (shown in magenta in the preceding scan load description) of the next scan load. As previously described, the `select_sticky_status` signal goes up or down at the end of the `shift1` packet, which explains why the global sticky status observed in the `shift0` packet happens in the next scan load.

- **last_scan_load** — This control bit scans in during the `i0` time slot of the `post_shift0` packet. It indicates to the ScanHost that the current scan load is the last one and to stop performing any comparisons in the next scan loads that may continue to flow through the datapath, which are to complete the testing of other ScanHosts with more comparators to test.
- **clear_sticky_status** — This control bit is scanned in during the `i0` time slot of the `post_shift1` packet. When the value is “1”, the `clear_status` signal, shown in [Figure 8-7](#) on page 412, goes high during the `shift1` packet within the next scan load. As mentioned in the preceding “`select_sticky_status`” section, the tool observes the global sticky status

during the shift0 packet, which is before the clear_sticky_status takes effect. Therefore, it observes the effect of the clear_sticky_status in the next scan load, assuming no new mismatches were injected during the post_shift0 and post_shift1 packet to cause the global sticky status to go up again after it was cleared at the end of the shift1 packet.

The on-chip comparator self-test pattern set comprises six phases, described in the following sections. Every compare in the pattern uses a unique label for identification, constructed as in the following:

```
<ssh_icl_instance>.<phase_id>.<packet_id>.scanin# for the scan in values
<ssh_icl_instance>.<phase_id>.<packet_id>.exp# for the expected values
<ssh_icl_instance>.<phase_id>.<packet_id>.mask# for the mask values
<ssh_icl_instance>.<phase_id>.<packet_id>.status# for the status values
```

The output patterns file such as STIL will have one such bit_annotation for each output time slot to help quickly identify the meaning of any mismatches. Refer to the section “[Retargeted Symbolic Variables](#)” in the *Tessent IJTAG User’s Manual* for information about the bit annotations.

The following describes the six phases of the on-chip compare self-test, with the faults from [Figure 8-7](#) on page 412 that they each cover.

Phase 1

This phase consists of a single scan load formed with one set of the five packets shown previously. This scan load uses consistent scanin and expected values for both shift cycles so that it generates no mismatches and the sticky statuses get initialized to 0. The phase_id is “phase1”.

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 0 0 0 0 0 0 0 0 0 0 L L L
shift0:     1 1 1 0 0 0 0 0 0 0 L L L
shift1:     0 0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 0 L L L
post_shift1 0 0 0 0 0 0 0 0 0 0 L L L
```

Phase 2

This phase consists of *N* scan loads used to test that the *N* XOR gates in [Figure 8-7](#) on page 412 are not stuck at 0 when the scanin value is 0 and the expected value is 1. The phase_id is “phase2_X” when testing the Xth comparator. The following is the scan load used to test comparator 0.

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 0 0 0 0 0 0 0 0 0 0 L L L
shift0:     0 0 0 0 0 0 0 0 0 0 L L L
shift1:     0 0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 0 0 0 0 0 0 H L L
post_shift1 0 0 0 0 0 0 0 0 0 0 L L L
```

Phase 3

This phase consists of N scan loads used to test that the N XOR gates in Figure 8-7 on page 412 are not stuck at 0 when the scanin value is 1 and the expected value is 0. The phase_id is “phase3_ X ” when testing the X th comparator. The following is the scan load used to test comparator 0.

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 0 0 0 0 0 0 0 0 0 L L L
shift0:     1 0 0 0 0 0 0 0 0 L L L
shift1:     0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 0 0 0 0 0 0 H L L
post_shift1 0 0 0 0 0 0 0 0 0 L L L

```

Phase 4

This phase consists of two scan loads. It tests that the sticky statuses are able to hold. As shown in Figure 8-7 on page 412, there is a global sticky status and optional sticky statuses per output chain. The ScanHost/OnChipCompareMode/sticky_status_resolution property controls the presence of the sticky statuses per output chains. The phase_id for those two scan loads are “phase4_0” and “phase4_1”.

The effect of the select_sticky_status control bit takes effect at the end of the shift1 packet within the same scan load. The effect of the clear_sticky_status happens at the end of the shift1 packet within the next scan load.

When only the global sticky status is present, the pattern looks like the following diagram. The select_sticky_status takes effect at the end of the shift1 packet. The three red Hs in the second scan load correspond to the global sticky status being observed during the status time slots of the shift0 packet. The global sticky status remembers the faults injected during Phase2 and Phase3 and clears at the end of the shift1 packet of the second scan load.

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0:     1 1 1 0 0 0 0 0 0 L L L
shift1:     0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 L L L
post_shift1 1 0 0 0 0 0 0 0 0 L L L // clear_sticky_status

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0:     1 1 1 0 0 0 0 0 0 H H H
shift1:     0 0 0 0 0 0 0 0 0 L L L // status cleared here
post_shift0 0 0 0 1 1 1 0 0 0 L L L
post_shift1 0 0 0 0 0 0 0 0 0 L L L

```

When the sticky statuses per output chain are present, the pattern looks like the following diagram. The global sticky status observation is identical to what was described previously. However, because the sticky statuses per output chain are present, the tool observes them during the first scan load. The clear_sticky_status control requested during the first scan load takes effect after the shift1 packet of the next scan load, which explains why the status values return to being L during the second scan load.

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0:     1 1 1 0 0 0 0 0 0 L L L
shift1:     0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 H H H
post_shift1 1 0 0 0 0 0 0 0 0 H H H // clear_sticky_status

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 L L L
shift0:     1 1 1 0 0 0 0 0 0 H H H // global sticky status
shift1:     0 0 0 0 0 0 0 0 0 L L L // status cleared here
post_shift0 0 0 0 1 1 1 0 0 0 L L L
post_shift1 0 0 0 0 0 0 0 0 0 L L L

```

Phase 5

This phase consists of N scan loads. It tests that all inputs of the wide OR gate feeding the global sticky status register are not stuck at 0. The tool generates faults for each comparator in its respective scan load by setting the shift0 value to 0 and expecting a H. It observes the global sticky status in the status bits of the Shift0 packet of the next scan load. The phase_id for those two scan loads is “phase5_x”.

The following diagram shows the first two scan loads with the sticky status per chain output feature enabled. The failure appears during the post_shift1 packet, even though the tool detected the fault in the post_shift0 packet because of the presence of the blue flip-flops in [Figure 8-7](#) on page 412:

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0      0 1 1 0 0 0 0 0 0 L L L // global sticky status
shift1      0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 L L L
post_shift1 1 0 0 0 0 0 0 0 0 H L L // clear_sticky_status

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0      1 0 1 0 0 0 0 0 0 H H H // global sticky status
shift1      0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 L L L
post_shift1 1 0 0 0 0 0 0 0 0 L H L // clear_sticky_status

```


The following diagram shows the first two scan loads with the sticky status per chain output feature turned off. The failure appears during the post_shift0 packet because of the absence of the blue flip-flops in [Figure 8-7](#):

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 1 0 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0      0 1 1 0 0 0 0 0 0 0 L L L // global sticky status
shift1      0 0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 H L L
post_shift1 1 0 0 0 0 0 0 0 0 L L L // clear_sticky_status

edt_update: 1 0 0 0 0 0 0 0 0 L L L // select_sticky_status
shift0      1 0 1 0 0 0 0 0 0 H H H // global sticky status
shift1      0 0 0 0 0 0 0 0 0 L L L
post_shift0 0 0 0 1 1 1 0 0 0 L H L
post_shift1 1 0 0 0 0 0 0 0 0 L L L // clear_sticky_status

```

Phase 6

This phase consists of one scan load. It observes the last global sticky status result associated to the last comparator that was triggered during the last scan load of Phase 5. During this phase, the tool also fault-injects all comparators to set all sticky statuses per comparator and observes them with an IJTAG scan load as a last step when the feature is present. The phase_id for this scan load is “phase6”.

```

          i0 i1 i2 e0 e1 e2 m0 m1 m2 s0 s1 s2
edt_update: 0 0 0 0 0 0 0 0 0 L L L
shift0      0 0 0 0 0 0 0 0 0 H H H
shift1      0 0 0 0 0 0 0 0 0 L L L
post_shift0 1 0 0 1 1 1 0 0 0 H H H // last scan load
post_shift1 0 0 0 0 0 0 0 0 0 L L L

```

Types of Clock Networks To Use With SSN

The configuration of the clock network you use with SSN depends on the logic in your design. Frequency multipliers and dividers, along with other nodes, can help you optimize your clock network and avoid large a cumulative delay value.

The streaming scan network is a synchronous bus going from a set of inputs through several stages of pipelining across many physical regions until it comes back out on a set of outputs. The pipeline stages occur within the various node types along the datapath.

The [SSN DftSpecification](#) offers several node types to help with crossing clock domain boundaries. The following sections explain various ways in which you can implement clock trees to best meet your requirements.

Hierarchical Clock Tree

The most common and straightforward clock scheme is to build a hierarchical clock tree for the entire SSN datapath or for the SSN datapath within groups of physical blocks. Implementation of one or more hierarchical clock trees requires space in the upper physical regions to place the clock buffers around the child physical blocks. You may not want to or be able to build a hierarchical clock tree that is common for all child physical blocks. Instead, you can create more than one smaller Clock Tree Synthesis (CTS) regions. If you do this, you must use a pair of [BusFrequencyDivider](#) and [BusFrequencyMultiplier](#) nodes to reliably transfer the data from one clock domain to the next.

- To see how to insert the clock domain transfer node within the top physical regions, refer to [Example 1 of the BusFrequencyDivider](#) topic.
- To see how to insert the clock domain transfer node such that the source and destination clock domains remains localized to each physical block, refer to [Example 2 of the BusFrequencyDivider](#) topic.
- To see how to insert the clock domain transfer node inside the receiving physical block to keep the number of pins crossing the physical block boundary to a minimum, refer to [Example 3 of the BusFrequencyDivider](#) topic. When you do this, you need an extra miniature clock tree within the receiving physical block for the [BusFrequencyDivider](#) node and must use a source synchronous timing interface between the sourcing and receiving physical block. Refer to the next section for an explanation of such timing interfaces.

Refer to the [Using the BusFrequencyDivider to Cross CTS Regions](#) section of the [BusFrequencyDivider](#) topic in the *Tessent Shell Reference Manual* for an explanation on how the [BusFrequencyDivider](#) and [BusFrequencyMultiplier](#) nodes work together to provide an arbitrarily large tolerance of skew between the transmitting and receiving clock domains by increasing the `frequency_ratio` value.

Source Synchronous Timing Interface

A source synchronous timing interface, also called a forward timing interface, relies on the fact that the clock follows the data. The physical block sourcing the data to the receiving node also sources the clock input at the same time. The delay of the datapath closely matches the delay of the clock path. The data is launched on one edge of the clock at the source and is captured on the opposite clock edge at the destination. As long as the delay on the datapath matches the delay of the clock path within half a clock period, the transfer of data is synchronous and reliable.

[Example 3 of the BusFrequencyDivider](#) topic uses this technique. The transmitting physical block ends with a [Pipeline](#) node, which launches the data on the positive edge of the clock. The receiving [BusFrequencyDivider](#) is built with the `input_retimed` property set to “on” so that it is equipped with a negative edge retiming flip-flop stage on its input data bus. The retiming stage ensures that the receiving node strobbs the data on the opposite edge of the clock relative to the launch edge of the transmitter.

The forward timing technique is used in [Example 3 of the BusFrequencyDivider](#) topic to hand off data from one physical block to the next without requiring the balance clock tree of one block to enter the other. It also uses the narrow data bus when crossing the physical block boundaries. It is important to use this technique combined with a BusFrequencyDivider and BusFrequencyMultiplier node pair so that you can return to using a hierarchical clock tree for the other section of the datapath. Although it is possible to use a source synchronous interface between each physical block, due to its easy implementation, this is not a recommended practice in a large design, because this would accumulate a CTS insertion delay in each physical block. The data output of the last physical block would become so delayed relative to the clock input coming from the tester that it would be impossible to reliably sample the output data on the ATE and find a strobe point that would work for best- and worst-case conditions.

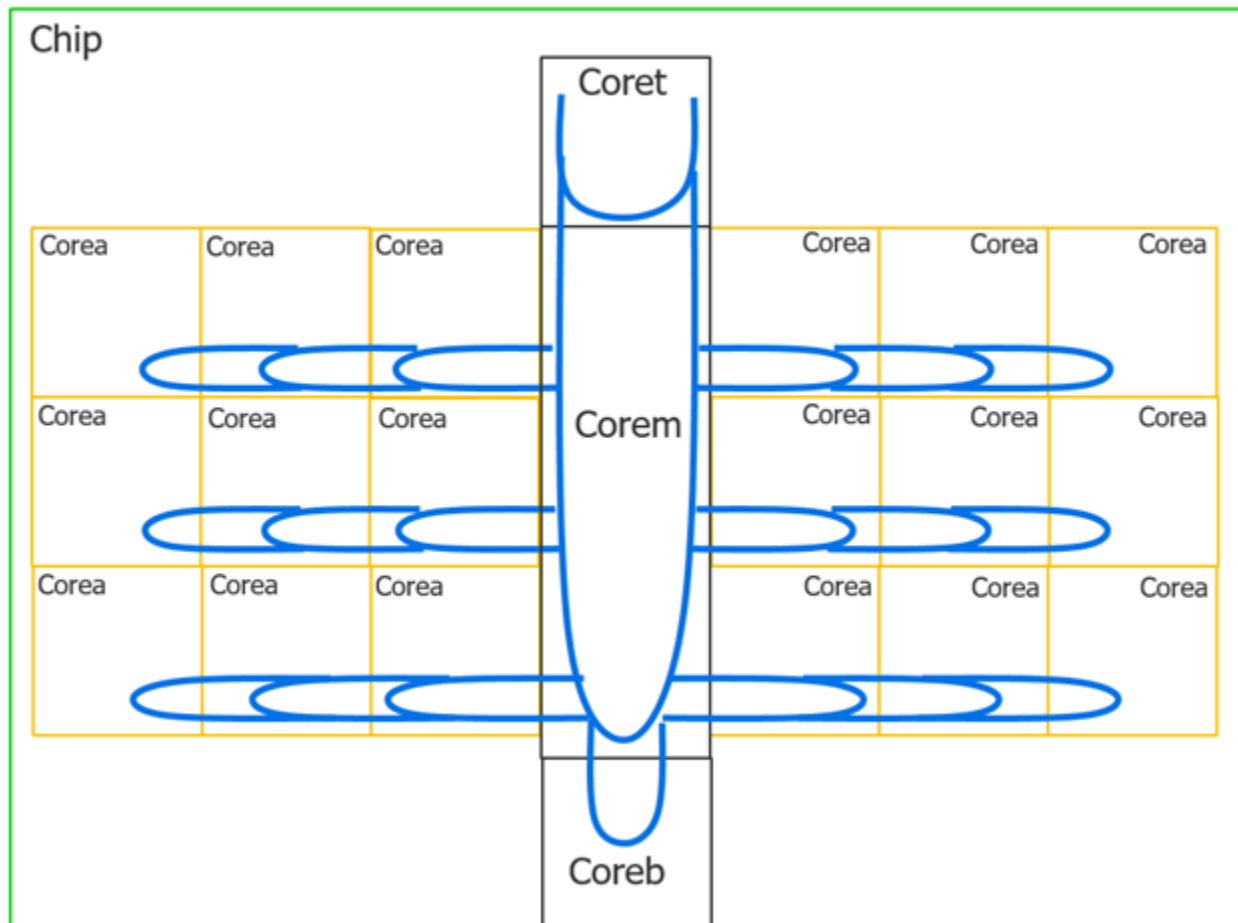
The following sections explain how to use forward design timing between each node when the datapath goes and comes back between each pair of physical blocks.

Combining Forward and Loop Timing for Tiling Layout Flows

When you are implementing a pure tiling layout flow, you do not have any logic between the blocks. Everything is self-contained within the child physical blocks, which simply abut each other. In such a case, it is not possible to use a hierarchical clock tree to synchronize a clock to all child physical blocks. Instead, each block provides the clock to the next. As the clock goes through several blocks, it accumulates a large delay because of the clock delay in each block. By the time the clock reaches the N th block, the delay is so large that it is no longer possible to reliably strobe the data on the tester. To avoid accumulating delay in a tiling configuration, you must implement the datapath through your physical block so that it goes and comes back through the block as shown in [Example 3 of the SSN](#) topic. Use a forward timing interface to go from one physical block to the next and a loop timing interface to get the returned data back from the next block. Both the input and the output data are localized in the first block, so it is easy to synchronize with the tester. Refer to the next section to see how to minimize the loop timing between the chip and the tester to increase the SSN shift rate.

[Figure 8-8](#) shows a block diagram of a chip where the SSN datapath starts from the center top of the device and makes a loop going down the middle of the chip. The datapath also branches out to both sides in mirrored groups of physical blocks called “Corea.” The GPIO are all localized in the top center block, including the clock input. The timing with the tester is completely localized into that physical block and is predictable before you complete the full chip layout. As [Example 3 of the SSN](#) topic illustrates, the timing of the datapath is localized between each pair of blocks and not affected by distant elements. In the example, there are three instances of corea on top of each other. These correspond to three Corea instances in following figure, where the bottom corea instance from the example is the Corea instance closest to the center in the figure.

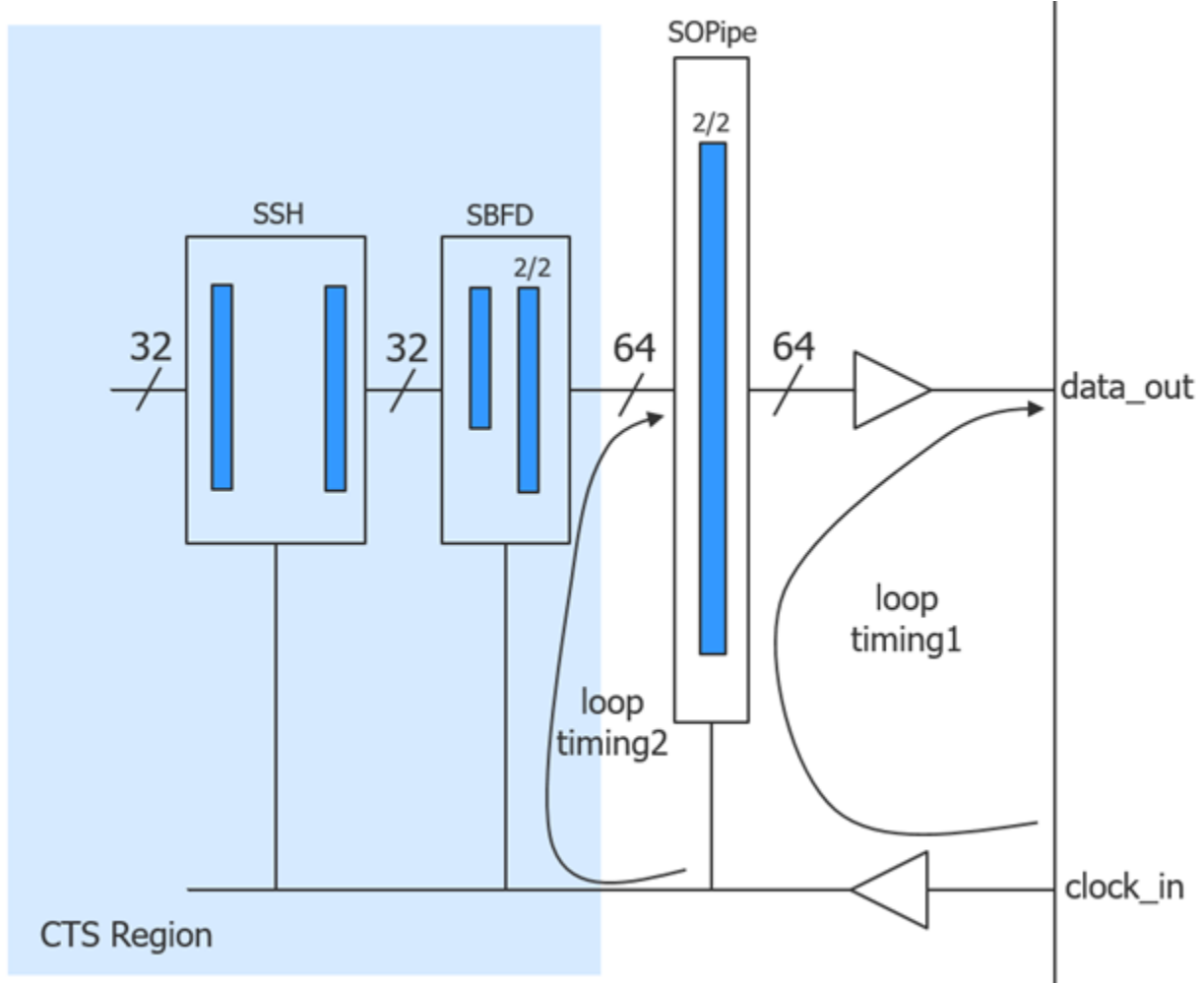
Figure 8-8. SSN Datapath in Tiling Architecture



Minimizing Loop Delay With the Tester

When a tester interacts with the SSN datapath of a chip, the critical timing path is always the loop timing path that starts from the clock supplied by the tester and continues to the data coming back from the chip. To be able to shift the SSN bus at 200 MHz, the worst case delay of the clock coming into the chip and reaching the last stage of the pipelining, with the data coming out of the chip, must be less than 4 ns. As described in the [Using the BusFrequencyDivider to Enable Faster Internal Bus Clock Frequency](#) section of the BusFrequencyDivider topic in the *Tessent Shell Reference Manual*, when the delay is 4 ns in the worst case, it is about 1.5 ns in the best case. This leaves very little opening in which to place a strobe point on the ATE that is reliable in both best- and worst-case conditions. To minimize the internal bus width, it is easy to operate the SSN bus at 400 MHz, but you need to use a BusFrequencyDivider node to bring the external data rate to 200 MHz so that the loop timing delay is less than a clock period. If you used a hierarchical clock tree, a large portion of the loop timing would be consumed in the clock tree delay itself. [Figure 8-9](#) shows how to use an [OutputPipeline](#) node placed on a miniature local clock tree to retime the data before it goes out the chip. This eliminates the large CTS delay in the loop timing.

Figure 8-9. Stepping Down the Clock Tree on the Last Pipeline Stage



Broadcast to Identical SSN Datapaths

When multiple identical SSN datapaths exist, it is beneficial to reuse the same copy of scan data going to each datapath. When you broadcast scan data to identical datapaths, you must set up observation for the outputs of each identical datapath independently.

You can use input broadcast for test when the physical implementation of the datapaths would benefit from broadcasting to their inputs from the same source. When you use a copy of the same data across multiple identical datapaths, this reduces scan data volume, because the same copy of input data tests the datapaths. Furthermore, testing identical datapaths concurrently can reduce test time.

However, not all designs with multiple datapaths benefit from input broadcast. For example, multi-die designs with nonidentical datapaths do not benefit from broadcasting their inputs from the same source, because nonidentical dies do not use the same scan payload. Instead, each datapath requires its own payload. Similarly, designs with mux access to their datapaths would

not benefit from input broadcast, because the mux access to the different dies requires serial testing.

“Identical” refers to both the physical specifications of the datapaths and the configuration of the datapaths. For example, if one instance has on-chip compare enabled and a second otherwise identical instance has on-chip compare disabled, these instances are not eligible for input broadcast. The datapaths must be physically identical and their SSHs must be configured identically. The only exception to this is that you can include extra pipelining or a *disabled* SSH (which acts like pipeline stages) after the last active SSH.


Even when broadcasting to identical datapaths, top-level ATPG can still be run. This means that broadcast to SSN datapaths has a broader use than retargeting. In this case, the lower-level instances must load the same scan data, whether performing retargeting or top-level ATPG.

Independent observation of datapaths is required both during loading and unloading of the datapath.

Yield Statistics on ATE With SSN

You may use per-pin fail limits to prevent a single core from filling up the fail buffer on the tester during manufacturing. You may also perform partial good die testing and need to identify failing cores live on the tester without performing full failure diagnosis. In these cases, you can create SSN patterns with packet constraints to enable both partial good die testing and yield statistics tabulation for failing cores while using per-pin fail limits. This feature is useful in cases where the pass-fail sticky bit is unavailable for indicating the presence of a failure.

Note

 Creating SSN patterns with packet constraints is useful for testing *non*-identical cores for which the SSH is not implemented with the on-chip compare feature.

Use the “set_ssn_options -packet_constraints no_rotation” command to stop rotation of the packet around the SSN bus, as shown in [Figure 8-10](#). In this figure, the packet is nine bits wide and contains five bits of Core A and four bits of Core B. The SSN bus is eight bits wide [7:0]. The packet rotates around the bus by one bit in bus_clock cycle 0 in the top half of the figure. As the bus_clock cycles increase, the packet continues to rotate around the bus. When you stop rotation of the packet with the set_ssn_options command, the tool pads the packet to make the packet size a multiple of the bus width, as shown in the bottom half of the figure. In this example, the packet increases to two bus_clock cycles wide (0 and 1), with seven bits of padding added to time slots one through seven in Cycle 1.

Figure 8-10. Packet Rotation on the SSN Bus

Normal operation (no padding, possible packet rotation)
Packet = 9 bits

		Cycle					
		5	4	3	2	1	0
Bus bit	7	A-5-2	A-4-3	A-3-4	B-2-0	B-1-1	B-0-2
	6	A-5-1	A-4-2	A-3-3	A-2-4	B-1-0	B-0-1
	5	A-5-0	A-4-1	A-3-2	A-2-3	A-1-4	B-0-0
	4	B-4-3	A-4-0	A-3-1	A-2-2	A-1-3	A-0-4
	3	B-4-2	B-3-3	A-3-0	A-2-1	A-1-2	A-0-3
	2	B-4-1	B-3-2	B-2-3	A-2-0	A-1-1	A-0-2
	1	B-4-0	B-3-1	B-2-2	B-1-3	A-1-0	A-0-1
	0	A-4-4	B-3-0	B-2-1	B-1-2	B-0-3	A-0-0



Packet padding to disable packet rotation
Packet = 16 bits (2 tester cycles)

		Cycle					
		5	4	3	2	1	0
Bus bit	7	----	B-2-2	----	B-1-2	----	B-0-2
	6	----	B-2-1	----	B-1-1	----	B-0-1
	5	----	B-2-0	----	B-1-0	----	B-0-0
	4	----	A-2-4	----	A-1-4	----	A-0-4
	3	----	A-2-3	----	A-1-3	----	A-0-3
	2	----	A-2-2	----	A-1-2	----	A-0-2
	1	----	A-2-1	----	A-1-1	----	A-0-1
	0	B-2-3	A-2-0	B-1-3	A-1-0	B-0-3	A-0-0

Regardless of the number of bus words the packet occupies, when you use the “set_ssn_options -packet_constraints no_rotation” command, the tool adds padding to the packet to make the packet a multiple of the bus width. This enables a predictable mapping from top-level data_out ports to the active SSHs. Because the packet can be multiple bus words, the mapping is different for each bus_clock cycle, as shown in the following example.

This example illustrates the annotations that the write_patterns command adds to the SSN STIL or WGL pattern files to show the SSH to SSN data_out port mapping required when you use the “set_ssn_options -packet_constraints no_rotation” command.

Example 8-1. Annotations for SSN Mapping With Packet Rotation Disabled

```
Ann { * TESSENT_PRAGMA ssn_mapping -begin * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[3]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[4]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[5]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[0]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[1]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[2]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[3]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[4]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[5]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[0]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[1]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[2]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2" * }
Ann { * TESSENT_PRAGMA ssn_mapping -end * }
```

This mapping enables the ATE to set up a fail limit per $pin \times (shift_cycle \% cycle_repetition)$, where *cycle_repetition* is the number of bus words a packet occupies. In this way, you can ensure that all cores can be observed and the ATE can enforce a per-pin limit for failure messages.

If you need to map back to the specific from_scan_out bus of the SSH without dealing with the complexities of bandwidth tuning, you can turn off data throttling, combined with deactivating the packet rotation. This typically costs test efficiency, so it may not suit your overall run time needs. Use the “set_ssn_options -packet_constraints no_rotation -throttling off” command to disable data throttling and stop the rotation of the packet around the bus.

The following example shows annotations that the `write_patterns` command adds to the SSN STIL or WGL pattern files to show the SSH to SSN `data_out` port mapping required when you use the “`set_ssn_options -packet_constraints no_rotation`” command.

Example 8-2. Annotations for SSN Mapping With Packet Rotation and Throttling Disabled


```
Ann { * TESSENT_PRAGMA ssn_mapping -begin * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[3]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3"
-ssh_chain_group 3 -ssh_chain_group_chain_index 2 * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[4]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3"
-ssh_chain_group 3 -ssh_chain_group_chain_index 3 * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[5]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 1 -ssh_chain_group_chain_index 0 * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[0]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3"
-ssh_chain_group 2 -ssh_chain_group_chain_index 4 * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[1]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3"
-ssh_chain_group 3 -ssh_chain_group_chain_index 0 * }
Ann { * TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 0 -design_port "ssn_bus_data_out[2]" -ssh_icl_instance
xtea_3.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_3"
-ssh_chain_group 3 -ssh_chain_group_chain_index 1 * }
```

```
Ann {* TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[3]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 1 -ssh_chain_group_chain_index 4 *}
Ann {* TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[4]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 2 -ssh_chain_group_chain_index 0 *}
Ann {* TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[5]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 2 -ssh_chain_group_chain_index 1 *}
Ann {* TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[0]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 1 -ssh_chain_group_chain_index 1 *}
Ann {* TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[1]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 1 -ssh_chain_group_chain_index 2 *}
Ann {* TESSENT_PRAGMA ssn_mapping -datapath_id 1 -cycle_repetition 2
-cycle_mod 1 -design_port "ssn_bus_data_out[2]" -ssh_icl_instance
xtea_2.xtea_rtl_tessent_ssn_scan_host_1_inst -core_instance "xtea_2"
-ssh_chain_group 1 -ssh_chain_group_chain_index 3 *}
Ann {* TESSENT_PRAGMA ssn_mapping -end *}
```

Manufacturing Patterns With SSN

When you use the streaming scan network (SSN), you create manufacturing patterns from a netlist that has been through layout and timing closure. You should create these patterns without any ATPG pattern limits to achieve maximum test coverage. Simulate these patterns using a cycle-based simulator, backannotating SDF delays on the design. For larger designs, it may not be practical to simulate all scan patterns.

Note

 The process of creating manufacturing patterns is different from creating signoff patterns. For a description of signoff patterns with SSN, refer to the section “[Signoff Patterns With SSN](#)” on page 444.

Use the `set_load_unload_timing_options` command to set the timing of the SSN when creating manufacturing patterns. If you share the SDC procedure `set_load_unload_timing_options` between the Tessent tools and static timing analysis, it is only required for you to source the shared file. Sharing the SDC procedure `set_load_unload_timing` ensures the timing of the manufacturing patterns written from the Tessent tools precisely matches the timing of the design.

[Table 8-1](#) shows the patterns you should use to test your silicon with SSN. These patterns are recommended for both first silicon test and production test. It is also recommended for both that you start with the least complex patterns and incrementally verify the SSN with each pattern up through the most complex patterns. The ICLNetwork Verify patterns are the least complex, gradually leading to Retargeted SSN patterns and (where applicable) Top-Level ATPG SSN patterns. For more detailed descriptions of the patterns in this table, refer to the section “[Signoff Patterns With SSN](#)” on page 444.

Only designs that you have implemented with on-chip compare require the OCComp Self-Test Pattern shown in this table. This patterns verifies the SSN on-chip compare logic is free of any stuck-at faults. For more information about SSN on-chip compare patterns, refer to the section “[How To Write SSN On-Chip Compare Patterns](#)” on page 435.


Table 8-1. SSN Manufacturing Pattern Summary

	Order of Priority	→	→	→	→	Order of Priority
	ICLNetwork Verify Patterns ¹	Continuity Pattern	OCComp Self-Test Pattern ²	Loopback Pattern	Retargeted SSN Patterns	Top-Level ATPG SSN Patterns
First silicon test	X	X	X	X	X (chain test)	X (chain test)
					X (scan test)	X (scan test)
Production test	3	3	X	3	X (chain test)	X (chain test)
					X (scan test)	X (scan test)

1. Includes verification of streaming-through-IJTAG

- 2. Required only when the SSH is equipped with on-chip compare
- 3. Optional pattern that can be used to identify cause of failure

Note

 Manufacturing patterns achieve maximum test coverage of the device under test and do not have any pattern limits when created.

The following topics describe the SSN patterns and, because you can write the procedures that make up SSN patterns to separate files, they also describe the strict order you must follow when applying the manufacturing patterns to the tester when you have written the different procedures of the SSN pattern to separate files:

Manufacturing Pattern Quick Reference	428
SSN Pattern Structure	429
How To Write Complete SSN Patterns	430
How To Write ssn_setup and ssn_end Procedures to Separate Files	432
How To Write SSN On-Chip Compare Patterns	435
How To Write Streaming-Through-IJTAG Patterns	436
SSN Debug on the Tester	438

Manufacturing Pattern Quick Reference

The table in this topic summarizes the signoff patterns for use at first silicon test and production test. You should simulate the Verilog pattern equivalent of these patterns without error before releasing them to be used on the tester.

Table 8-2. Manufacturing Pattern Quick Reference

Pattern Type	Limits	Description
ICLNetwork verify	None	Verifies ICL instruments (tool and user) are readable and writable along the IJTAG serial path. Verifies the full scope of the IJTAG network, including streaming-through-IJTAG path. Runs at TCK period.
Continuity	None	Verifies SSN bus integrity between bus data_in and data_out. Runs at bus_clock period.

Table 8-2. Manufacturing Pattern Quick Reference (cont.)

Pattern Type	Limits	Description
Parallel scan	None	Verifies each scan register can reliably capture. Capture clock and scan signals are cut points on SSH that test bench pulses. For retargeted scan patterns, can verify top-level clocking to lower-level cores.
OCCompare self-test	None	Verifies no stuck-at faults exist in SSH on-chip compare logic.
SSH loopback	None	Verifies SSN network can reliably deliver packetized data (excluding path to EDT). All SSN nodes are programmed with <code>ssn_setup</code> procedure as though full pattern is to be run. Runs at <code>bus_clock</code> period.
Serial chain	None	Delivers packetized data to all SSH and verifies all chains shift without error. Saves only nonmasking patterns. <code>bus_clock</code> in the SSH generates scan signals. Runs off <code>bus_clock</code> .
Serial scan	None	Delivers packetized data to all SSH and verifies all chains shift and capture without error. SSN end-to-end test. <code>bus_clock</code> in the SSH generates scan signals. Runs off <code>bus_clock</code> .

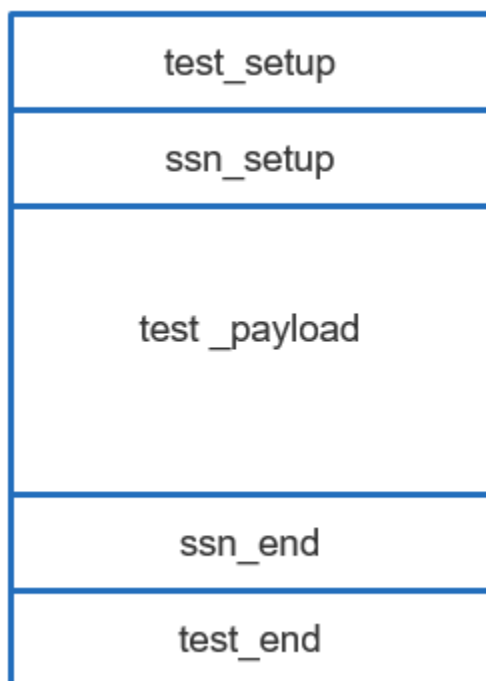
SSN Pattern Structure

This section describes the basic structure of the SSN pattern as the `write_patterns` command writes it.

When you use the streaming scan network (SSN), there are additional procedures that are part of every SSN pattern: `ssn_setup` and `ssn_end`. Furthermore, SSN delivers the scan test data as a stream of data over the SSN bus. The `ssn_setup` and `ssn_end` procedures combine with the existing `test_setup` and `test_end` procedures to create an SSN pattern.

The `ssn_setup` and `ssn_end` procedures configure the SSN nodes before and after you apply the `test_payload` to the network. The `test_payload` is packetized scan data that streams over the SSN bus, which delivers it to the streaming scan hosts (SSH) and scan chains. The `ssn_setup` procedure includes information to program the SSH nodes, and the `ssn_end` procedure resets all the active SSH nodes without resetting the SSH configuration. The `ssn_end` procedures also unload the sticky bits when on-chip compare is active.

Figure 8-11. Procedure and Data Structure of an SSN Pattern



How To Write Complete SSN Patterns

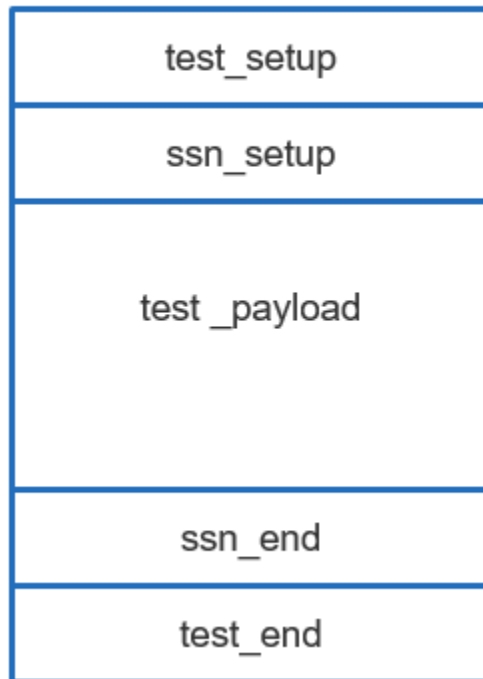
You can write SSN patterns to a single pattern file or to multiple files containing a set maximum number of scan loads and all of the required procedures.

To produce the simplest SSN scan pattern, run the `write_patterns` command using only a format and the `-replace` switch. The following example and figure show an SSN pattern with the SSN-specific procedures along with the `test_setup` and `test_end` procedures. The SSN procedures `ssn_setup` and `ssn_end` occur directly before and after the `test_payload`, respectively.

Example 8-3. Writing a Complete SSN Scan Pattern

```
write_patterns chip.stil -stil -replace
```

Figure 8-12. SSN Procedures With test_setup and test_end in Pattern Order
chip.stil

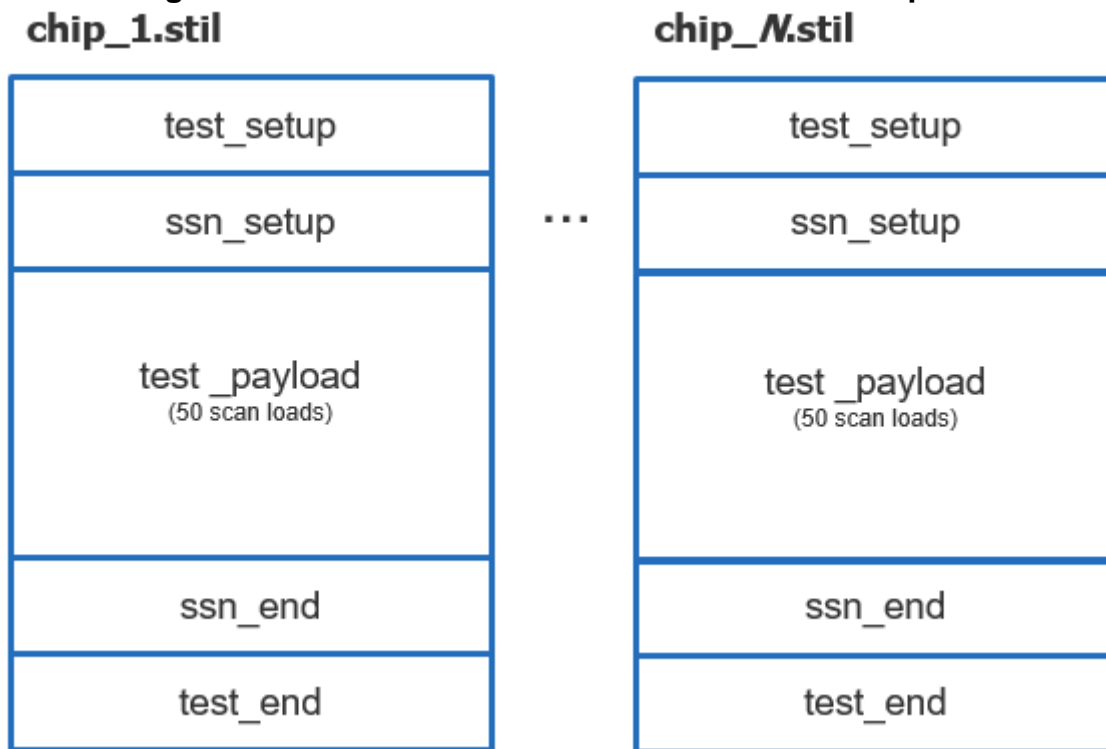


When you write SSN patterns using the `-maxloads` option, as in the following example, the `write_patterns` command duplicates the `test_setup`, `ssn_setup`, `test_payload`, `ssn_end`, and `test_end` procedures across all files, as the figure illustrates.

Example 8-4. Writing SSN Patterns With the `-maxloads 50` Option

```
write_patterns chip.stil -stil -maxloads 50 -replace
```

Figure 8-13. SSN Patterns With the -maxloads 50 Option



How To Write `ssn_setup` and `ssn_end` Procedures to Separate Files

Duplicating setup procedures for large manufacturing counts may result in the setup procedures dominating test time that should be allocated to applying the `test_payload`. To avoid this, you can write procedures for the SSN patterns to separate files in a similar fashion to how you can write `test_setup` and `test_end` procedures to separate files for non-SSN patterns. This enables more tester time to be spent applying `test_payload` patterns.


Caution

When you write out the `test_setup`, `ssn_setup`, `ssn_end`, `test_end`, and `test_payload` separately, you are responsible to ensure that the patterns are applied to the tester in the correct order. If they are not, you will observe mismatches on the SSN bus output.


Follow these rules when you write the procedures of SSN patterns to separate files:

- Each `write_patterns` command must use the same pattern parameters (such as `-begin`/`-end` and `-pattern_sets`).
- The individual procedures must be applied in the correct order on the tester.
- No additional clock cycles (including TCK) or pin constraints should be applied to the DUT before or after applying the different pattern files.

Tip

 SSN patterns are natively written to a single pattern file. Use `write_patterns` command options to control whether it excludes a procedure when it writes the pattern files

Tip

 When you write the SSN payload out separately and use the `-maxloads` option, ensure the number of loads is greater than the number of cycles of `ssn_setup` and `ssn_end`.

[Example 8-5](#) shows how to optionally write some of the procedures of the SSN pattern to separate pattern files. It writes the `test_setup`, `ssn_setup`, and `test_end` procedures separately from the SSH loopback and payload patterns. By writing out `test_setup`, `ssn_setup`, and `test_end` separately, you can apply numerous payload patterns without the penalty of applying slow TCK procedures more than once.

Example 8-5. SSN Patterns With Separate `test_setup` and `ssn_setup`

```
write_patterns test_setup.stil -test_setup only -stil -replace
write_patterns ssn_setup.stil -ssn_setup only -stil -replace
write_patterns payload.stil -test_setup off -ssn_setup off \
    -test_end off -stil -maxloads 500 -replace
write_patterns test_end.stil -test_end only -stil -replace
```

The patterns the commands in this example write must be applied on the tester in the correct order to avoid mismatches on the SSN bus output. The SSN data stream is dependent on the order of how the patterns are applied on the tester. [Figure 8-14](#) illustrates the correct order in which to apply the patterns.

The `test_setup` procedure depends on your design and your custom chip configuration. The `ssn_setup` procedure prepares the SSN network by configuring each SSH node. The payload patterns must be applied in the correct order to maintain the expected streaming data to the SSH. The `ssn_end` procedure, which is part of the payload pattern, is applied as part of the payload and prepares the network for the next pattern file. Finally, after the last payload pattern is applied, the `test_end` procedure is applied, reconfiguring the IJTAG network to a state that is equivalent to the state after `iReset`.

Note


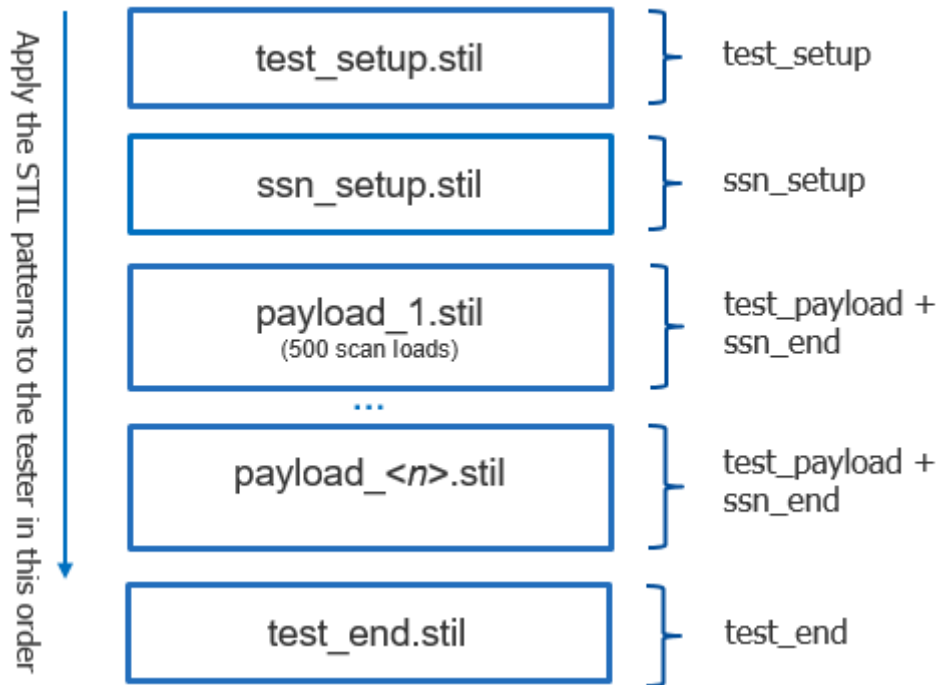
 The figure illustrates the correct order in which to apply the patterns.

Figure 8-14. SSN Pattern Files With Order for Application on Tester



Using -all_* Options To Minimize File Count

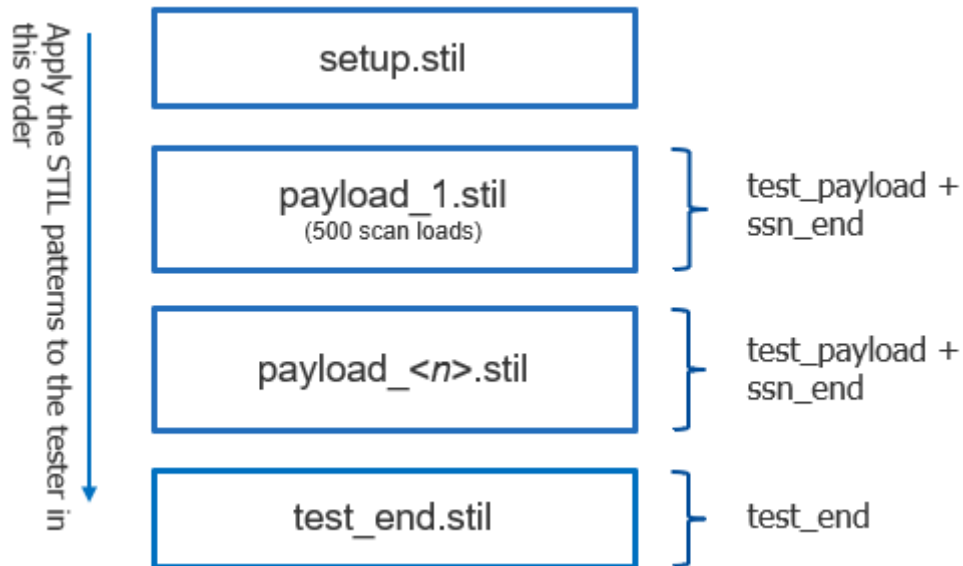
You can group the test_setup and ssn_setup procedures into a single file by specifying the -all_setup_only switch, as in the following example.

Example 8-6. SSN Pattern Files With Combined Single test_setup and ssn_setup

```
write_patterns setup.stil      -all_setup_only -stil -replace
write_patterns payload.stil    -test_setup off -ssn_setup off \
    -test_end off -stil -maxloads 500 -replace
write_patterns test_end.stil    -test_end only -stil -replace
```


When you write SSN patterns with the -all_setup_only switch, it produces a combined setup STIL file that is separate from the test_payload and end procedures, as in the following figure.

Figure 8-15. SSN Pattern Files With Combined Setup Showing Tester Order



You can similarly combine the end procedures, ssn_end and test_end, into a single pattern file using the “write_patterns -all_end_only” switch.

Note

 You cannot combine the -all_end_only switch with the maxloads option, because the ssn_end procedure must be applied after each payload pattern file, as shown in [Figure 8-14](#) on page 434 and [Figure 8-15](#).

How To Write SSN On-Chip Compare Patterns

The on-chip compare feature of SSN is one of the primary features to make your test process more efficient. It is common to use it when you have identical core instances or you are performing partial good die testing. When present, the on-chip compare hardware in the SSH compares expected and measured scan responses. You must test and verify that the on-chip compare hardware and the sticky bit status registers in each SSH are free of any stuck-at faults.

The following example shows how to write the on-chip compare self-test pattern alongside the other SSN patterns. It also uses the option to write the test_end procedure to a separate file. Use these commands as a guide for using on-chip compare with your design.

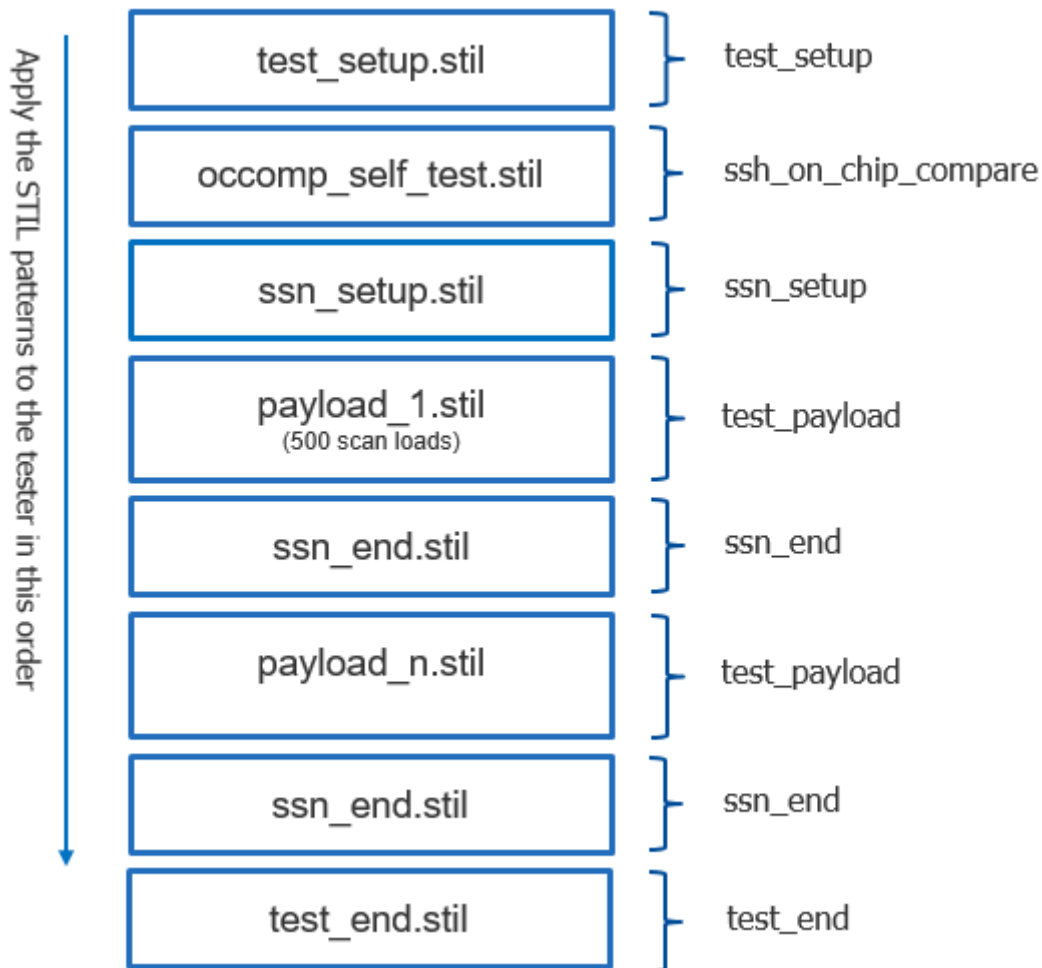
Example 8-7. Writing Each Procedure of the SSN Pattern to Individual Files

```
write_patterns test_setup.stil      -test_setup only -stil -replace
write_patterns occompsel_test.stil -test_payload   -stil -pattern_set ssh_on_chip_compare \
-replace
write_patterns ssn_setup.stil      -ssn_setup only  -stil -replace
write_patterns payload.stil        -test_payload    -stil -maxloads 500 -replace
write_patterns ssn_end.stil        -ssn_end only    -stil -replace
write_patterns test_end.stil       -test_end only   -stil -replace
```

The patterns in this example must be applied on the tester in the correct order to avoid mismatches on the SSN bus output. The SSN data stream depends on the order of how the patterns are applied on the tester.

This example extends the idea of the example in the section “[How To Write ssn_setup and ssn_end Procedures to Separate Files](#)” on page 432 by writing the on-chip compare self-test and the separate test_end procedure. As described previously, the on-chip compare self-test verifies the on-chip compare logic in the SSH. The on-chip compare self-test should be applied before any payload patterns when you have enabled on-chip compare.


Figure 8-16. SSN Pattern Files With On-Chip Compare Self-Test



How To Write Streaming-Through-IJTAG Patterns


Streaming-Through-IJTAG patterns are SSN patterns delivered to the active SSHs through TDI and measured on TDO with TCK as the synchronous clock source.

Note


 Streaming-Through-IJTAG patterns can only be applied when you have selected the streaming interface using the `set_ssn_options` command.

While Streaming-Through-IJTAG patterns are being applied, the TAP finite-state machine is put into the Shift-DR state while data is being delivered to the network through TDI. The SSH functional behavior during Streaming-Through-IJTAG is the same as when you use the parallel bus. The only difference is the delivery of streaming scan data to the network as a single bit through TDI.

Note

 Any SSN pattern created using the parallel bus can be retargeted through the top-level TAP controller and converted to a Streaming-Through-IJTAG pattern.

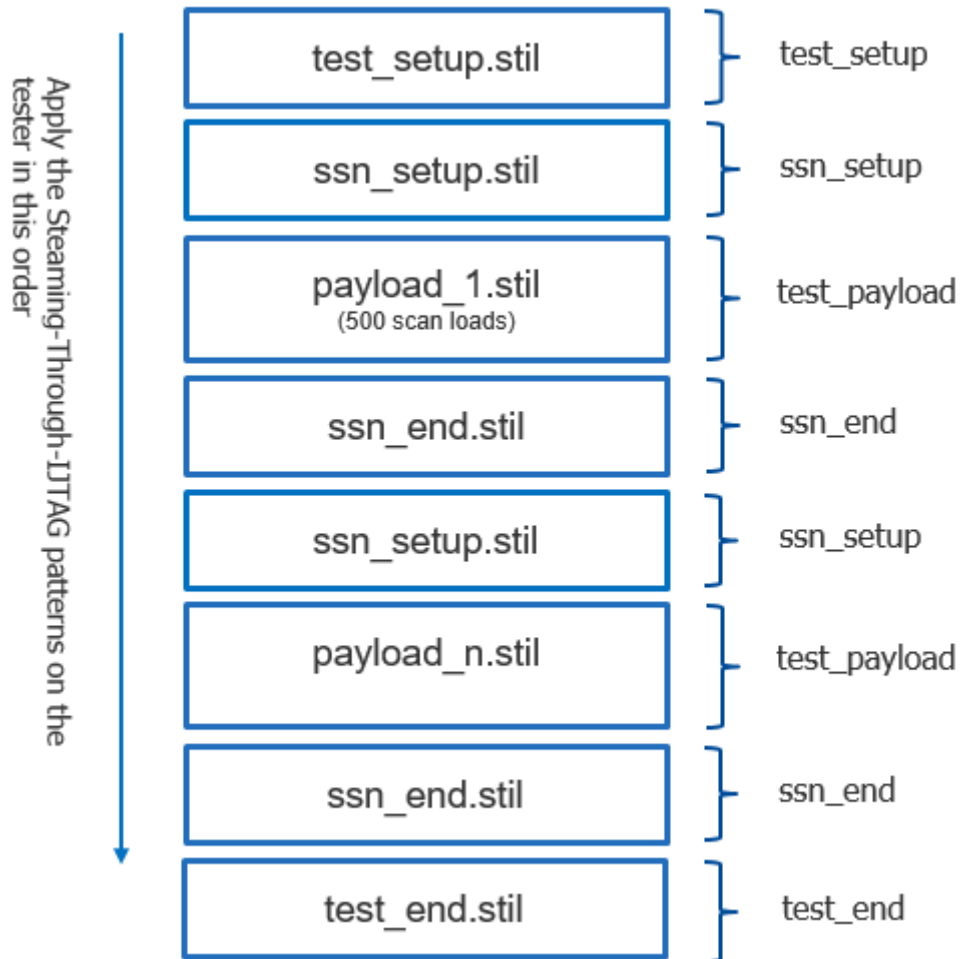
Note

 Writing the on-chip compare hardware self-test pattern is not compatible with Streaming-Thorough-IJTAG mode.

Example 8-8. Writing Streaming-Through-IJTAG Patterns

```
set_ssn_options -streaming_interface ijtag
...
write_patterns test_setup.stil -test_setup only -stil -replace
write_patterns ssn_setup.stil -ssn_setup only -stil -replace
write_patterns ssn_end.stil -ssn_end only -stil -replace
write_patterns payload.stil -test_payload -stil -maxloads 500 -replace
write_patterns test_end.stil -test_end only -stil -replace
```

Figure 8-17. Tester Application Order for Streaming-Through-IJTAG



SSN Debug on the Tester

Use a combination of patterns to help narrow down the source of the failure when SSN patterns fail on the tester. It is important to apply SSN patterns to the tester in the correct order.

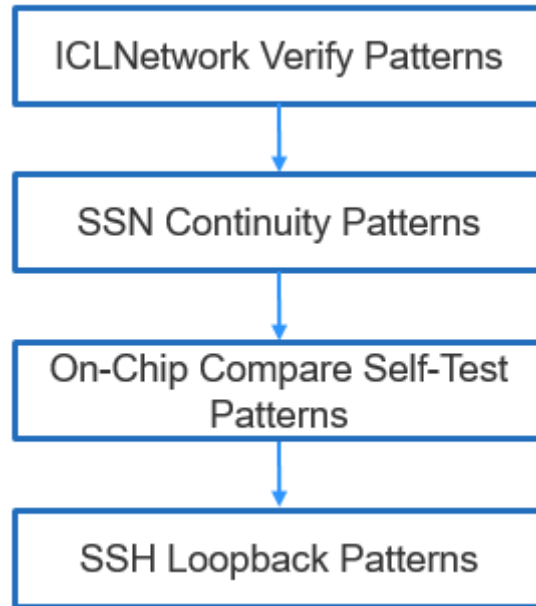
During silicon debug, you can reuse the verification patterns used during insertion and integration of the SSN. Use the following patterns to narrow the scope of the potential source of the failure when applied in the correct order, incrementally verifying the functional behavior of the SSN:

- **ICLNetwork Verify Patterns** — Confirms the IJTAG registers of the SSN can be accessed.
- **Continuity Patterns** — Confirms the parallel bus is connected to each SSN node and each branch of SSN multiplexer is accessible from bus_in to bus_out.
- **SSH Loopback Pattern** — Confirms the network can reliably deliver packetized data to the active SSHs without involving the EDT/scan chains.

- **On-Chip Compare Self-Test Pattern** — Confirms there are no stuck-at faults in the on-chip compare logic, including the sticky bit status registers.

The examples in the following sections show how to create and apply these patterns on the tester. This flow diagram indicates the order in which to apply them and incrementally verify the functional behavior of the SSN:

Figure 8-18. Order To Apply Patterns for SSN Pattern Failure Debugging



ICLNetwork Verify Patterns

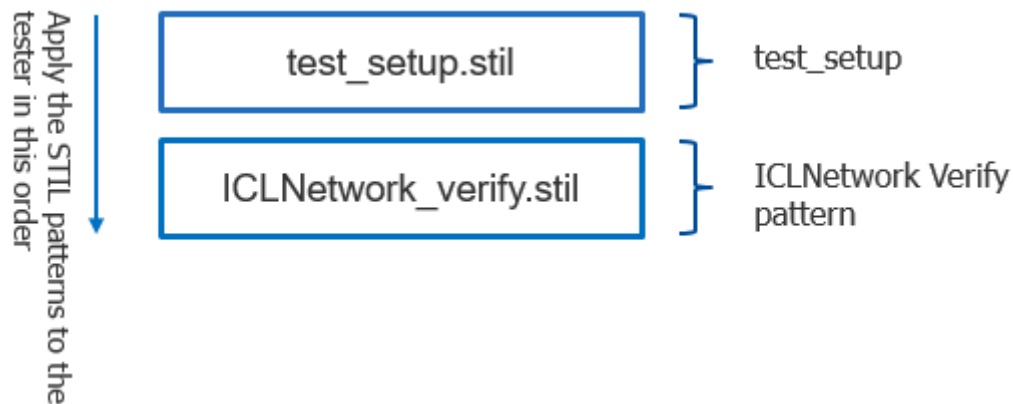
The following example shows the commands to write ICLNetwork verify patterns that you can apply on the tester. These patterns verify that the IJTAG registers in the SSN are accessible and that they can be read and written without any errors. Passing ICLNetwork verify patterns indicates the SSN network is properly configured, including the SSH nodes that are programmed as part of the `ssn_setup` procedure. To see the order of application for ICLNetwork verify patterns, refer to [Figure 8-19](#).

Example 8-9. Writing ICLNetwork Verify Patterns in STIL Format

```

set_context patterns -ijtag
...
open_pattern_set icl_network
  create_icl_verification_patterns
close_pattern_set
write_patterns ICLNetwork_verify.stil -stil -replace
  
```


Figure 8-19. Order To Apply ICLNetwork Verify Patterns to the Tester



Continuity Patterns

The following example shows the commands to write SSN continuity patterns for application on the tester. These patterns verify that the parallel bus is connected to each SSN node and that each branch of an SSN multiplexer is accessible from bus_in to bus_out. A passing SSN continuity pattern indicates the parallel bus in the design has no opens or gaps. To see the order of application for SSN continuity patterns, refer to [Figure 8-20](#).

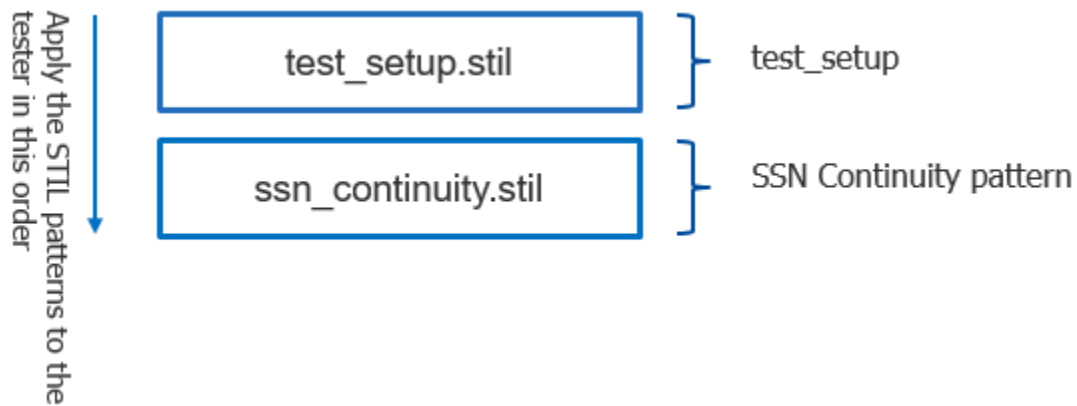
Note

 Each of the SSN multiplexers along the parallel bus is configured individually with an IJTAG iProc followed by an iCall to that procedure. Refer to the script in the section “[Second DFT Insertion Pass: Inserting Top-Level EDT, OCC, and SSN](#)” on page 385 for a working example of how the iCall reconfigures the SSN multiplexers.

Example 8-10. Writing SSN Continuity Patterns in STIL Format

```
set_context patterns -ijtag
...
open_pattern_set ssn_continuity -usage ssn
  create_ssn_continuity_patterns
close_pattern_set
write_patterns ssn_continuity.stil -stil -replace
```


Figure 8-20. Order To Apply SSN Continuity Patterns to the Tester



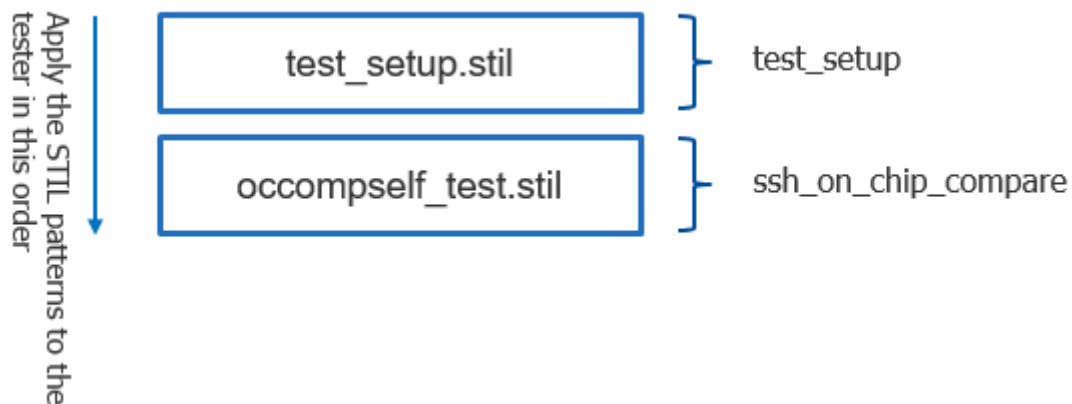
On-Chip Compare Self-Test Patterns

The SSH on-chip compare self-test is only necessary when one or more SSHs are equipped with on-chip compare hardware. The following example shows the commands to write the SSH on-chip compare self-test patterns for application on the tester. A passing on-chip compare self-test indicates the on-chip compare hardware and sticky bit status registers are free of any stuck-at faults. To see the order of application for on-chip compare self-test patterns, refer to [Figure 8-21](#).

Example 8-11. Writing SSH On-Chip Compare Self-Test Patterns

```
write_patterns test_setup.stil -test_setup only -stil -replace
write_patterns occompsel_test.stil -test_payload -stil \
  -pattern_set ssh_on_chip_compare -replace
```

Figure 8-21. Order To Apply SSN On-Chip Compare Self-Test Patterns



SSH Loopback Patterns

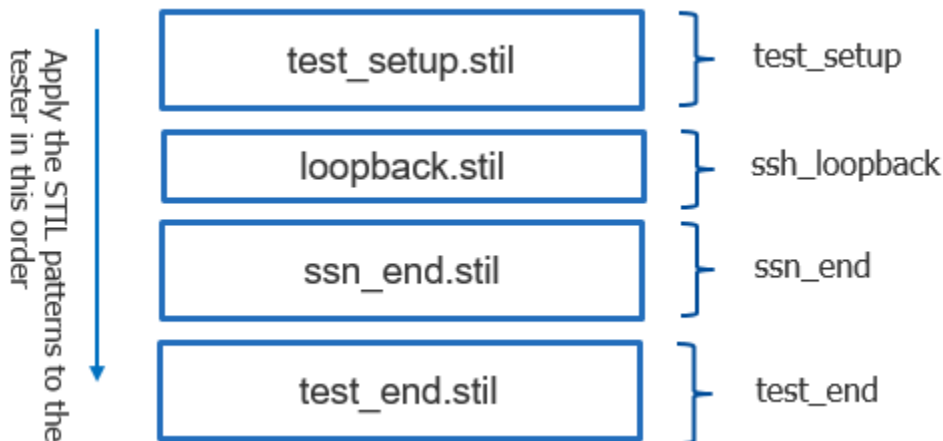
The SSH loopback pattern is the culmination of the first two test patterns: ICLNetwork verify and SSN continuity patters. Use this pattern with any configuration of active cores during ATPG and scan pattern retargeting. The following example shows commands to write the SSH loopback patterns for application on the tester.

Example 8-12. Writing SSH Loopback Patterns in STIL Format

```
write_patterns test_setup.stil -test_setup only -stil -replace
write_patterns ssn_setup.stil -ssn_setup only -stil -replace
write_patterns loopback.stil -test_payload -stil -pattern_set ssh_loopback -replace
write_patterns ssn_end.stil -ssn_end only -stil -replace
write_patterns test_end.stil -test_end only -stil -replace
```

In this example, the SSH loopback patterns test the SSH without involving the EDT and scan chains. The SSH loopback pattern uses the `ssn_setup` procedure to configure the SSH based on the payload of the active cores. During the SSH loopback pattern, the streaming interface (bus or IJTAG) delivers packetized data to each active SSH that is programmed to match the payload. The `ssn_end` procedure is applied to the network after the SSH loopback pattern. When you combine SSH loopback patterns and payload patterns, you must place the SSH loopback patterns before the payload patterns, because the SSH loopback patterns confirm that the network can reliably deliver packetized data, as described previously. A passing SSH loopback pattern indicates the SSN can reliably deliver packetized data to the current configuration of active cores. To see the order of application for SSH loopback patterns, refer to the following figure.

Figure 8-22. Order To Apply SSH Loopback Patterns

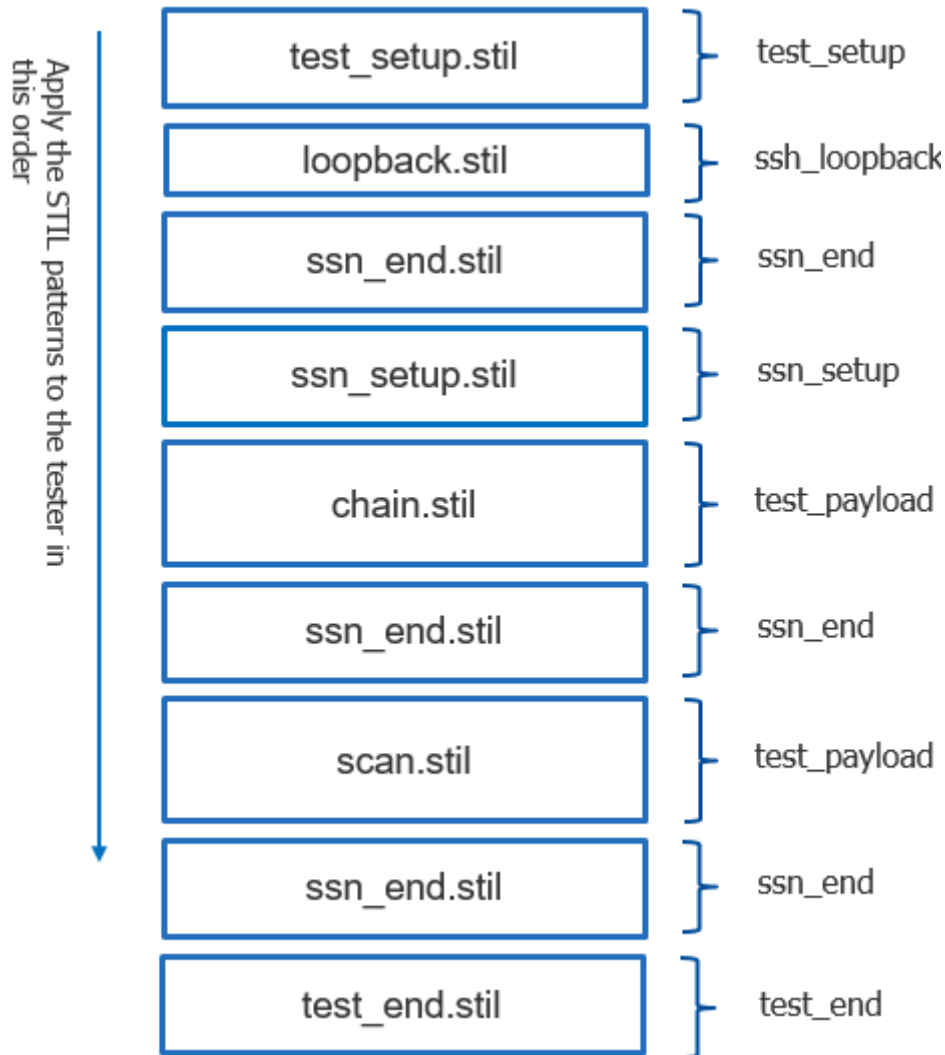


The following example shows the commands to write the chain and scan patterns with the loopback pattern for application on the tester. By extending the previous example to include the chain and scan patterns, you can further narrow the scope of failing patterns on silicon. To see the order of application for SSH loopback, chain, and scan patterns on the tester, refer to [Figure 8-23](#).

Example 8-13. Writing SSH Loopback, Chain, and Scan Patterns to Individual Files

```
write_patterns test_setup.stil -test_setup only -stil -replace
write_patterns ssn_setup.stil -ssn_setup only -stil -replace
write_patterns loopback.stil -test_payload -stil -pattern_set ssh_loopback -replace
write_patterns chain.stil -test_payload -stil -pattern_set chain -replace
write_patterns scan.stil -test_payload -stil -pattern_set scan -replace
write_patterns ssn_end.stil -ssn_end only -stil -replace
write_patterns test_end.stil -test_end only -stil -replace
```

Figure 8-23. Order To Apply SSH Loopback, Chain, and Scan Patterns



Signoff Patterns With SSN

This section describes the list of signoff patterns you should simulate at the block level, during top-level ATPG, and during scan retargeting. Signoff patterns are the minimum set of patterns you should simulate to verify your design with the DFT-inserted structures before releasing the design to layout. Use this signoff process to find potential problems in your design with a minimum set of patterns that provides a maximum amount of simulation coverage.

Note



The process of creating signoff patterns is different from creating manufacturing patterns. For a description of manufacturing patterns with SSN, refer to the section “[Manufacturing Patterns With SSN](#)” on page 427.

The following sections describe the unique behavior of each type of signoff pattern when written with SSN present. The signoff pattern guidelines in these sections minimizes the risk of problems occurring in your post-layout netlists. For each test, the section does the following:

- Identifies the synchronous source of the SSN.
- Explains the purpose of the test.
- Describes the behavior of the SSH when writing patterns at the block level, during top-level ATPG, and during pattern retargeting.

You should create signoff patterns without any ATPG pattern limits (for example, by specifying “[set_atpg_limits](#) -pattern_count off”). The methods for writing the patterns are described in the following sections. For more examples of writing patterns, refer to “[Tessent SSN Workflows](#)” on page 352. Create patterns using both the stuck-at and transition fault models. Simulate the patterns in the following order, which incrementally verifies the SSN network:

1. Parallel Scan
2. SSH Loopback
3. Serial Chain
4. Serial Scan


Each pattern uses the previously validated step. You should create and simulate these patterns until they are error-free before signoff, regardless of whether you choose parallel bus or IJTAG streaming for SSN.

Note



You should create signoff patterns only after you have successfully simulated the ICL Network Verify and SSN Continuity patterns without any errors.

Note

 Verification of the SSN at intermediate levels of hierarchy in your design is necessary prior to top-level ATPG or pattern retargeting when you have assembled the datapath by hand or using a script.

Block-Level Signoff Patterns	445
Top-Level Signoff Patterns	449
Signoff Pattern Quick Reference	452

Block-Level Signoff Patterns

You should simulate the block-level patterns without error before signoff. Block-level patterns include parallel scan patterns, SSH loopback patterns, serial chain patterns, and serial scan patterns. Each of these pattern types has its own set of prerequisites.

Parallel Scan Patterns

Prerequisites:

- ICL network verify patterns pass
- SSN continuity patterns pass

During parallel pattern simulation with SSN, only scan pins of synchronous elements are active during scan shift and capture procedures, as in testing without SSN. These patterns' purpose is to validate the capture procedure for a large volume of scan patterns by simulating them in a reasonable amount of time.

While you simulate these patterns, the synchronous clock source to the SSN is not active, and the data delivered to the scan cells is not packetized data. The SSH is not actively transferring data between the SSN bus and the EDT/scan chains, and the scan signals generated by the SSH in the serial patterns are replaced by cut points that the tool automatically creates. The transitioning of the scan signals precisely models the protocol of the scan pattern. The shift_capture_clock and capture clock run at their default clock periods unless you use the set_load_unload_timing_options and set_capture_timing_options commands to specify otherwise.

The following is an example of writing SSN parallel patterns:

```
write_patterns parallel.v -verilog -pattern_set scan -replace
```

This should be the first pattern you simulate before signoff.

SSH Loopback Patterns

Prerequisites:

- Parallel scan patterns pass

During SSH loopback pattern simulation, the tool delivers random packetized data to the SSH. These patterns' purpose is to verify that SSN can correctly process packetized data without error. The SSH loopback pattern tests exclude the EDT/scan chains from the test and include the SSH and other SSN nodes between bus_in and bus_out.

During simulation of this pattern, the SSN bus_clock is the clock source, and packetized data synchronously streams over the SSN. The SSH extracts the correct bits from the packet based on the ssn_setup procedure. The data loops back into the SSH, skipping the EDT/scan chains. After the data loops back into the SSH, it returns to the packet in the same time slot as the input data. As the packet synchronously streams through the network, the expected values are strobed on bus_out. During simulation of this pattern, only the bus_clock runs, and the scan signals from the SSH do not toggle and are constrained to zero.

The following is an example of writing SSH loopback patterns:

```
write_patterns ssh_loopback.v -verilog -serial -pattern_set ssh_loopback -replace
```

Serial Chain Patterns

Prerequisites:

- SSH loopback patterns pass

During serial chain pattern simulation, the tool delivers packetized scan shift data to the EDT/scan chains through SSH. These patterns' purpose is to verify that the scan chains can reliably shift scan data without error.

During simulation of these patterns, the SSN bus_clock is the clock source, and packetized data synchronously streams over the SSN. The SSH extracts the correct number of bits from the packet and transfers them to the EDT/scan chains. The scan signals are synchronously created within the SSH and clock the scan circuit and transition it between shift and capture states. Just as in non-SSN chain test patterns, the capture clock is suppressed. During unloading of the scan chains, the SSH puts the scan unload data back into the packet. When on-chip compare is off, the scan unload data is added back into the packet in the same time slots as the input data. When on-chip compare is enabled, the data is added back into the packet but does not overwrite the input data. It is added to the packet in a location determined by the number of status groups.

The shift_capture_clock runs at the default clock period unless you use the set_load_unload_timing_options command to specify otherwise.

Typically, the SSN bus_clock runs at a slower frequency than specified with the set_load_unload_timing_options command at the block level. The bus clock period is governed

by the packet size relative to the bus width. When the packet size is less than two bus_clock cycles, the SSN bus_clock is reduced to the shift_capture_clock frequency. To increase the bus_clock frequency to the default or to the maximum frequency you specified using the set_load_unload_timing_options command, use the [SIM_SSN_MAXIMUM_BUS_SPEED](#) test bench parameter. When you write the test bench with this parameter set to one, the tool increases the size of the packet so that the bus_clock runs at its maximum frequency in a Verilog test bench.

It can be difficult to debug mismatches in a serial SSN simulation, because the SSH obfuscates the EDT channels/scan chains. Furthermore, there may be any number of pipeline stages along the SSN bus, further complicating the analysis of the root cause to the failure. To debug serial SSN simulations, use the [SIM_PARALLEL_MONITOR](#) test bench parameter. When you set this parameter to one, the test bench monitors the load and unload values of each memory element and echoes the instance name to the log file for easier analysis of any failure.

For large designs, simulating all chain test patterns can be time-consuming and impractical. Use the “set_chain_test -type nomask” command to enable you to efficiently simulate the shift path of all your scan chains without testing the magic logic of the EDT decoder. Combine the “set_chain_test -type nomask” command with the “write_patterns -end 0” command to write a single chain test pattern that tests the shift paths of all the scan chains. A single chain test pattern that simulates all scan chains is all you require.

The following example commands write a single serial chain test pattern that excludes testing any of the EDT decode logic:

```
set_chain_test -type nomask  
write_patterns chain_test.v -verilog -serial -end 0 -pattern_set chain -replace
```

Serial Scan Patterns

Prerequisites:

- SSN serial chain patterns pass

During serial scan pattern simulation, the tool delivers packetized scan shift data to the EDT/scan chains through SSH. These patterns’ purpose is to verify that the scan chains can reliably shift and capture without error.

When simulating this pattern, the SSN network operates exactly as described in the preceding subsection, Serial Chain Patterns, with the addition that capture cycles are included in this pattern. The correct number of capture pulses comes through the OCC clock control bits. During stuck-at fault testing, the SSH synchronously creates the capture clock from the SSN bus_clock. The SSH capture clock frequency comes from the default clock period unless you use the set_capture_timing_options command to specify otherwise. During transition fault testing, the functional clock is the capture clock. This pattern is an SSN end-to-end test.

For large designs, simulating all serial scan patterns can be time-consuming and impractical. Use the “write_patterns -end 0” command to write a single end-to-end scan test pattern that is sufficient to test the SSN along with the previously specified patterns.

You can use the SIM_SSN_MAXIMUM_BUS_SPEED test bench parameter, as described in the [Serial Chain Patterns](#) section, to simulate the serial scan patterns at the maximum SSN bus frequency.

You can use the SIM_PARALLEL_MONITOR test bench parameter, as described in the Serial Chain Patterns section, to debug the serial scan patterns.

The following example command writes a single serial scan test pattern:

```
write_patterns scan_test.v -verilog -serial -end 0 -pattern_set scan -replace
```

Block-Level Signoff Pattern Summary

The following table summarizes the block-level patterns used for verification from integration to pattern creation of the SSN. You should run the pattern verification from least complex to most complex. When you follow this order, it verifies the SSN incrementally, making it easier to identify problems along the way.

Depending on your flow, you can perform verification of the SSN during integration either at the RTL or the gate level. For both cases, the ICLNetwork verification patterns also verify the [Streaming-Through-IJTAG Scan Data](#) path of the SSN. The entry point and path within the SSH is the same, regardless of access IJTAG registers in the SSH or streaming scan data through the IJTAG interface of the SSH.

Table 8-3. Block-Level Verification Pattern Summary

	Least complex	→	→	→	Most complex
	ICLNetwork Verify Patterns ¹	Continuity Pattern	Scan Pattern Parallel	Loopback Pattern	Scan Pattern Serial
SSN integration ²	X	X			
Post-synthesis validation (non-scan)	X	X			
Core-level ATPG pattern validation				X	X (chain test)
			X (scan test)		X (scan test)

1. Includes verification of the streaming-through-IJTAG SSH path

2. SSN integration at RTL, gate level, or both

Top-Level Signoff Patterns

You should simulate the top-level patterns without error before signoff. Block-level patterns include parallel scan patterns, SSH loopback patterns, serial chain patterns, and serial scan patterns. Each of these pattern types has its own set of prerequisites.

As part of the signoff process, you should retarget cores on a per-module basis. Do not group different modules for retargeting during signoff. Grouping different modules for retargeting is a process performed for manufacturing.

Parallel Scan Patterns

Prerequisites:

- ICL network verify patterns pass
- SSN continuity patterns pass
- Child core scan patterns pass, as described in “[Block-Level Signoff Patterns](#)” on page 445
- Scan graybox has been created for all lower-level blocks

Top-level parallel scan patterns operate in two different situations:

- **Parallel pattern during top-level ATPG with child cores** — During ATPG, the top-level scan chains and the wrapper scan chains of each child core are active. This pattern tests the intra-domain capture between the top level and the child cores. The active SSHs in the parallel pattern are capture-aligned. You should write the parallel pattern without a pattern count limit.
- **Parallel pattern during pattern retargeting** — Normally, during scan pattern retargeting with SSN, each core transitions between shift and capture independently. However, a Verilog test bench cannot accurately model this behavior for the cores. Therefore, in this test bench the active cores appear to be capture-aligned, although this is not the way that the implemented design will behave. Use this pattern to verify the clocking during pattern retargeting.

For a description of parallel scan pattern behavior and an example of how to write the pattern, refer to the section “[Parallel Scan Patterns](#)” in the topic “[Block-Level Signoff Patterns](#)” on page 445.

SSH Loopback Patterns

Prerequisites:

- SSN parallel scan patterns pass

During top-level ATPG when child cores are present, the top-level SSH loopback pattern delivers packetized data to the top-level SSH and the SSH of each child core. The cores are in

external mode. During simulation of this pattern, the tool forces the scan signals at the edges of the SSHs low. The packet contains data for the top-level SSH and the SSH for each child core. This pattern is short and simulates quickly. It verifies that each SSH can precisely extract and replace data in the packet.

You cannot retarget SSH loopback patterns. During pattern retargeting, the packet used in the SSH loopback test is created from the scan data for each core whose patterns is being retargeted. In this case, the purpose of the SSH loopback pattern is to test the SSN with a representative packet to be used during pattern retargeting.

For a description of SSH loopback pattern behavior and an example of how to write the pattern, refer to the section “[SSH Loopback Patterns](#)” in the topic “[Block-Level Signoff Patterns](#)” on page 445.

Serial Chain Patterns

Prerequisites:

- SSH loopback patterns pass

Top-level serial chain patterns operate in two different situations:

- **Serial chain pattern during top-level ATPG with child cores** — During ATPG, this pattern delivers packetized scan shift data to the top-level SSH and to the SSH of each child core. The child cores are in external mode, and the local SSH to those child cores shifts their wrapper chains. During simulation of this pattern, the capture clock is suppressed and the top-level SSH and child core SSHs are capture-aligned.
- **Serial chain pattern during pattern retargeting** — During simulation of this pattern, the tool creates the packet from each core whose patterns are being retargeted. Each core operates independently of the others, receiving a different number of bits per packet based on data throttling.

For a description of serial chain pattern behavior and an example of how to write the pattern, refer to the section “[Serial Chain Patterns](#)” in the topic “[Block-Level Signoff Patterns](#)” on page 445.

Serial Scan Patterns

Prerequisites:

- Serial chain patterns pass

Top-level serial chain patterns operate in two different situations:

- **Serial scan pattern during top-level ATPG with child cores** — During ATPG, this pattern delivers packetized scan data to the top-level SSH and to the SSH of each child core. The child cores are in external mode, and the local SSH to those child cores shifts

their wrapper chains. During simulation of this pattern, the scan chains shift and capture, and the top-level SSH and child core SSHs are capture-aligned.

- **Serial scan pattern during pattern retargeting** — During simulation of this pattern, the tool creates the packet from each core whose patterns are being retargeted. Each core operates independently of the others, receiving a different number of bits per packet based on data throttling.

For a description of serial chain pattern behavior and an example of how to write the pattern, refer to the section “[Serial Scan Patterns](#)” in the topic “[Block-Level Signoff Patterns](#)” on page 445.

Top-Level Signoff Pattern Summary

The following table summarizes the top-level patterns used for verification from integration to pattern creation of the SSN. You should run the pattern verification from least complex to most complex. If your design has more than two levels of hierarchy (for example, more than block and top), you should verify the SSN at all intermediate levels. This ensures that the datapath is properly connected. When you follow this order, it verifies the SSN incrementally, making it easier to identify problems along the datapath.

Depending on your flow, you can perform verification of the SSN during integration either at the RTL or the gate level. For both cases, the ICLNetwork verification patterns also verify the [Streaming-Through-IJTAG Scan Data](#) path of the SSN. The entry point and path within the SSH is the same, regardless of access IJTAG registers in the SSH or streaming scan data through the IJTAG interface of the SSH.

Table 8-4. Top-Level Verification Pattern Summary

	Least complex	→	→	→	Most complex
	ICLNetwork Verify Patterns ¹	Continuity Pattern	Scan Pattern Parallel	Loopback Pattern	Scan Pattern Serial
SSN integration ²	X	X			
Post-synthesis validation (non-scan)	X	X			
Top-level ATPG pattern validation				X	X (chain test)
			X (scan test)		X (scan test)
Pattern retargeting validation				X	X (chain test)
			X (scan test) ³		X (scan test)

1. Includes verification of streaming-through-IJTAG

2. SSN integration at RTL, gate level, or both

3. SSN retargeting parallel test bench only verifies clocking during scan pattern retargeting

Signoff Pattern Quick Reference

The table in this topic summarizes the signoff patterns at the block and top levels. You should simulate these patterns without error before releasing the design for layout.

Table 8-5. Signoff Pattern Quick Reference

Pattern Type	Limits	Description
ICLNetwork verify	None	Verifies ICL instruments (tool and user) are readable and writable along the IJTAG serial path. Verifies the full scope of the IJTAG network, including streaming-through-IJTAG path. Runs at TCK period.
Continuity	None	Verifies SSN bus integrity between bus data_in and data_out. Runs at bus_clock period.
Parallel scan	None	Verifies each scan register can reliably capture. Capture clock and scan signals are cut points on SSH that test bench pulses. For retargeted scan patterns, can verify top-level clocking to lower-level cores.
SSH loopback	None	Verifies SSN network can reliably deliver packetized data (excluding path to EDT). All SSN nodes are programmed with ssn_setup procedure as though full pattern is to be run. Runs at bus_clock period.
Serial chain	1	Delivers packetized data to all SSH and verifies all chains shift without error. Saves only nonmasking patterns. bus_clock in the SSH generates scan signals. Runs off bus_clock.
Serial scan	1	Delivers packetized data to all SSH and verifies all chains shift and capture without error. SSN end-to-end test. bus_clock in the SSH generates scan signals. Runs off bus_clock.

SSN SDC Constraints in the Design Flow

The following topics describe how to work with SDC constraints when SSN is present in the design.

Overview of SSN-Related SDC Procs and Constraints	454
Changes to SDC Procs for SSN Timing	455
SSN/SDC Proc Usage	456
SSN/SSH SDC Constraint Descriptions	462
SSN Bus Clock and SSN Bus Port Timing Constraints	462
SSN Bus Data Port Delays	463
SSN Datapath Timing Constraints	465
SSN FIFO Timing Constraints	468
SSN ScanHost Timing Constraints	471

Overview of SSN-Related SDC Procs and Constraints

SSN requires updates to the logictest-related constraint procs in the SDC output file that the `extract_sdc` command produces.

The following topics describe these updates to the SDC file procs and how to use them in synthesis, layout, hierarchical signoff static timing analysis (STA), and full-chip STA in the SSN context.

SSN SDC constraints work in conjunction with many of the Tcl procs dedicated to Tessent-logictest SDC constraints, such as those that define the `shift`, `slow_capture`, and `fast_capture` STA modes. This is possible because SSN is simply a faster and more flexible scan implementation than traditional implementations.

The generated constraints added for SSN perform the following functions:

- Declare the SSN bus clocks and define the timing of the SSN bus ports.
- Relax paths across node multipliers and node dividers with MCP constraints.
- Accurately time the SSH scan interface circuits, which includes creating two types of `edt_clock` and `shift_capture_clock`, and adding timing exceptions to the `scan_enable` and `edt_update` circuits, connections to LE (Leading Edge) and TE (Trailing Edge) scan flops, and loopback circuits.
- Add timing exceptions to and from subphysical block boundaries.
- Provide a way to time the SSN/SSH logic of the lower-level physical blocks.
- Properly time SSN nodes that can only stream through IJTAG, with no SSN bus clock.
- Provide ways to deactivate the SSN logic if needed, and prevent other mode clocks, such as `ijtag_tck`, from propagating to the scan flops.

For more information about constraints, refer to the section “[SSN/SSH SDC Constraint Descriptions](#)” on page 462.

The chapter “[Timing Constraints \(SDC\)](#)” on page 715 describes the flow usage of the standard logictest Tcl procs. The addition of SSN constraints adds only minor changes to the usage of these procs. You still use the following design flow:

1. In synthesis, invoke the `non_modal SDC` proc constraints.
2. In layouts, sequentially invoke `non_modal SDC` constraints followed by some modal logictest SDC procs.
3. In post-layout STA, time your different logictest modes one at a time, calling the provided “`*ltest_modal*`” procs for each mode.

The following sections provide more information about SDC procs with SSN:


Changes to SDC Procs for SSN Timing..... 455
SSN/SDC Proc Usage..... 456

Changes to SDC Procs for SSN Timing

The presence of SSN adds new constraint procs to the SDC file and modifies some other preexisting logictest-related SDC procs.

For a detailed description of the procs that are modified for SSN, refer to their original descriptions in the section “[LOGICTEST Instruments](#)” on page 735.

Note

 When you specify the SSH [scan_signals_bypass](#), all modified logictest proc constraints cover both the SSN mode and the bypass mode timing paths at the same time, so there is no need to separate your bypass-mode STA from your SSN-mode STA runs.

Summary of New and Modified Tcl Procs for SSN Timing Support

```
<prefix> ::= tessent_set
```

New SSN-Specific Procs

<prefix> _ltest_ssn

- Applies set_input/output delay constraints on SSN bus ports
- Creates clock groups between SSN clocks, functional clocks, and other DFT clocks
- Adds SSH constraints
- Adds MCP constraints for frequency multiplier and divider nodes
- Accepts “mode” argument with values “shift” or “non_modal”

set_load_unload_timing_options

- The SDC file equivalent of the command of the same name in Tessent Shell
 For usage information, refer to the [set_load_unload_timing_options](#) command reference page.
- Sets global Tcl timing variable values, which are used in SSN constraints

Modified Preexisting Procs

<prefix> _ltest_create_clocks

- Creates SSN bus clocks on top-level ports and generated clocks within the SSH logic

<prefix>_ltest_non_modal

- Calls <prefix>_ltest_ssn with the argument “non_modal”

<prefix>_ltest_modal_shift

- Calls <prefix>_ltest_ssn with the argument “shift”

<prefix>_ltest_modal_edt_slow_capture

- Calls <prefix>_ltest_ssn with the argument “non_modal”

<prefix>_ltest_modal_edt_fast_capture

- Forces the SSH scan_en output signal inactive

<prefix>_ltest_disable

- Blocks tessent_tck from propagating to SSH and scannable logic

New Procs for Chip-Level SSN STA

These are global chip-level versions of the procs in the previous subsection, which apply the same constraints over all SSN node instances across sub-physical block hierarchies.

- <prefix>_ltest_ssn_with_sub_PBs
- <prefix>_ltest_create_clocks_with_sub_PBs
- <prefix>_ltest_modal_shift_with_sub_PBs
- <prefix>_ltest_modal_edt_slow_capture_with_sub_PBs
- <prefix>_ltest_modal_edt_fast_capture_with_sub_PBs

SSN/SDC Proc Usage

Use SSN-related SDC procs during preparation for synthesis, layout, and STA (both pre- and post-layout). SSN/SSH implementation has a few important points in which it differs from implementation without SSN.

Much of this material is covered in greater detail in some portions of the chapter “[Timing Constraints \(SDC\)](#)” on page 715. However, the following sections describe some details that are specific to SSN/SSH implementations.

SSN/SDC Constraints for Synthesis

Preparing a block with SSN for RTL-to-gate synthesis does not differ significantly from preparing a standard EDT-inserted block, as described in “[Timing Constraints \(SDC\)](#)” on page 715. That section contains example synthesis scripts for both Design Compiler and Cadence Encounter that can help you understand the SDC constraints for synthesis. The

primary difference from the non-SSN flows for SSN is that you must specify your SSH scan interface global Tcl variables by calling the `set_load_unload_timing_options` proc instead of overriding them directly with a “set” command in your mainSDC script. Refer to the [set_load_unload_timing_options](#) command reference page for usage information for the command and Tcl proc.

The following summarizes the usage flow for preparing a block with SSN for synthesis:

1. Load your design.
2. Set your functional constraints.
3. Source the Siemens EDA *<design>.sdc* file.
4. Call the `tessent_default_variables` proc.
5. Redefine the global Tessent variables to meet your needs.
6. Call the “`set_load_unload_timing_options -usage ssn`” proc with the proper option values.


This redefines the default SSH global Tcl timing variables.

7. Call the `<prefix>_non_modal` proc.

This adds SDC for all of your Tessent DFT, including SSN.

8. Add synthesis control commands and run synthesis according to the process in the “[Timing Constraints \(SDC\)](#)” chapter.

Note

 The `<prefix>_non_modal` proc declares all SSN bus clocks and SSH-generated scan clocks and propagates them alongside your functional clocks to all your scannable domains. Such clocks may expose some invalid shift_capture_clock (SCC)-speed capture paths between otherwise asynchronous functional domains. Because synthesis usually runs with ideal clocks, these bogus timing paths are unlikely to fail hold timing; however, it is possible that they could still fail setup timing if your specified scan frequency is too high. Such a proc is also unusable as-is in layout without major modification, as described in the subsection “[Single-Mode vs Dual-Mode Constraining in Synthesis/Layout](#)” in the section “[LOGICTEST Instruments](#)” on page 735.

SSN/SDC Constraints for Layout

As described in the section “[Running Layout with Tessent Shell DFT](#)” on page 723, the recommendation for layout with Tessent is to load multiple timing mode constraints. This section also shows the preparation steps required prior to loading the constraints themselves.

If your design includes at least one SSH controller, your clock tree synthesis (CTS) step must include the CTS stop point declarations listed by the `tessent_get_cts_skew_groups_dict` proc

inside your generated SDC file. These points are located at the base of the SSH `edt_clock` and `shift_capture_clock` generated clock source pins. These prevent the potentially problematic balancing of the SSN datapath clock network with your potentially very large scan clock network.

The following sections describe how the presence of SSN affects these SDC constraints.

Mode 1: All Tessent DFT Logic Except `logictest`

proc: `<prefix>_non_modal off`

Loads non-modal constraints for all Tessent DFT controllers but turns off the `logictest` DFT logic, including all SSN bus nodes and the SSH. In this mode, some constraints are needed to prevent `tessent_tck` from propagating to the scannable domains through the SSH “ijtag streaming” and “capture with tck” logic. For more details, refer to the subsection “`<ltest_prefix>_non_modal`” on page 744.

Mode 2: SSN Bus and Modal Shift

proc: `<prefix>_modal_shift`

Forces the SSH and SSH bypass `scan_enable` signals to be tied active and covers all SSN bus and SSH scan interface timing paths. If the SSH supports bypass mode, both bypass and SSH modes are covered at once.

Mode 3: SSN Bus and Capture With Slow Clock

proc: `<prefix>_modal_edt_slow_capture`

Adds the same SSN constraints as the `<prefix>_modal_shift` proc but leaves the SSH `scan_en` signals toggling in order to time them using specifications from the [set_load_unload_timing_options](#) command. Optionally relaxes SCC intra- and cross-domain paths to avoid false violations. Its main function is to cover the `scan_en` signal timing path.

You can merge the preceding Modes 2 and 3 into a single stage to save on total layout runtime and complexity. You can do this only if your design meets the following criteria:

- Clock tree synthesis (CTS) has balanced the entire SSH `shift_capture_clock` source fanout, along with the same SSH `edt_clock` source fanout, so that there are no hold issues on same-edge cross-domain paths.
- All cross-domain paths can still meet setup timing in one SCC cycle. Capture clock waveforms are assumed to be the same as those of the shift clock.

In this case, you only need to run the `<prefix>_modal_edt_slow_capture` proc, which then covers both the shift paths and capture paths in addition to properly timing the `scan_enable` signal with the specified number of setup and hold cycles.

For designs with SSN, the `<prefix>_ltest_modal_edt_slow_capture` proc includes the possibility of defining one set of generated clocks per OCC in the fanout of each SSH. This results in relaxed hold for all capture between different OCCs. This also relaxes paths between SIBs or Bscan to or from other OCC domains. Turn on this feature by defining the following global Tcl variable in your top-level calling script:

```
set tessent_relax_xdomain_capture_paths 1
```

Setting this variable means the tool supports hardened OCCs and all-design-level STA. Each generated clock is defined at the OCC `slow_clock` input pin, to permit a possible `shift_capture_clock` from the parent level to go through the `fast_clock` pin in multi-level designs. If the OCC persistent `slow_clock` buffer is visible in the netlist, the multicycle path constraint is defined on its input pin.

You can modify each generated clock OCC pin by editing a global OCC dictionary, but support for full custom OCC is not available.

If you do not define this variable, you can also choose from the following options:

- Not relaxing any path—this means closing stuck-at slow capture timing, for setup and hold, across those generated clocks coming from the OCC
- (default) Applying a global same-edge multicycle path -hold to the SSH `shift_capture_clocks`

Limitation for bypass mode constraints: When the SSH bypass is present, the bypass mode `scan_en` and `edt_update` input ports have no MCP declaration, unlike the SSH local `scan_en` signal sources, which MCP constraints relax based on timing specifications from the `set_load_unload_timing_options` command. For more details, refer to the section “[SSN/SSH SDC Constraint Descriptions](#)” on page 462.

Furthermore, the bypass mode test clock frequency is assumed to be the same as the SSH test clock frequency, which is unlikely to be the case. If you intend to run the SSH-bypass scan at a slower frequency than the SSH-based scan, you must override the SSH-bypass `test_clock` definition in your main script after calling the procs listed previously. You may also need to add your own MCP declarations for both `scan_en` and `edt_update` top-level ports, as in the following example:


```
<prefix>_modal_edt_slow_capture
create_clock -period <bypass test_clock period> \
  [get_ports test_clock port]
Set_multicycle_path 2 -setup -from [get_ports <scan_enable port>]
Set_multicycle_path 2 -hold -from [get_ports <scan_enable port>]
Set_multicycle_path 2 -setup -from [get_ports <edt_update port>]
Set_multicycle_path 2 -hold -from [get_ports <edt_update port>]
```

SSN/SDC Constraints for STA Signoff on the Current Design

The Tessent logictest DFT STA Signoff flow is described in the sections “[Checking Your Functional Logic Alone](#)” on page 726 and “[LOGICTEST Instruments](#)” on page 735. The modal SDC procs these sections describe also cover both the fast SSN bus logic and the SSH scan interface logic. For more details on the SSN constraints, refer to the section “[SSN/SSH SDC Constraint Descriptions](#)” on page 462. Therefore, to perform STA signoff on your SSN DFT logic, you must run the same modes and the same procs as for the EDT/logictest logic; that is, the following:

- **Mode shift STA** — Call proc `<prefix>_modal_shift`
- **Mode edt_slow_capture STA** — Call proc `<prefix>_modal_edt_slow_capture`
- **Mode edt_fast_capture STA** — Call proc `<prefix>_modal_edt_fast_capture`

Note

 As with [SSN/SDC Constraints for Layout](#), if your design meets the configuration conditions described in the note in that section, you can run only the `<prefix>_modal_edt_slow_capture` proc to cover both the shift and `edt_slow_capture` modes simultaneously.

SSN/SDC Constraints for Hierarchical or Whole-Chip STA

The hierarchical STA flow with SSN stays the same as that described in the section “[Hierarchical STA in Tessent](#)” on page 752 for as long as you use extracted timing models (ETM or *.lib*) to represent the timing of child physical blocks (PBs). If your flow requires full or partial child PB instance netlists, you must replace your call to the usual logictest modal procs with their “*_with_sub_PBs” equivalent; that is, the following:

- `<prefix>_ltest_modal_shift_with_sub_PBs`
- `<prefix>_ltest_modal_edt_slow_capture_with_sub_PBs`
- `<prefix>_ltest_modal_edt_fast_capture_with_sub_PBs`

These procs include retargeted SDC constraints for all SSN logic inside all of your individual child PBs. Tessent-generated graybox models include all SSN logic. Without them, the parent SSN bus clock would enter through the block’s SSN bus pins and propagate directly to the


block's scannable logic. This, in turn, would overconstrain the child SSH-generated DFT signals. The following is an example of such a retargeted constraint:

```
array set tesseract_ssh_mapping_with_sub_PBs {
    ssh2 top_rtl2_tesseract_ssn_scan_host_1_inst
    ssh0 corea_i1/corea_rtl2_tesseract_ssn_scan_host_1_inst
    ssh1 corea_i2/corea_rtl2_tesseract_ssn_scan_host_1_inst
}
...
if {$mode eq "shift"} {
    set_case_analysis 1 [tesseract_get_pins \
        $tesseract_ssh_mapping_with_sub_PBs(ssh0)/tesseract_persistent_cell_scan_en_buf/Y]
    set_case_analysis 1 [tesseract_get_pins \
        $tesseract_ssh_mapping_with_sub_PBs(ssh1)/tesseract_persistent_cell_scan_en_buf/Y]
    set_case_analysis 1 [tesseract_get_pins \
        $tesseract_ssh_mapping_with_sub_PBs(ssh2)/tesseract_persistent_cell_scan_en_buf/Y]
}
```

When you run top-level logicstest STA with isolated lower-level PBs, your calling script must also call the proc `<prefix>_ltest_lower_pbs_external_mode`. This takes care of constraining the DFT signal and the OCC logic of the lower PBs. For example, to set the shift STA mode, you must run the following command sequence:

```
tesseract_set_default_variables
set_load_unload_timing_options <list of timing options>
# (or source the file containing this command)
<prefix>_ltest_modal_shift_with_sub_PBs
<prefix>_lower_pbs_external_mode
```

Note

 Using `*_with_sub_PBs` procs is subject to the following limitations:

- These procs create retargeted SSN/SSH SDC constraints for all instances of SSN/SSH modules in the child PBs; however, all PBs end up using the same set of timing parameter variables (such as the bus clock, the shift clock frequency, and the number of scan_en/edt_update extra setup/hold cycles) as the level of the current design. Child PBs may have been designed and laid out with different parameter sets, with potentially faster internal shift clocks than their parents. If this applies to any of your PBs, you must manually fix their associated constraints in your `extract_sdc` file.
- SSN/SSH constraints are declared for all PB instances, so you must load all of those instances to avoid errors in reading constraints.
- You cannot set child PBs to internal ltest mode only using the provided procs in the SDC file. If you need to do this, you can copy your `<prefix>_lower_pbs_external_mode` proc, change its name, and manually change the settings of the `int_ltest_en` and `ext_ltest_en` signals to put all child PBs into internal mode.

SSN/SSH SDC Constraint Descriptions

The following sections provide more detailed descriptions of SDC constraints for SSN and SSHs.

SSN Bus Clock and SSN Bus Port Timing Constraints	462
SSN Bus Data Port Delays	463
SSN Datapath Timing Constraints	465
SSN FIFO Timing Constraints	468
SSN ScanHost Timing Constraints	471

SSN Bus Clock and SSN Bus Port Timing Constraints

The SSN bus clock is declared using variables that contain a period in nanoseconds and a multiplier that aligns the units with your timing tool.

The following example shows the variables you use to declare the SSN bus clock:


```
global tessent_ssn_bus_clock_network_period
global time_unit_multiplier
set local_ssn_bus_clock_network_period \
    [expr $tessent_ssn_bus_clock_network_period * $time_unit_multiplier]
create_clock <port> -name tessent_ssn_bus_clock_network \
    -period $local_ssn_bus_clock_network_period -add
```

In this example, the `tessent_ssn_bus_clock_network_period` variable is always given in ns. The `time_unit_multiplier` variable converts the ns unit to the one used by the timing tool. For example, set a value of 1000 if the timing tool uses ps instead of ns. In the Tessent recommended flow, you set the bus clock network period variable indirectly with the following proc in your primary timing script:

```
set_load_unload_timing_options -usage ssn -ssn_bus_clock_period
```

The option values corresponds to the maximum possible SSN network speed across your entire design, even if your current level test patterns do not require that speed. Refer to the [set_load_unload_timing_options](#) command reference page for more information on using it in the SSN reference flow.

Note

 If your SSN scan host implementation uses only Streaming-Through-IJTAG, your generated SDC file does not contain this declaration. For more information about Streaming-Through-IJTAG, refer to “[Streaming-Through-IJTAG Scan Data](#)” on page 405.

SSN Bus Data Port Delays

The SSN data bus port external delays reflect the hard-coded timeplates in Tessent test patterns, relative to the `tessent_ssn_bus_clock_network` clock edges.

The SSN test patterns do the following:

- Force SSN bus data primary inputs at 0% of the tester period or 50% of the `bus_clock` period. Refer to the following discussion for further explanation.
- Cause the bus clock to rise at 0% of the tester period.
- Measure SSN bus data primary outputs at 0 ns into the next tester period.

This translates to an external delay of zero for both directions.

The first node of an SSN bus might capture its input data at every rising edge of the bus clock or on the first phase of a multi-phase node. This is the case for nodes of type Pipeline, ScanHost, or Multiplexer, or one of BusFrequencyMultiplier with its `capture_phase` property set to “transmitter”. For such nodes, the following is true:

- If the current design level is chip, the SSN patterns force the bus inputs at 50% of the bus clock period inside the tester period, thus giving a half `bus_clock_period` margin for both setup and hold.
- Otherwise, the patterns assume a synchronous data path across the block boundaries and force the bus inputs at 0% of the tester period.

For all other node types, SSN patterns force the bus inputs at 0% of the tester period.

The 0% output delay permits a full bus clock period to the setup margin of the bus data out loop timing path, as shown in the figure “[Schematic Showing Loop Timing Path for Bus Data Out](#)” in the section “[BusFrequencyDivider](#)” in the *Tessent Shell Reference Manual*.

The input/output delay constraints include an “`-add_delay`” option that enables them to share their primary pin with functional signals when you apply the non-modal constraints.

The following example illustrates the input/output delay constraints:

```
set_input_delay 0.0 -add_delay \
  -clock tessent_ssn_bus_clock_network \
  [tessent_get_ports {gpio[0]}]

set_output_delay 0.0 -add_delay \
  -clock tessent_ssn_bus_clock_network \
  [tessent_get_ports {gpio[16]}]
```

At the chip level, the tester directly feeds input/outputs. Therefore, input/output pin delays are based on the actual pattern waveforms, and no adjustments or virtual clocks are necessary. For a chip, constraints look like the following:

```
set_input_delay 0.0 -add_delay \  
-clock tessent_ssn_bus_clock_network \  
[tessent_get_ports {gpio[1]}]  
  
set_output_delay 0.0 -add_delay \  
-clock tessent_ssn_bus_clock_network \  
[tessent_get_ports {gpio[6]}]
```

For a block, in order to account for pre- and post-layout designs with varying `ssn_bus_clock` latency and in order to facilitate timing path budgeting, the proc declares a virtual clock and `set_input/output_delay` constraints use global user-modifiable Tcl variables, such as in the following example:


proc xxx_ltest_set_timing_variables_default:

```
set tessent_ssn_bus_input_delay_percentage 25.  
set tessent_ssn_bus_output_delay_percentage 25.
```

proc xxx_ltest_ssn:

```
set local_ssn_bus_clock_network_period \  
[expr $tessent_ssn_bus_clock_network_period * $time_unit_multiplier]  
set local_ssn_bus_input_delay  
[expr $tessent_ssn_bus_input_delay_percentage/100. * \  
$local_ssn_bus_clock_network_period]  
set local_ssn_bus_output_delay \  
[expr $tessent_ssn_bus_output_delay_percentage/100. * \  
$local_ssn_bus_clock_network_period]  
  
set_input_delay $local_ssn_bus_input_delay -add_delay \  
-clock tessent_ssn_virtual_bus_clock_network \  
[tessent_get_ports {ssn_bus_data_in[0]}]  
set_output_delay $local_ssn_bus_output_delay -add_delay \  
-clock tessent_ssn_virtual_bus_clock_network \  
[tessent_get_ports {ssn_bus_data_out[0]}]
```

Note

 To accurately reflect the generated test patterns, the following occurs at the chip level:

- If the first SSN datapath node is a `receiver_1x_pipeline` (which captures at the falling edge of the `ssn_bus_clock`), the preceding input delay is set to 0.0 ns.
 - If the first node is anything else (that is, captures at the rising edge of the `ssn_bus_clock`), the input delay is set to $0.5 \times \text{ssn_bus_clock_period}$.
-

For a core, the input delay is always set to $0.0 + \langle \text{an external delay timing budget} \rangle$, assuming the following:

The driving node exists inside the chip, either in the parent design or in another core.

SSN source nodes always toggle their output data on the `ssn_bus_clock` posedge, and full-speed data paths likely need balanced source/destination clocks.

If the datapath crosses a non-balanced clock, it does so through an SSN divider/multiplier combination. This combination is less sensitive to these input/output delay constraints, because it relaxes with additional multicycle path (MCP) constraints.

SSN Datapath Timing Constraints

Some sections of the SSN datapath may run with a bus data rate that is a fraction of the bus clock speed, such as at half- or quarter-speed, thus behaving as multicycle paths (MCPs) across SSN nodes. Such paths occur when inserting SSN nodes of the types `BusFrequencyDivider`, `BusFrequencyMultiplier`, or `OutputPipeline` in your datapath specifications.

For more details about these nodes and their usage, refer to the section “[BusFrequencyDivider](#)” in the *Tessent Shell Reference Manual*.

The three previously mentioned nodes (`BusFrequencyDivider`, `BusFrequencyMultiplier`, and `OutputPipeline`) intercept the data bus bits with rows of flops that capture or update all bus data bits at a specified phase of the bus clock. In the SDC file, the phase of each node is represented by two numbers “ $(n\ m)$ ” where m is the number of bus clock cycles for a single data cycle. For example, consider a quarter-speed SSN data bus section. The section may require MCPs between the following types of nodes:

- A `BusFrequencyDivider` node with a (1 4) phase

This means that the divider latches its output data at the beginning of the first (1) cycle for a duration of four (4) cycles.

- A `BusFrequencyMultiplier` node with a (3 4) phase

The specified phase number (3) represents the clock cycle number that the input data is latched at the beginning of. The output data rate of a multiplier always equals the clock rate.

- An `OutputPipeline` node with a (1 4) phase

The node features a single pipeline flop row that captures at the beginning of the first (1) cycle of four (4).

As another example, a bus datapath between a (1 4) divider and a (3 4) multiplier ends up with timing margins of two cycles of setup and two cycles of hold, making that combination suitable for data handoff between two skewed CTS regions:

```
# -----  
# From bus_frequency_divider (phase 1 4) to bus_frequency_multiplier  
# (phase 3 4)  
set_multicycle_path -setup 2 -start \  
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \  
    -to    [tessent_get_cells <multiplier node instance>/datapath/r0*]  
set_multicycle_path -hold 3 -start \  
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \  
    -to    [tessent_get_cells <multiplier node instance>/datapath/r0*]
```

A bus datapath between the previously referenced (1 4) divider node and a (1 4) output pipeline node features timing margins of four cycles of setup and zero cycles of hold, thus requiring balanced source and destination clocks:

```
# -----  
# From bus_frequency_divider (phase 1 4) to output_pipeline (phase 1 4)  
set_multicycle_path -setup 4 -start \  
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \  
    -to    [tessent_get_cells <multiplier node instance>/datapath/r0*]  
set_multicycle_path -hold 3 -start \  
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \  
    -to    [tessent_get_cells <multiplier node instance>/datapath/r0*]
```

Datapaths from SSN bus output ports of a (1 4) OutputPipeline node also feature four cycles of setup margin, as shown in the figure “[Output Data Slow Down Using BusFrequencyDivider Node](#)” in the section “[BusFrequencyDivider](#)” in the *Tessent Shell Reference Manual*, because the test patterns capture the bus data at the rising edge of Phase 1, leaving all four bus cycles for data to close its output timing loop:

```
# -----  
# From output_pipeline (phase 1 4) to output bus ports  
set_multicycle_path -setup 4 -start \  
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \  
    -to    [tessent_get_ports <list of output bus ports>]  
set_multicycle_path -hold 3 -start \  
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \  
    -to    [tessent_get_ports <list of output bus ports>]
```

More subtle MCP constraints are required when handling datapath sections that run across bus clocks that operate at frequencies that are ratios of each other. This could result in, for example, a (1 2) divider fanning out to a (3 4) multiplier, which implies that the destination multiplier is clocked at double the frequency of the source divider. In such cases, the MCP constraint always

sets its setup and hold number of cycles relative to the faster of the two clocks, using either the `-start` or the `-end` option:

```
# -----
# From bus_frequency_divider (phase 1 2) to bus_frequency_multiplier
# (phase 3 4)
set_multicycle_path -setup 2 -end \
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \
    -to   [tessent_get_ports <multiplier node instance>/datapath/r0*]
set_multicycle_path -hold 2 -end \
    -from [tessent_get_cells <divider node instance>/datapath/r1*] \
    -to   [tessent_get_ports <multiplier node instance>/datapath/r0*]
```

Input and output pins of child physical blocks may also include phases if they internally connect to either SSN multiplier or divider nodes. These phases are reported in the module description of their `.icl` file. Because the internal cells of a physical block are not normally visible to the parent SDC, you must set MCP constraints using the `-through` switch with the PB boundary pins, as in the following example:

```
# -----
# From PB output pins (phase 1 2) to PB input pins (phase 0.5 1)
set_multicycle_path -setup 1 -start \
    -through [tessent_get_pins <PB1 list of bus data output pins> \
    -through [tessent_get_pins <PB2 list of bus data input pins>]
set_multicycle_path -setup 1 -start \
    -through [tessent_get_pins <PB1 list of bus data output pins> \
    -through [tessent_get_pins <PB2 list of bus data input pins>]
```

The `extract_sdc` command itself actively traces the `.icl` files of the design to find which actual SSN node is connected to which other SSN nodes and whether phase specifications are involved. Such tracing enables handling of bus networks with multiplexer nodes and connections between partial lists of SSN bus ports or subphysical block pins, which can further complicate the list of MCPs. For the sake of clarity and self-documentation, the SDC file reports the traced list of connections under a comment banner labeled “SSN list of node connections.”

This banner shows all connections, including those not involving clock phases, as in the following example:

```
#-----  
# SSN list of node connections  
#  output_pipeline 'top_rtl2_tessent_ssn_out_pipe_out_inst'  
#    --> SSN Bus Data Outputs  
#  bus_frequency_divider 'top_rtl2_tessent_ssn_bus_freq_div_1_inst'  
#    --> output_pipeline 'top_rtl2_tessent_ssn_out_pipe_out_inst'  
#  scan_host 'top_rtl2_tessent_ssn_scan_host_1_inst'  
#    --> bus_frequency_divider 'top_rtl2_tessent_ssn_bus_freq_div_1_inst'  
#  pipeline 'top_rtl2_tessent_ssn_pipe_2_inst'  
#    --> scan_host 'top_rtl2_tessent_ssn_scan_host_1_inst'  
#  bus_frequency_multiplier 'top_rtl2_tessent_ssn_bus_freq_mult_1_inst'  
#    --> pipeline 'top_rtl2_tessent_ssn_pipe_2_inst'  
#  physical_block 'corea_i1' (launching phase = 1/2)  
#    --> bus_frequency_multiplier 'top_rtl2_tessent_ssn_bus_freq_mult_1_inst'  
#  physical_block 'corea_i2' (launching phase = 1/2)  
#    --> physical_block 'corea_i1' (strobing phase = 0.5/1)  
#  pipeline 'top_rtl2_tessent_ssn_pipe_1_inst'  
#    --> physical_block 'corea_i2' (strobing phase = 0.5/1)  
#  SSN Bus Data Inputs  
#    --> pipeline 'top_rtl2_tessent_ssn_pipe_1_inst'  
#-----
```

SSN FIFO Timing Constraints

An SSN FIFO advantageously replaces a combination of BusFrequencyDivider and BusFrequencyMultiplier nodes in cases where the clock domain handoff can occur within a single physical partition.

For a full description of the FIFO circuit and node definition usage, refer to the section “[Fifo](#)” in the *Tessent Shell Reference Manual*.

A Fifo node includes an input register for storing input packets, similar to that of a BusFrequencyDivider, and it selects the output packet based on the phase counter value, similar to the operation of a BusFrequencyMultiplier. Similar to datapaths between BusFrequencyDivider and BusFrequencyMultiplier explained in the section “[SSN Datapath Timing Constraints](#)” on page 465, a set_multicycle_path declaration can relax FIFO internal paths from the input register. The setup and hold parameters for this declaration depend the following property values inside the DftSpecification/SSN/Datapath/Fifo wrapper:

- frequency_ratio
- in_clock_to_out_clock
- in_clock_to_out_clock_skew_programmable

You can think of the last two properties as defining a “deskewing” mode.

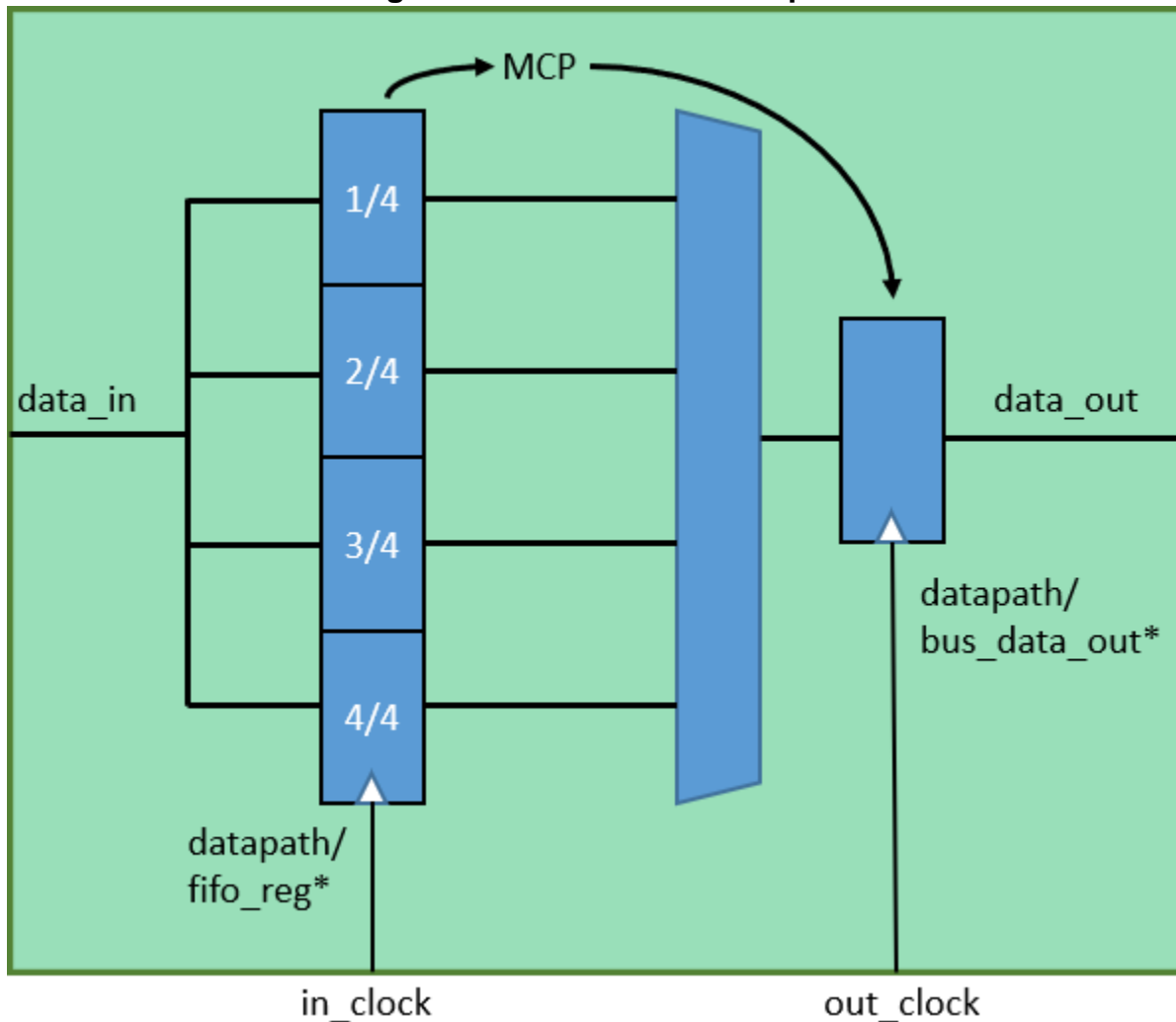
Data output from the FIFO input register remains stable for a number of clock cycles equal to the value of the FIFO frequency_ratio property. The FIFO output register captures this data

either in the middle of these cycles (when `in_clock_to_out_clock_skew` is `early_or_delayed`) or at the end of these cycles (when `in_clock_to_out_clock_skew` is `delayed_only`). Refer to the figures “[Waveform for FIFO early_or_delayed Configuration](#)” and “[Waveform for FIFO delayed_only Configuration](#)” in the “`Fifo`” reference page in the *Tessent Shell Reference Manual* to see examples of these capture cycles. Because of this, your SDC FIFO multicycle path (MCP) constraint “-setup” parameter is set equal to the FIFO `frequency_ratio` property, and its “-hold” parameter adjusts depending on its `in_clock_to_out_clock_skew` property value. The setup and hold calculations occur explicitly using Tcl variables in your SDC.

If you set `in_clock_to_out_clock_skew_programmable` to “on”, the SDC enables you to dynamically select the deskewing mode just before applying the SDC in the timing tool. For each such FIFO instance in your design, select the deskewing mode by defining the Tcl variable `tessent_ssn_fifo_deskew_setting(sfifo<n>)` in your primary script. This variable name is mapped; refer to the primary calling script in the “[Examples](#)” section in this topic to see how to use these variables. The `in_clock_to_out_clock_skew` property determines the default deskew mode when you do not set these variables.

The following figure illustrates how the MCP is applied in a FIFO.

Figure 8-24. FIFO MCP Example



Examples

Example of FIFO Settings in Your Primary Calling Script

The following example shows how to use Tcl variables in the primary calling script to set up the FIFO and control programmable skew settings:

```
tessent_set_default_variables
tessent_ssn_fifo_deskew_setting(sfifo0)    delayed_only
tessent_ssn_fifo_deskew_setting(sfifo1)    early_or_delayed
tessent_ssn_fifo_deskew_setting(sfifo2)    delayed_only
tessent_set_non_modal
# this proc calls the tessent_set_ltest_ssn proc, which appears in the
# next example section
```

Examples of FIFO SDC File Contents

The following is an example of content appearing in the SDC proc `tessent_set_default_variables`:

```
array set tessent_sfifo_mapping {
    sfifo0 top_rtl2_tessent_ssn_fifo_1_inst
    sfifo1 core1/core_rtl2_tessent_ssn_fifo_1_inst
    sfifo2 core2/core_rtl2_tessent_ssn_fifo_1_inst
}
```


The following is an example of content appearing in the SDC proc `tessent_set_ltest_ssn`:

```
global tessent_ssn_fifo_deskew_setting
array set sfifo_setup_multiplier {delayed_only 1 early_or_delayed 0.5}
# sfifo0:
# frequency_ratio : 4
# in_clock_to_out_clock_skew: delayed_only
# in_clock_to_out_clock_skew_programmable: on
set deskew_setting delayed_only


if {[info exists tessent_ssn_fifo_deskew_setting(sfifo0)]} {
    set deskew_setting $tessent_ssn_fifo_deskew_setting(sfifo0)
}

set frequency_ratio 4
set_multicycle_path \
    -setup [expr int($frequency_ratio * $sfifo_setup_multiplier($deskew_setting))] \
    -from [tessent_get_cells $tessent_sfifo_mapping(sfifo0)/datapath/fifo_reg*] \
    -to [tessent_get_cells $tessent_sfifo_mapping(sfifo0)/datapath/bus_data_out*]
set_multicycle_path \
    -hold [expr $frequency_ratio - 1] \
    -from [tessent_get_cells $tessent_sfifo_mapping(sfifo0)/datapath/fifo_reg*] \
    -to [tessent_get_cells $tessent_sfifo_mapping(sfifo0)/datapath/bus_data_out*]
```

Note

 The preceding constraints repeat for `sfifo1` and `sfifo2`, but with variations for the actual MCP constraints that depend on the `DftSpecification` property settings for these FIFOs.

Note

 The section in bold in the previous example is present only when programmable skew is enabled; that is, `in_clock_to_out_clock_skew_programmable` is set to “on”.

SSN ScanHost Timing Constraints

The ScanHost node generates the scan control and data signals when it drives the logic test modules of a block. It generates all of these signals from the SSN `bus_clock`.

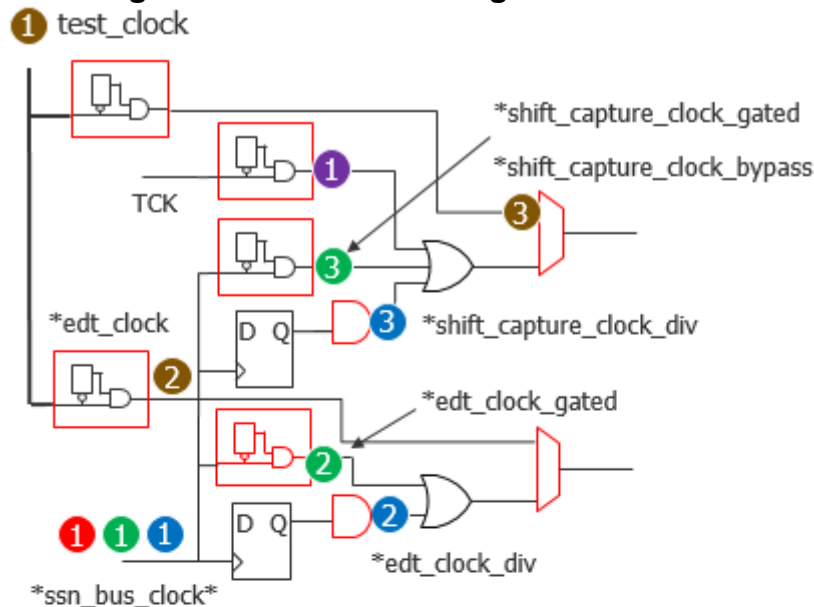
It is recommended that you clearly understand the SSH scan interface operation and features described in the section “[ScanHost Scan Interface Operation and Timing Requirements](#)” of the

[ScanHost](#) reference page in the *Tessent Shell Reference Manual*, as a prerequisite for the SSH SDC descriptions in this section.

SSH Scan Clocks

A given Tessent test SSN scan pattern may drive the SSN bus clock port with one of three different frequencies. The frequency depends on whether the ScanHost is bypassed or used for scan with gated or with divided shift clocks. To precisely determine the timing that the hardware implements, the SDC needs to create three clocks on that port. The following figure shows all root clocks and generated clocks that may exist in the SDC.

Figure 8-25. SSH Clock Signal Generation



Notes

- Persistent cells are shown in red
- Bypass mode clocks (in brown) are optional
- *ssn_bus_clock* defined on input ports
- All generated clocks defined on pins of persistent cells
- All clock names prefixed with "tessent_" in the script
- All SSH gen clock names prefixed with tessent_ssh<n>
- Clocks of different colors are physically exclusive.

SSH Clocks Relations and Speeds

- ① tessent_ssn_bus_clock_network
- runs at maximum possible bus speed
- ① tessent_ssn_bus_clock_scan_slow
- source for ssh gated clocks
- runs at shift_clock speed
- ① tessent_ssn_bus_clock_scan_fast
- source for ssh divided clocks.
- period = shift_clock_period/<n>
- ② ③ divided_by_1 versions of master clock ①
- ② ③ divided_by_<n> versions of master clock ①
- ② ③ Divided_by_1 version of master clock ①
- ① Set_disable_timing prevents tck from propagating to scan domains

Call the [set_load_unload_timing_options](#) command to configure the SDC clock and control signal timing parameters, and to configure pattern generation so that the parameters of the generated patterns are consistent with those used during timing closure and analysis. The

following table lists `set_load_unload_timing_options` arguments that define Tcl variables used in SDC. If your primarytiming script does not explicitly set these options, then they use the default values from the table.

Table 8-6. `set_load_unload_timing_options` Arguments and SDC Variables

Option	SDC Timing Variable	Default
<code>-ssn_bus_clock_period</code>	<code>tessent_ssn_bus_clock_network_period</code>	2.5 ns
<code>-shift_clock_period</code>	<code>tessent_ssn_bus_clock_scan_slow_period</code>	10.0 ns
<code>-scan_en_setup_extra_cycles</code>	<code>tessent_scan_en_setup_extra_cycles</code>	1
<code>-scan_en_hold_extra_cycles</code>	<code>tessent_scan_en_hold_extra_cycles</code>	1
<code>-edt_update_setup_extra_cycles</code>	<code>tessent_edt_update_setup_extra_cycles</code>	1
<code>-edt_update_hold_extra_cycles</code>	<code>tessent_edt_update_hold_extra_cycles</code>	1

The first SSN bus clock is called `ssn_bus_clock_network` (red dot in [Figure 8-25](#)). It times the SSN network operation at its maximum possible datapath speed, whether or not the ScanHost node is active.

When the ScanHost node is active, the SSN bus clock pin generates the `shift_capture_clock` and the `edt_clock` used to operate the scan circuitry. One scan bit per chain gets shifted in per SSN packet. If the SSN packet is small enough to occupy less than two bus clock cycles on the network, the scan clocks can be generated from the SSN bus clock using a clock gater. The test patterns then adjust the SSN bus clock frequency to match the required `shift_clock_period`. However, when the SSN packet is large enough to occupy a minimum of N bus clock cycles on the network, a faster frequency clock is applied to the `ssn_bus_clock` pin and is divided to generate the shift clocks using a clock divider flop, maintaining as much as possible a fifty-percent duty cycle. Because these two clock paths differ slightly, both must be covered in SDC and STA, requiring the SDC to create one generated clock for the gated clock and another for the divided clock.

When the test patterns generate the scan clocks using a clock gater, they must increase the SSN bus clock period to be as large as the shift clock period, which defaults to 10 ns. The SDC version of the SSN bus clock created at the `shift_clock_period` is called `ssn_bus_clock_scan_slow` (green dot labeled 1 in [Figure 8-25](#)) and is referenced as the source of the generated clocks `ssn_shift_capture_clock_gated` and `ssn_edt_clock_gated`, which are defined on the outputs of the `edt_clock_cg` and `shift_capture_clock_cg` persistent cells (green dots 2 and 3 in [Figure 8-25](#)).

When the test patterns generate the scan clocks using a clock divider, another SDC clock called `ssn_bus_clock_scan_fast` is required (blue dot 1 in [Figure 8-25](#)) and becomes the `-master` option of the divided by N generated clocks called `ssn_shift_capture_clock_div` and `ssn_edt_clock_div`, which are defined on the output of the `edt_clock_div_and` and `shift_capture_clock_div_and` cells (blue dots 2 and 3 in [Figure 8-25](#)).

Regardless of the size of the SSN packets in your final design, the `ssn_bus_clock_scan_fast` clock frequency in SDC must be set as the largest possible integer multiple of your specified shift clock period, so that the SDC can use it in a simple “`create_generated_clock -divided_by <n>`” constraint without exceeding the maximum `ssn_bus_clock_network` clock frequency. A simple case of this is when `shift_clock_period/ssn_bus_clock_period` is exactly an even integer. Then the period of the `ssn_bus_clock_scan_fast` clock is exactly equal to the period of the `ssn_bus_clock_network` clock. However, consider an example where the `ssn_bus_clock_period` is 2.5 ns and the `shift_clock_period` is 12 ns. Then the period of `ssn_bus_clock_fast` must be set to 3 ns, so that $3 \text{ ns} \times 4 = 12 \text{ ns}$. This accurately reflects the way the SSN scan test patterns themselves also slow the bus clock down to 3 ns if given the same timing specifications and if the packet size is four bus_clock cycles.

If the ratio of your `shift_clock` period to your `ssn_bus_clock` period is larger than two, the SDC creates the SSH divided scan clock with a period equal to the period of the SSH gated scan clocks. However, if the ratio is less than two, the created divided clock is slower than the gated clock, as it would be whenever your SSN scan packet is larger than two clock cycles.

Even more clock declarations are required if the SSH supports the optional scan bypass mode. These clocks are represented by the brown dots in [Figure 8-25](#). Both the `*edt_clock` and `*ssh_shift_capture_clock` are `divided_by 1` versions of the `test_clock`. For simplification of the constraints, the generated SDC file assumes a `test_clock` period that always matches the value specified by the “`set_load_unload_timing_options -shift_clock_period`” command, even though the latter value ideally applies only to the SSN scan mode and not the bypass mode.

The preceding clocks belong to clock groups that correspond to their dot colors in [Figure 8-25](#). Red, blue, green, and brown clocks never run at the same time. Therefore, the SDC declares them as “`-physically_exclusive`.” All such clocks are also asynchronous to all functional domain

clocks, because both clock types may interact at the same time on the pre-OCC branches during scan. The following is an excerpt from a sample design:

Figure 8-26. SSH Clock Group Constraints

```
# SSN is run either with fast or slow clock exclusively
set_clock_groups -physically_exclusive \
    -group {tessent_ssn_bus_clock_scan_fast* tessent_ssh*_div*} \
    -group {tessent_ssn_bus_clock_scan_slow* tessent_ssh*_gated*} \
    -group {tessent_ssn_bus_clock_network*}

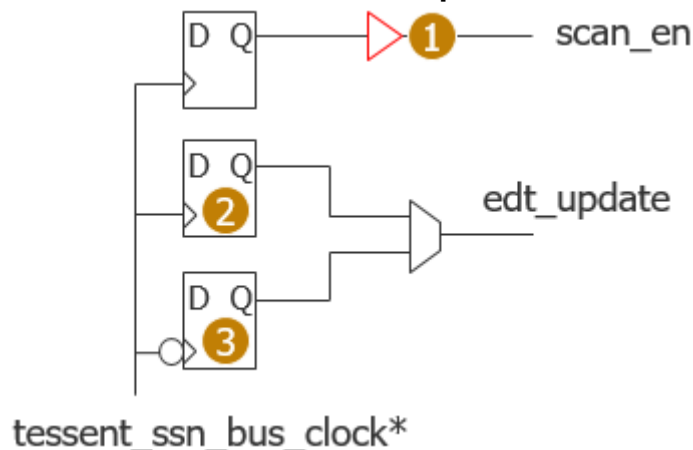
# SSH internal mode and bypass ltest modes are never run together at the same time.
set_clock_groups -physically_exclusive \
    -group {tessent_ssn* tessent_ssh*_div tessent_ssh*_gated} \
    -group {tessent_ssh*_bypass tessent_virtual_slow_clock \
        tessent_edt_clock tessent_test_clock}

# SSN/Scan clocks are asynchronous to all other clocks.
set_clock_groups -asynchronous -name scan_clocks \
    -group {tessent_ssn* tessent_ssh* tessent_virtual_slow_clock \
        tessent_edt_clock tessent_test_clock}
```

scan_en and edt_update Signal Timing

The following figure shows the SSH circuit that generates the scan_en and edt_update signals.

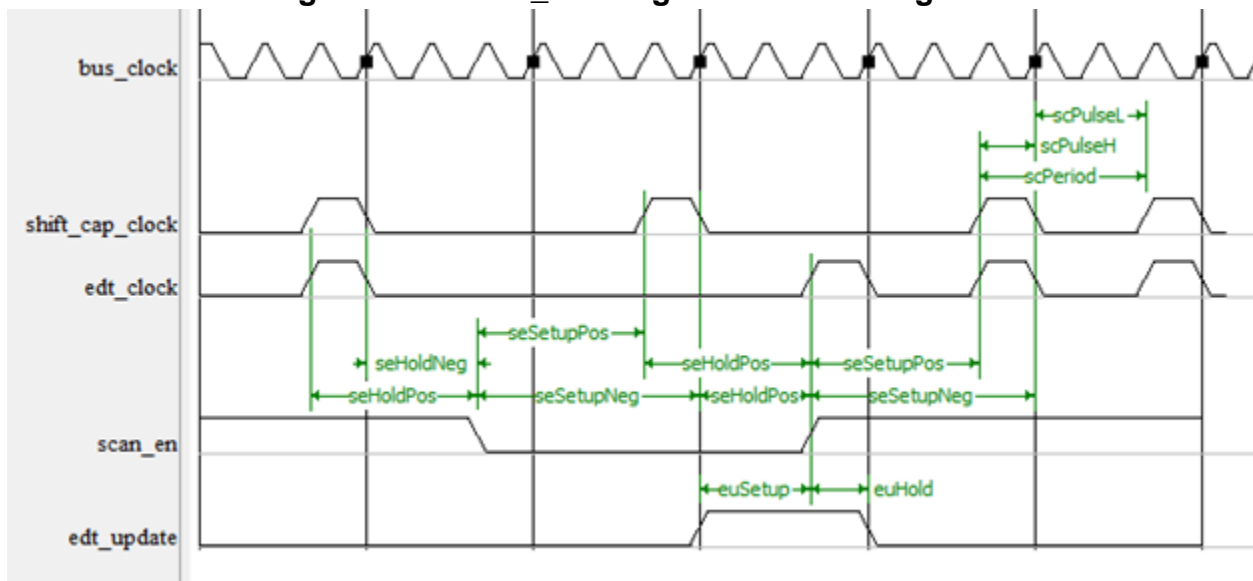
Figure 8-27. SSH scan_en and edt_update Generation Circuit



The SSH generates the `edt_update` signal with a negedge flop (brown dot 3) when `bus_clock_period = scan_clock_period` to provide negedge retiming to its destination EDT controller's posedge flops. The posedge flops run off the late `edt_clock` relative to the `bus_clock`. When `edt_clock` is divided, on the other hand, a posedge source flop (brown dot 2) can properly toggle the `edt_update` signal 0.5 `edt_clock` cycles before and after the posedge of the destination flop.

Although the scan_en signal may propagate to either posedge or negedge scannable flops on a delayed SCC clock tree, the fact that SCC is gated while scan_en toggles makes both its setup and hold timing safer without needing to rely on a negedge flop source like for edt_update. Furthermore, the scan_en setup margin to negedge flops gets a half-cycle bonus, because the first SCC falling edge after a scan_en transition always occurs after the first rising edge. The following figure shows the margin definitions.

Figure 8-28. scan_en Margin Definition Diagram



By default, although these signals show some very safe setup and timing margins, you can still extend them by calling the [set_load_unload_timing_options](#) command. As discussed previously, this proc sets several global Tcl variables used in timing exceptions discussed in the following information.

In the Tessent SDC file, the hold and setup margins of all MCP constraints are derived dynamically from the variable values in [Table 8-6](#) and from the bus_clock/scan_clock frequency ratios. Non-even bus/scan clock ratios, such as 3 or 5, add more complexity for divided clock constraints, because they mean that the scan clock duty cycle can no longer be 50%, which affects the number of bus clock cycles between its rising and falling edges. To account for this, the SDC file derives Tcl variables such as \$lowPulseCycles and \$highPulseCycles and uses them in the divided clock MCPs, at the price of added complexity. The following excerpt from an SDC file shows how all setup and hold margins are calculated. In this excerpt, the Tcl variable names intentionally match the setup and hold margin names shown in [Figure 8-28](#).

```

set freq_ratio [expr int($local_ssn_bus_clock_scan_slow_period\
    $tessent_ssn_bus_clock_scan_fast_period)]
set half_ratio [expr $freq_ratio/2]
# Divided clock duty cycle isn't 50% if freq_ratio is odd.
set highPulseCycles [expr $half_ratio]
set lowPulseCycles [expr $freq_ratio - $highPulseCycles]
# Scan_en to gated scan clocks:
set seSetupPos [expr 1 + $tessent_scan_en_setup_extra_cycles]
set seHoldPos [expr 1 + $tessent_scan_en_hold_extra_cycles]
set seSetupNeg [expr 1 + $seSetupPos]
set seHoldNeg $tessent_scan_en_hold_extra_cycles
# Scan_en to divided scan clocks:
set seSetupPos_div [expr $seSetupPos * $freq_ratio]
set seHoldPos_div [expr $seHoldPos * $freq_ratio]
set seSetupNeg_div [expr $seSetupPos_div + $highPulseCycles]
set seHoldNeg_div [expr $seHoldPos_div - $highPulseCycles]
# edt_update to gated scan clocks:
set setup [expr 1 + $tessent_edt_update_setup_extra_cycles]
set hold $tessent_edt_update_hold_extra_cycles
# edt_update to divided scan clocks:
set setup [expr $lowPulseCycles + \
    $tessent_edt_update_setup_extra_cycles*$freq_ratio]
set hold [expr $highPulseCycles + \
    $tessent_edt_update_hold_extra_cycles*$freq_ratio]

```

The following is an example of how the preceding \$setup and \$hold Tcl variables are typically used in the SDC file MCP constraints:

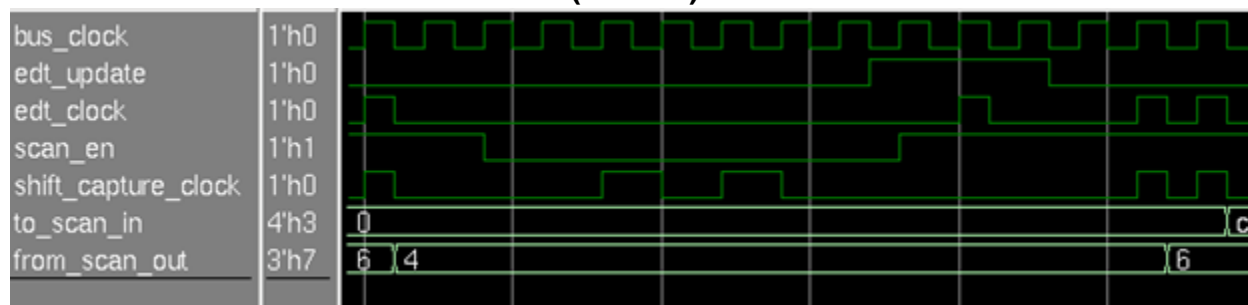
```

set_multicycle_path -setup $setup -start \
    -from [tessent_get_cells $tessent_ssh_mapping(ssh0)/clock_gen/edt_update_delayed*] \
    -to [tessent_get_clocks tessent_ssh*_div]
set_multicycle_path -hold [expr $setup-1 + $hold] -start \
    -from [tessent_get_cells $tessent_ssh_mapping(ssh0)/clock_gen/edt_update_delayed*] \
    -to [tessent_get_clocks tessent_ssh*_div]

```

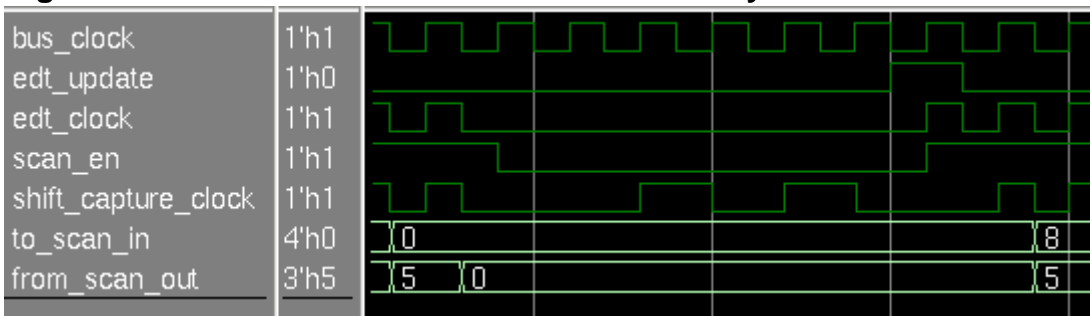
To help you visualize these constraints, the following figures show some actual simulation results of the timing of the scan_en and edt_update signals relative to the SSN bus and SSH scan clocks:

Figure 8-29. Gated Scan Clocks With All *extra_cycle* Variables Set to 1 (Default)



The following figure is the same as the previous, but the *extra_cycle* variables are set to zero. This results in shorter setup and hold margins.

Figure 8-30. Gated Scan Clocks With All *extra_cycle* Variables Set to 0



In this figure, due to some inner SSH logic requirements, the scan_en setup and hold paths get some bonus margins over the gated clocks case.

Figure 8-31. Divided Shift Clocks With Ratio of 4 and *extra_cycle* Variables at 0

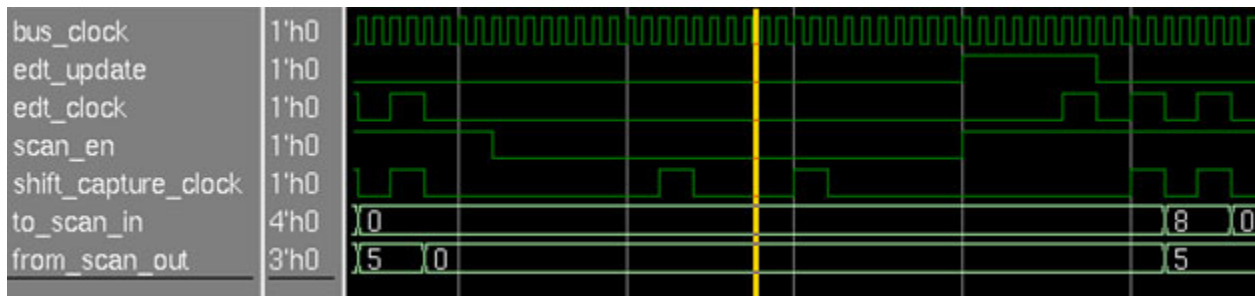
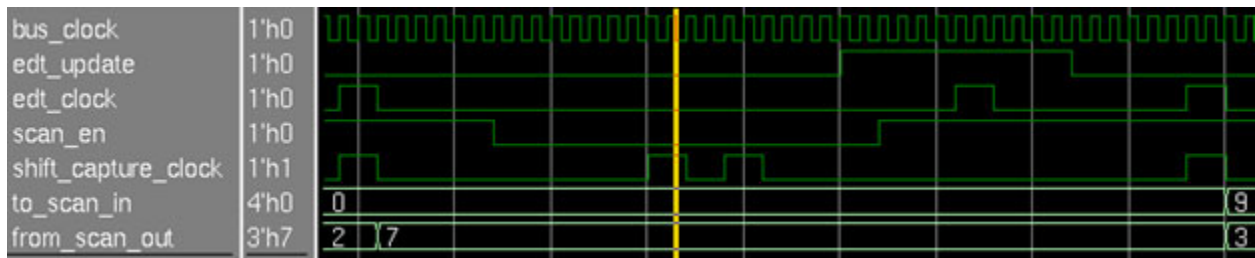


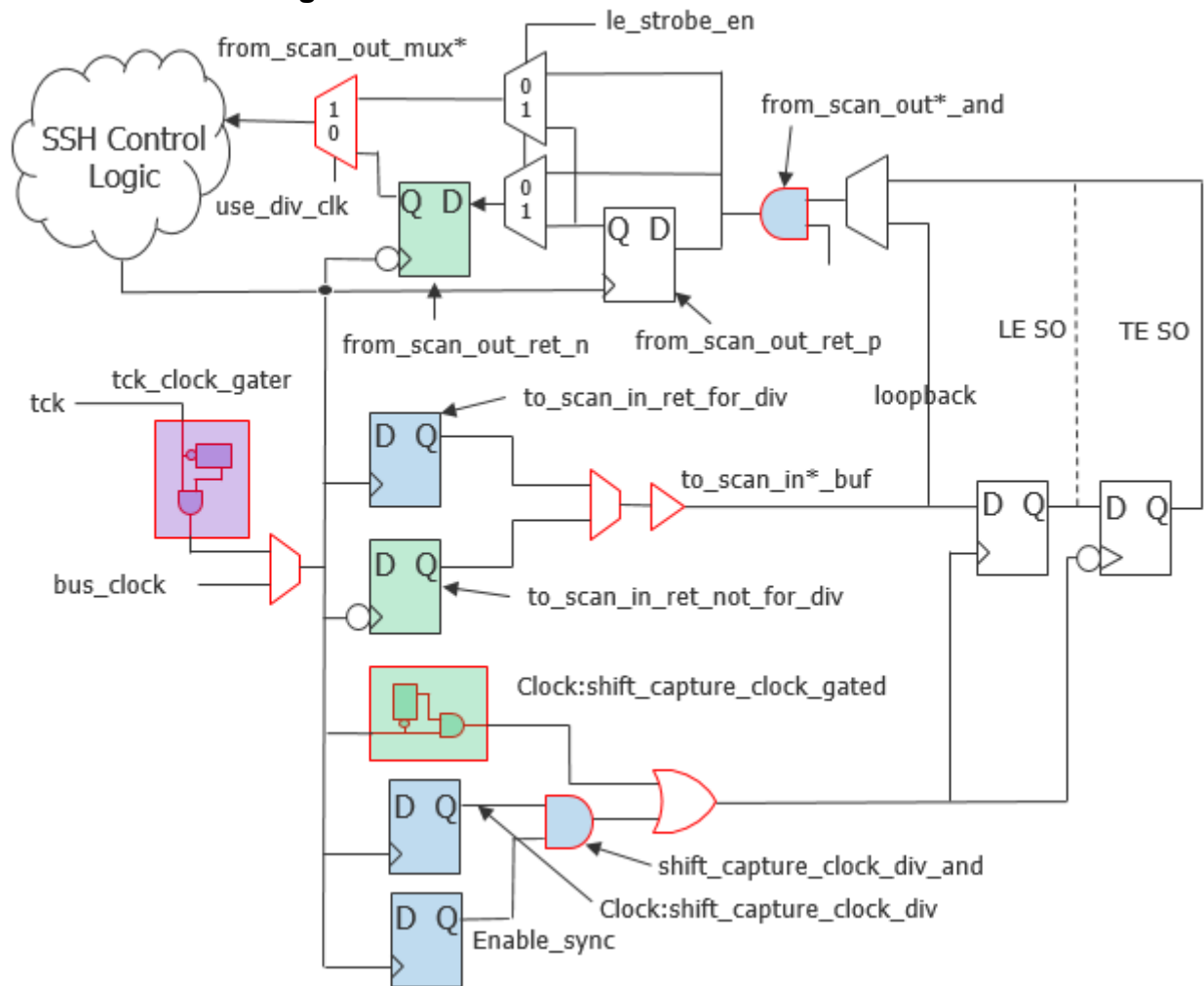
Figure 8-32. Divided Shift Clocks With Ratio of 4 and *extra_cycle* Variables at 1



Scan Chain Interface Timing

The SSH scan chain interface circuit appears in the following figure. Given the many different clocks and the two operation modes involved, this circuit requires many timing exceptions of different natures, such as set_disable_timing, set_false_path, and set_multicycle_paths.

Figure 8-33. SSH Scan Chain Interface Circuit



The following describes how this circuit operates:

- Persistent cells appear with red borders.
- All clock names appear without the “tessent_” prefix.
- The two flops on the right-hand side represent the scan chains. Their SI input comes from the SSH to_scan_in* pins, and their SO outputs go back to the SSH through the from_scan_out* pins.
- Direct loopback paths exist in some modes between the SSH to_scan_in* and from_scan_out* pins.
- Logic elements that are active only during scan with gated shift clocks are colored green.
- Logic elements that are active only during scan with divided shift clocks are colored blue.

- The from_scan_out*_and (blue) cell strobes the SO data of the returning chain when running with divided shift clocks, at the last of the N cycles of the divided clock. It is transparent when running with the gated clock.
- Chains may feature either a trailing edge (TE) or strobing edge (SE) chain terminating SO flop.
 - When running with gated shift clocks, the from_scan_out_ret_n (green) flop captures TE SO data.
 - With both gated and divided clocks, the from_scan_out_ret_p flop captures the LE SO data.
 - With divided clocks, the SSH control logic directly captures TE SO data through the from_scan_out_mux.
- The enable_sync (blue) signal is static during test and must be prevented from reaching the shift_capture_clock and edt_clock generation circuit.
- The tck_clock_gater (purple) cell feeds a gated version of tessent_tck to the SSH logic when it runs in Streaming-Through-IJTAG mode.

Scan Chain Interface Timing Exceptions

The following two tables summarize the SDC's false_path and set_multicycle_path declarations with regard to the preceding SSH chain interface circuit. For improved clarity, all clock names appear without the "tessent_" prefix.

Table 8-7. SSH Chain Interface Circuit SDC False Paths

Source	Destination	Reason
to_scan_in_ret_not_for_div	ssh*_div ssn_bus_clock_network ssn_bus_clock_fast	Green logic only active with slow clocks
to_scan_in_ret_for_div	ssh*_gated	Blue logic only active with fast clocks
ssh*_gated	-through from_scan_out_mux/ <out>	With slow clocks, all chain SOs are captured by a single flop and do not reach the rest of the SSH logic
-to/-from ssh*_div, ssn_bus_clock_network, ssn_bus_clock_scan_fast	-from/-to from_scan_out*_ret_n	This negedge flop only captures TE SO with slow clocks and is inactive with fast clocks
-fall_from ssh*_gated	from_scan_out*_ret_p	This flop only captures chain LE (rising edge) SO flops
-rise_from ssh*_gated	from_scan_out*_ret_n	This flop only captures chain TE (falling edge) SO flops.

Table 8-7. SSH Chain Interface Circuit SDC False Paths (cont.)

Source	Destination	Reason
-from virtual_force_pi	-through to_scan_in*_buf/ <out> -through from_scan_out*_and/<out> -to virtual_measure_po	Ignore stray paths from edt_channel_in* to edt_channel_out* through SSH scan loopbacks when bypass mode is present
-from flop enable_sync	-to ssh*_gated, ssh*_div	enable_sync signal is static during test but fans out to scan logic through the scan_en and edt_update signals
-from/-to ssh_bus_clock_scan_fast, ssn_bus_clock_network	-to/-from SSH logic flop last_in_bits_in_current_bus_word_ret (not shown in preceding figure)	SSH negedge flops are active only when using slow clocks.

Table 8-8. SSH Chain Interface Circuit SDC Multicycle Paths

Source	MCP	Destination	Reason
flop to_scan_in_ret_for_div	s:N/2 h:N/2	-start -to Clock ssh*_div	Adjust setup/hold to chain SI LE flop.
Clock ssh*_div	s:N h:0	-through from_scan_out*_and/ <out> -to ssn_bus_clock_scan_fast -end	Strobed TE SO flop data is captured directly by SSH control logic
Clocks ssn_bus_clock_scan_fast * ssn_bus_clock_network	s:N h:0	-through to_scan_in*_buf/<out> -through from_scan_out*_and/ <out> -to clocks <same as -from>	Scan loopback paths are MCPs of ssn_bus fast clocks

Additional SSH Scan Chain Interface Constraints

- For the sake of simplifying the SSH constraints, block tessent_tck from propagating to SSH logic, because it is redundant with the SSN bus clock, which imposes tighter constraints:

set_disable_timing <ssh_instance>/*tck_clock_gater/GCK

- Prevent the timing tool from finding a stray clock path through the enable_sync source flop CLK-to-Q timing arc in the fan-in of generated clock definitions on that same AND gate's output pin:

**set_disable_timing <ssh_instance>/clock_gen/*edt_clock_div_and/A1
set_disable_timing <ssh_instance>/clock_gen/*shift_capture_clock_div_and/A1**

- Because all SSH clock multiplexers switch clocks statically at the beginning or end of the test, avoid irrelevant noisy warnings from timing tools such as Primetime PTE-060:

`set_disable_clock_gating_check <ssh_instance>/*clock_mux`

Tessent SSN Examples and Solutions

This section contains Tessent solutions for DFT scenarios and problems specific to SSN. This includes applications and flows that solve issues resulting from the presence of specific design objects or the use of specific implementation flows.

The solutions are organized in the following sections:

Third-Party OCCs With SSN 484

Third-Party OCCs With SSN

Third-party OCC connections require specific handling when you are using SSN.

Problem

Third party OCCs need connections to `scan_en` and `shift_capture_clock`, which do not exist until the SSN [ScanHost](#) node is inserted. This section explains how to automate these connections.

Solution

The third-party OCCs are assumed to be pre-inserted into the design when SSN is inserted as described in the section [“Second DFT Insertion Pass: Inserting Block-Level EDT, OCC, and SSN”](#) on page 357. The `scan_en` and `slow_clock` pins on the OCC instances are tied low.

Follow these steps to properly connect the OCC to the ScanHost node:

1. Before calling the `check_design_rules` command, run the `“add_dft_control_points -type dynamic_dft_signal -dft_signal_name scan_en”` command while pointing to the `scan_en` pin of each OCC instance.

```
set occs [get_instances my_occ*]
add_dft_control_points [get_pins scan_en -of_inst $occ] \
    -type dynamic_dft_control \
    -dft_signal_source_name scan_en
```

2. Run the `“add_dft_control_points -type dynamic_dft_signal -dft_signal_name shift_capture_clock”` command while pointing to the `slow_clock` pin of each OCC instance:

```
add_dft_control_points [get_pins slow_clock -of_inst $occ] \
    -type dynamic_dft_control \
    -dft_signal_source_name shift_capture_clock
```

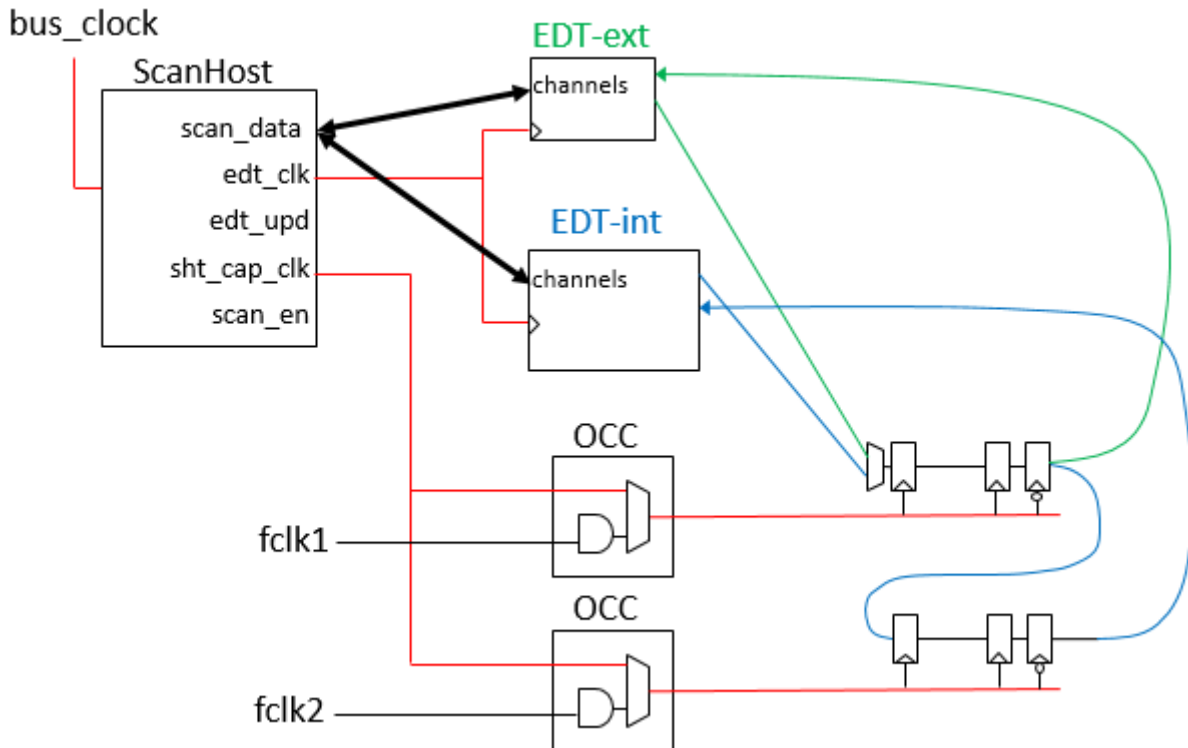
Discussion

In the solution described in the previous subsection, you use the `add_dft_control_points` command to make the `scan_en` and `shift_capture_clock` connections to the OCC. This is the

general command you use to connect any DFT signal to any destination. When you insert the [ScanHost](#) node, the insertion process remaps dynamic DFT signals `scan_en`, `edt_update`, `edt_clock`, and `shift_capture_clock` to the output pins of the ScanHost node.

Because the ScanHost node delivers the scan data in both the internal and external scan modes, your third-party OCC must still be able to inject the `slow_clock` into the clock_domains when `scan_en` is high, even when the OCCs are not active. In a wrapped core, you have an OCC on each clock input. Those OCCs are active during the internal scan modes but are typically inactive during the external test modes, such that the sourcing for the capture clock is from a single OCC localized at the functional clock source of the domain. Those inactive OCCs must, however, still inject the `shift_capture_clock` generated by the local ScanHost node when `scan_en` is high during the external modes. This is the [shift_only_mode](#) feature described in the Tessent OCC reference page. The presence of this feature enables easier timing closure because the entire shift timing is internal to the physical block and completely derived from the SSN bus clock, as illustrated in the following figure:

Figure 8-34. Localized Scan Timing With Shift-Only Mode OCC



If you want your legacy scan access mechanism to coexist with SSN in your first few designs that use SSN, add the `scan_en` and other dynamic DFT signals on their original sources as you do currently. Then, continue to preconnect the `scan_en` and `slow_clock` inputs of your third-party OCC to those sources. When the ScanHost node is inserted, the complete fanout of the specified sources moves so that the output pins of the ScanHost node drive them, and your original sources are connected to the `bypass_in` pins of the ScanHost. See [Example 2 in the ScanHost](#) reference page for more information about this flow.

Unlike with the Tessent OCC, you must describe the third-party OCC to the Tessent Tool. To do this, use the “[add_clocks -capture_only](#)” command manually on each OCC output clock pin, and provide a clock control definition as described in the “[Support for Internal Clock Control](#)” section of the *Tessent Scan and ATPG User’s Manual*.

SSN Frequently Asked Questions

This section contains frequently asked questions and possible solutions about SSN.

1. **Is it possible for an SSN datapath to have a branch that is narrower than the main datapath?**

Tessent SSN does support this, but it is not recommended. IPs with narrower bus widths than the parent design are supported, but the IP is should be rebuilt with the proper width.

Use the SSN multiplexer node to isolate the narrower branch of the bus from the parent design. When an SSH on the narrower bus is active, the effective bus width is scaled down to the narrower bus width.

An IP with a wider bus than the parent design is not supported.

2. **Can I have multiple independent active datapaths?**


SSN supports multiple SSN datapaths operating concurrently at different frequencies for retargeting. During top-level ATPG, the effective bus width is scaled down to the narrower datapath width. The two datapaths are capture-aligned, with padding added to the packet to ensure capture alignment.

3. **Can I change the SSN bus_clock and shift_capture_clock frequencies during retargeting?**

You can change the bus_clock and shift_capture_clock frequencies during retargeting after the patterns have already been created at the block level.

During retargeting, set the set_current_physical_block command to the scope of the physical block that you want to change either frequency. Use the set_load_unload_timing_options command to change the frequencies.

Note

 The SSN bus_clock frequency operates at the slowest specified frequency set by the set_load_unload_timing_options command at any physical block.

4. **How do I exclude a physical block from top-level ATPG with SSN?**

Do not add the core instance for the EDT inside the physical block that you want to exclude. Without the core instance of the EDT added, the SSH is disabled and behaves as a two-stage pipeline.

5. What design view of my physical blocks should I use during pattern retargeting?

Use the graybox design view of the physical block whose patterns you want to retarget.

6. How do I program my SSN multiplexer nodes?

Use the generated PDL to write the select of the SSN multiplexer using the `set_test_setup_icall` command.

```
set_test_setup_icall \
    "chip_top_ssh_insertion_tessent_ssn_mux_returnPath_inst.setup
    select_secondary_bus 1" -append
```

7. How is the negative edge of scan_en synchronized across all SSH during top-level ATPG?

During top-level ATPG, one SSH may enter the capture state before another SSH in the same active datapath. This may happen with certain bus configurations and is normal behavior. During top-level ATPG, all active SSH nodes are capture-aligned. Each SSH receives the same number of packets before entering or leaving the capture state. This guarantees that each physical block does not miss any capture clock pulses.

8. Can I use a custom test_setup sequence with “pulse TCK” and “force TMS” to initialize an analog IP and still use SSN?

Yes, add your custom test_setup sequence using the `set_procfname_name` command, and the SSN initialization is appended at the end of your custom sequence.

9. How do I enable Streaming-Through-IJTAG?

Use the `set_ssn_options` command to enable the Streaming-Through-IJTAG mode.

```
set_ssn_options -streaming_interface ijtag
```

10. How do I decide the width of my SSN bus?

Reuse the same number of GPIO used for scan in/out without SSN. The SSN `data_in` and `data_out` ports should be symmetrical.

11. How fast can I run my SSN bus?

The default SSN bus_clock frequency is 400 MHz. A wide high-speed bus requires physical resources. Consult with your physical implementation team to work within the physical design budget.

12. Do all my physical blocks have to shift at the same frequency?

No, each physical block can shift at an independent frequency.

13. Can I change the shift frequency of one of my partitions during retargeting?

You can lower the shift frequency of a physical block after the patterns have been created. Set the `set_current_physical_block` command to the physical block to change

the frequency and use the “set_load_unload_timing_options -shift_clock_period” command to change the shift clock frequency. You can see the changed frequency using the report_load_unload_timing_options or get_load_unload_timing_options commands.

14. Do I have to insert SSH, EDT, and OCC in the same DFT insertion step?

The SSH, EDT, and OCC do not have to be created or inserted at the same time. However, if you do not insert them at the same time in one DFT insertion step, you are responsible for making the physical connections between them and for the multiplexing logic that shares the path back to the SSH between external and internal test EDT channels.

SSN Limitations

Use of SSN is subject to limitations in pattern writing, ScanHost identification, failure mapping, Streaming-Through-IJTAG mode, automatic configuration, and certain commands that are incompatible with SSN.

Pattern Writing Limitations

Pattern writing using the write_patterns command for designs including SSN has the following limitations:

- The following write_patterns command switches are not supported for SSN:
 - -PATtern_size
 - -MEMory_size
 - -SCAN_Memory_size
- For hierarchical designs, writing core-level patterns in the PatDB format using the write_patterns command is the only source for retargeting scan patterns to the top level. ASCII format patterns are not supported for retargeting.

ScanHost Identification Limitations

Other instruments, such as EDT and OCC, can use the add_core_instances command to associate a core description currently in memory with a specified core instance or instances in the design.

- Do not use the add_core_instances command to add SSN ScanHost core instances.

The ScanHost instances are inferred from the ICL, and the active ScanHost instances are automatically identified.

Failure Mapping Limitations

Failure mapping in the presence of SSN does not support designs with the following characteristics:

- When you have compare groups with multiple instances, failure mapping is not able to uniquely identify the failing instance.

Streaming-Through-IJTAG Limitations

You cannot use Streaming-Through-IJTAG if all of the following are true:

- There are enabled pipelines that you specified using the `icl_port` attribute `tessent_pipeline_stages` on the streaming path.
- One or more of the active scan hosts on that path are capture-aligned with other active scan hosts in the streaming scan network.

For more information about Streaming-Through-IJTAG, refer to the section “[Streaming-Through-IJTAG Scan Data](#)” on page 405.

Automated SSN Datapath Configuration

SSN does not support automated programming of SSN multiplexers. You must manually configure these multiplexers using IJTAG to ensure that all of the SSH you intend to use are along the active datapath between the SSN `bus_in` and `bus_out`.

LogicBIST Functionality

You cannot implement SSN and LogicBIST on the same design without access to beta functionality. If you need to implement both sets of functionality in the same design, contact your Siemens Digital Industries Software representative.

Command Limitations

You cannot use the `macrottest` command with SSN.

Manual Integration of SSN Datapath

The SSN datapath is subject to the following limitations for manual integration of the datapath:

- You must ensure there are no incidental inversions or permutations of bits on the SSN bus.
- The bit sequence of the datapath must be maintained throughout the SSN bus.

STARTPAT and ENDPAT in SSN Serial Test Bench

SSN serial test benches do not contain STARTPAT and ENDPAT statements. The STARTPAT/ENDPAT mechanism does not work with serial test benches in SSN, because the SSN stream is a single shift pattern that cannot be broken up. If you need to break up the pattern manually, save it again using -begin and -end values instead. Refer to the section “[Verilog Plusargs](#)” in the *Tessent Scan and ATPG User’s Manual* for more information about using Verilog plusargs like STARTPAT and ENDPAT.

Chapter 9

Tessent Examples and Solutions

This chapter contains Tessent solutions for specific DFT scenarios and problems. This includes applications and flows that solve common, difficult issues encountered in DFT.

How to Avoid Simulation Issues When Using the Two-Pin Serial Port Controller ...	491
How to Handle Clocks Sourced by Embedded PLLs During Logic Test.....	494
How to Design Capture Windows for Hybrid TK/LBIST.....	500
How to Use Boundary Scan in a Wrapped Core.....	502
How to Use an Older Core TSDB With Newly-Inserted DFT Cores	504
TAP Configuration	506
Insert a Stand-Alone TAP in a Design	506
Insert a TAP with an IJTAG Host Scan Interface.....	508
Insert a Compliance Enable TAP with an IJTAG Interface	509
Insert a Daisychained TAP	511
Insert a Primary TAP	512
Insert a Secondary TAP	514
Connecting to a Third-Party TAP.....	517
How to Set Up Third-Party Synthesis	517
How to Set Up Support for Third-Party OCCs	520
How to Configure Files for Third-Party OCCs	520
Test Logic Insertion	521
Configuration for Scan Insertion	523
Pattern Generation and Simulation.....	523
Post-Synthesis Update.....	527
ICL and TCD Post-Synthesis Update.....	527
Limitations Related to SystemVerilog Interface Arrays.....	528
Updating ICL Attributes From the Design.....	529
Matching Requirements for Port Names in Post-Synthesis Update.....	530
Design Name Mapping Commands	531

How to Avoid Simulation Issues When Using the Two-Pin Serial Port Controller

You may observe some compare failures during signoff simulations using the Two-Pin Serial Port (TPSP) interface. This topic describes a common cause and its solution.

Problem

Some pad models with internal pull resistors also model delays for the pull value, which allows a “Z” or “X” value to propagate to the clock pin of the TRST generator. This will cause a simulation artifact that will propagate an “X” on the network’s TRST signal and effectively fail the simulation.

Solution

The failure is purely a simulation artifact. The solution uses a Verilog side file that prevents an undetermined value from propagating to the TRST generator’s clock pins.

```

`timescale 1ns/1ns
module tpsp_artifact_solution;
  `ifndef TPSP_INST
    `define TPSP_INST top_rtl_tessent_twopinserialport_tpsp_inst
  `endif
  always@(TB.DUT_inst.`TPSP_INST.tessent_persistent_cell_spio_in_trst_clk_buf.A) begin
    if (TB.DUT_inst.`TPSP_INST.tessent_persistent_cell_spio_in_trst_clk_buf.A == 1'bx)
      force TB.DUT_inst.`TPSP_INST.tessent_persistent_cell_spio_in_trst_clk_buf.Y = 1'b0;
    else
      #0.1 release TB.DUT_inst.`TPSP_INST.tessent_persistent_cell_spio_in_trst_clk_buf.Y;
    end
  end
endmodule

```

Using the “run_testbench_simulation” command inside Tessent Shell

1. Create a Verilog side file using the code above, such as “tpsp_artifact_solution.v”.
2. Determine the path to your TPSP instance. If you do not know the path, then you can use the commands below to find it.

```

set tpsp_module [get_icl_modules -filter
tessent_instrument_subtype=="tpsp_controller"]

set tpsp_icl_instance [get_icl_instances -of_modules $tpsp_module]

set tpsp_instance [get_single_name [get_instances -of_icl_instances
$tpsp_icl_instance]]

set tpsp_sim_path [convert_design_path_format $tpsp_instance -to_dot_separator]

```

3. Define the library sources for simulation in the Tessent shell.

```

set_simulation_library_sources -v ../data/sim_models.v ../data/pad_models.v

```

4. Run the simulation using the extra options to provide the TPSP controller’s path, the extra Verilog side file, and the module inside.

```

run_testbench_simulations \
-simulation_macro_definitions "TPSP_INST=$tpsp_sim_path" \
-extra_verilog_files ../data/tpsp_artifact_solution.v \
-extra_top_modules tpsp_artifact_solution

```

Using the Questa Simulator outside Tessent Shell

1. Create a Verilog side file using the code above, such as “tpsp_artifact_solution.v”.

2. Load your design files with Questa’s “vlog” command.
3. Load the side file and provide the path to the TPSP controller instance using the “+define+” command unless it is hard coded in the side file.

```
vlog -work dut_work "../.././data/tpsp_artifact_solution.v" \
+define+TPSP_INST=path.to.instance.top_rtl_tessent_twopinserialport_tpsp_inst
```

4. Run the simulation using Questa’s “vsim” command, adding the module’s name from the Verilog side file. (Use “acc” optimization if simulating with SDF.)

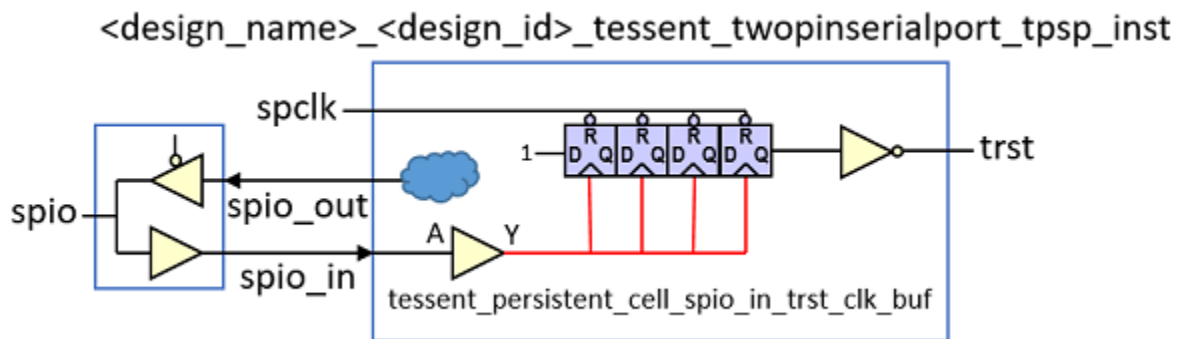
```
vsim -lib "dut_work" -L dut_work -do 'run -all; exit' \
-voptargs="+acc" \
p1_configuration \
p1_sdf \
tpsp_artifact_solution
```

Discussion

The failure is purely a simulation artifact. The proposed solution is to use a Verilog side file to prevent an undetermined value from propagating to the TRST generator’s clock pins.

The [Two-Pin Serial Port](#) (TPSP) controller is an interface for the 1149.1 TAP controller and can drive a TAP with two pins on the chip boundary. A 4-bit shift register inside the TPSP generates the TRST signal pulse for the TAP controllers it drives.

Figure 9-1. TRST Pulse Generator inside TPSP



In Figure 1, an undetermined value (“X”) may occur on the internal “spio_in” net and reach the clock pin of the TRST generator (shown in red), causing corruption of the simulation results. This register is sensitive to these states, and the TPSP protocol is such that the inout port is put in a high impedance state every third cycle. For these reasons, Tessent Shell requires a pull resistor modeled for the SPIO port, whether an [Inout Pad](#) or [Input Pad](#) model.

The pull value will drive the input immediately if the pad has no delay modeled. The registers will not get corrupted, and the simulation will end with the correct results. However, if your pad models a delay for the pull value, you will observe the problem. This problem is purely a simulation artifact and will not be present during manufacturing tests, and you can use a Verilog side file to avoid it.

To solve the simulation artifact when your pad model has a delay on the pull value, you can use a Verilog side file to prevent a corrupting value from reaching the TRST generator registers' clock pins. The side file changes the behavior of the persistent buffer inside the TPSP controller, "tessent_persistent_cell_spio_in_trst_clk_buf" (see Figure 1.), to disable the propagation of an undetermined value through the buffer. The output of the buffer cell (Y) is forced to "0" when the undetermined value is present on its input (A). The forced value is released with minor delay when the value on the buffer's input becomes determined.

An example Verilog side file was provided in the "[Solution](#)" on page 492. The code uses the "define" statement to provide the TPSP controller instance's design path. However, the path could be hard coded as well. Copy the code and create a new Verilog side file (Step 1). Before you can compile your new Verilog side file and run it with the testbench generated by Tessent, you must define values for several TCL variables (Step 2). Define the library sources for simulation using the [set_simulation_library_sources](#) command (Step 3). Finally, run the [run_testbench_simulations](#) command with extra options to use the Verilog side file (Step 4).

You can also compile and run the new Verilog side file directly in a simulation tool such as Questa. Copy the code and create a new Verilog side file (Step 1). Load your design files into the Questa simulator (Step 2). Load the Verilog side file and define the path to the TPSP controller instance using a "+define+" statement unless it is already hard coded in the side file (Step 3). Finally, run Questa simulation using the additional options to utilize the Verilog side file (Step 4).

In summary, you can use a Verilog side file to resolve a simulation artifact. A pad modeled with a delay on the pull value exhibits the problem. You can use a Verilog side file to prevent "X" value propagation by changing the behavior of a buffer inside the TPSP. The side file solution works with a Tessent testbench inside the Tessent shell or the Questa simulator outside the Tessent shell.

How to Handle Clocks Sourced by Embedded PLLs During Logic Test

Clocks sourced by an embedded PLL have local OCCs that are reused when testing parent physical regions. The flow is different when the parent physical regions are wrapped cores.

Problem

For embedded PLLs, such as the one shown inside "corec" in [Figure 9-2](#), the OCC inserted on the VCO output of the PLL is used during the internal logic test modes of the core. The OCC is also reused during the internal test modes of its parent physical regions.

When the PLL is inside a wrapped core that is the child of another wrapped core, you must ensure that the OCC is still controllable by the scan chains when running logic test modes of the top level.

Solution

Wrapped Core Only Used in the Top Level

If the PLL is embedded inside a wrapped core that is only used inside the top level physical region, then no further action is needed. The standard flow handles this scenario automatically, as described in “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141.

Wrapped Cores Used Inside Other Wrapped Cores

You must use the following procedure when the wrapped core in which the embedded PLL is located has another wrapped core above it, as shown in [Figure 9-2](#)

1. Add an extra DFT signal when you process the wrapped core that embeds the PLL in “[Second DFT Insertion Pass: Inserting Top-Level EDT and OCC](#)” on page 171 (for example, when processing corec) as follows:

```
add_dft_signals promoted_cells_mode
```

2. Create a new scan mode when you process the wrapped core that embeds the PLL in the “[Scan Chain Insertion](#)” step (for example, when processing corec) as follows:

```
set promoted_instances \
    [get_attributed_objects \
        -attribute_name wrapper_type_from_clock_source \
        -object_type instance]
if {[sizeof_collection $promoted_instances] > 0} {
    add_scan_mode promoted_cells_mode \
        -include_elements [get_scan_elements \
            -of_instances $promoted_instances]
}
```

The new scan mode contains the control flip-flops of the OCCs that need to be accessible from the top-level.

3. Modify the definition of the external scan mode when you process the parent wrapped core in the scan chain insertion step of the flow (for example, when processing corea).

```
set_attribute_value corea_i1 -name active_child_scan_mode \
    -value promoted_cells_mode

set_ext_scan_elements [add_to_collection \
    [get_scan_elements -class wrapper] \
    [get_scan_elements -of_child_scan_modes promoted_cells_mode ]]

add_scan_mode ext_mode -chain_length 32 \
    -include_elements $ext_scan_elements
```

Modifying the external scan mode definition enables you to include the promoted scan chains from the child wrapped cores in the external chain of the current wrapped core. This step ensures that the control flip-flops of the embedded OCCs are accessible by the test modes of the top level.

4. Depending on the hierarchy of your design, do one of the following:
 - o If your design only uses the wrapped core at the top level of the chip (such as corea), then no further modifications are required for the standard flow.
 - o If your design embeds the wrapped core inside another wrapped core (for example, there was a layer of wrapped core between corea and top), you need to recreate the promoted scan mode at the current level.

This step provides access to the OCC control bits in its external scan mode. The method shown in Step 3 is reapplied to that parent wrapped core as follows:

```
set_attribute_value corea_i1 -name active_child_scan_mode \
  -value promoted_cells_mode
add_scan_mode promoted_cells_mode \
  -include_elements [get_scan_elements \
    -of_child_scan_modes promoted_cells_mode]
```

Discussion

The example chip shown in the following figure illustrates functional clocking when a PLL is embedded inside a child physical region.

Figure 9-2. Example Chip with PLL Embedded Inside Lower Core

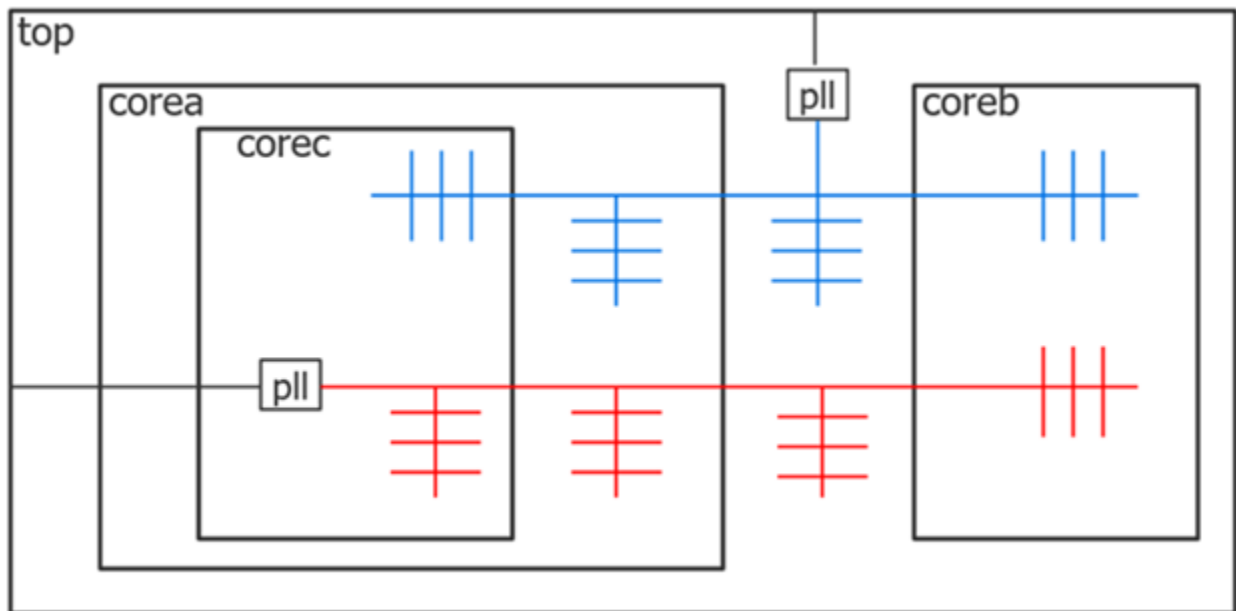
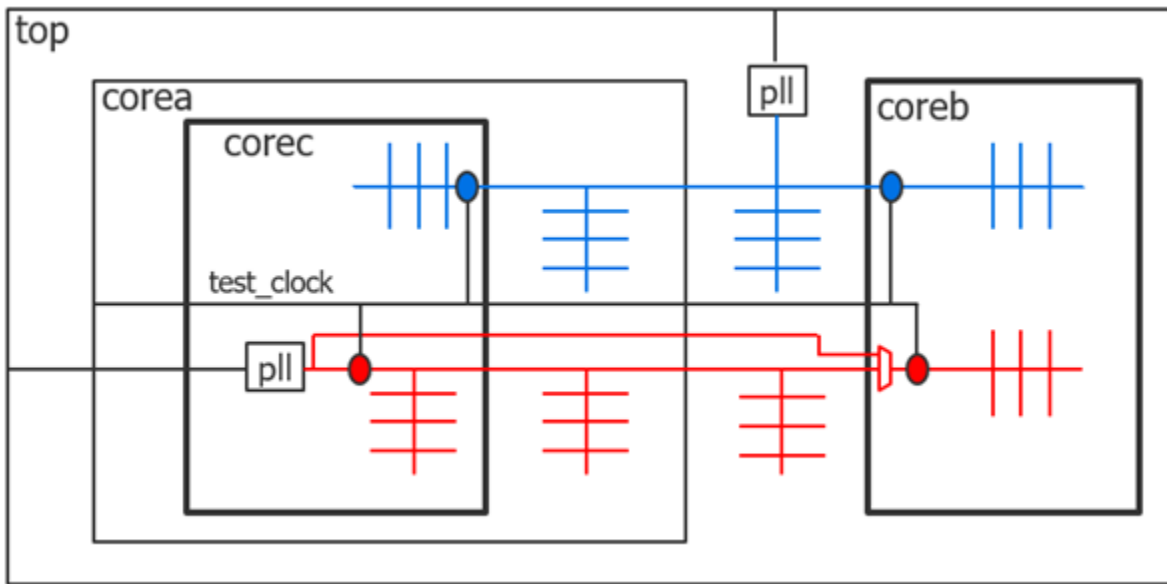


Figure 9-3 shows the location of the OCCs inserted inside corec and coreb. The two OCCs inside corec are active during its internal test modes to inject the shift clock during the shift cycles and to chop the functional clocks during capture cycles. The two OCCs inside coreb are also active during its internal test modes.

Figure 9-3. Active OCCs During Internal Test Modes of corec and coreb



If you want to run the internal test modes of corec in parallel with those in coreb, you need to provide a clock bypass path, such that the free running output of the PLL can continue to source the red clock of coreb when the OCC inside corec is active.

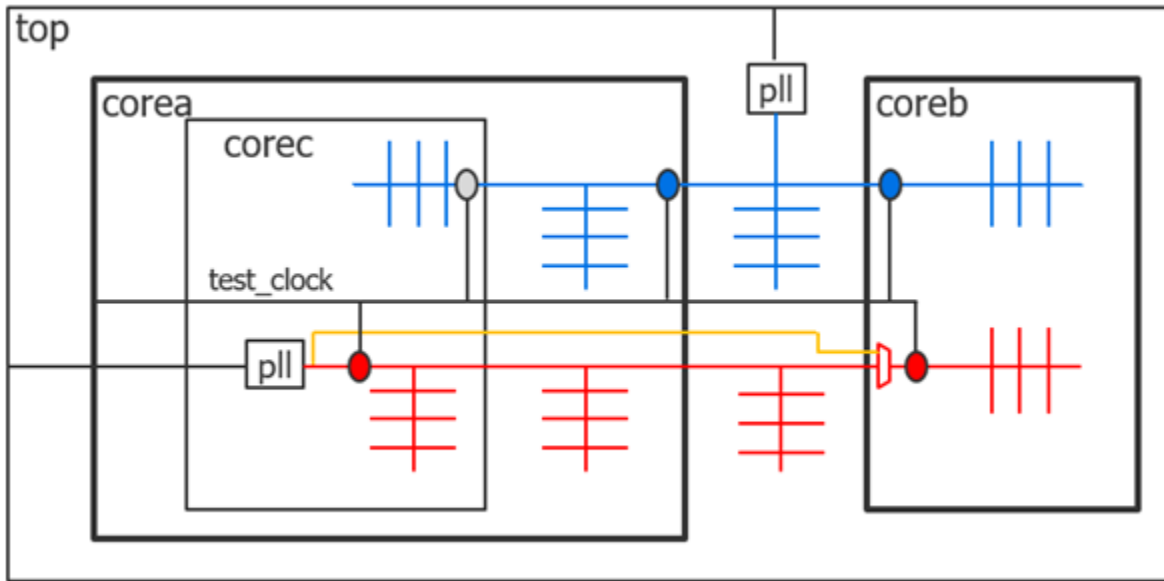
The bypass clock path is not required. The tool issues an error if you try to re-target an internal test mode of coreb in parallel with an internal test mode of corec or corea, and the bypass path is not present. The reason for this check is that the OCC at the input of the red domain of coreb expects and requires a free running clock when active. The red clock output of corea is not a free running clock when the red OCC inside corec is active. Instead, it is alternating between the shift clock and bursts of at-speed clock pulses.

If you provide the bypass clock path, you reduce the overall chip test time as you can concurrently test corec or corea with coreb. If you want to insert the clock bypass path within the DFT insertion process, use a [process_dft_specification.post_insertion](#) callback to create the ports and make the connections. Use the [intercept_connection](#) command to insert the multiplexer inside coreb. The best option to control the select input of the multiplexer is to register and add a new DFT signal. See the [register_static_dft_signal_names](#) command for more information.

The [get_dft_signal](#) command in the [process_dft_specification.post_insertion](#) callback gets the connection point for the added DFT signal. If the bypass path is manually added in the golden RTL, leave the select input of the multiplexer tied low and connect it to the DFT signal during the DFT process with the [add_dft_control_points](#) command. See of the [register_static_dft_signal_names](#) command description section for an example.

When you move up to corea, another OCC is inserted at the base of the blue clock domain to be used during its internal logic test modes, as shown in [Figure 9-4](#).

Figure 9-4. Active OCCs During Internal Test Modes of corea and coreb



The OCC on the blue domain inside corec is kept inactive, as it is during the functional mode. The clocking of the entire blue clock domain is controlled by the OCC located inside of corea at the base of the clock tree. The red OCC inside corec is active during the internal test mode of corea to control the scannable flip-flops on the red domain inside corea, as well as the wrapper flops on the red domain inside corec. Because the “fast_clock” input of that red OCC is not sourced by an input of corec, its scan chain is automatically promoted to its wrapper chains. This promotion enables the scan chain to be under ATPG control when running the internal test mode of corea.

If corec was not embedded within another wrapped core (such as coreb that is directly instantiated in the top level), the handling is completely automated and no deviation from the standard flow, described under “[Tessent Shell Flow for Hierarchical Designs](#)” on page 141, is necessary. However, when the wrapped core containing the embedded PLL is inside another wrapped core, you must follow the steps described under “[Solution](#)” on page 495 to enable the embedded OCC inside corec to be under ATPG control at the top level.

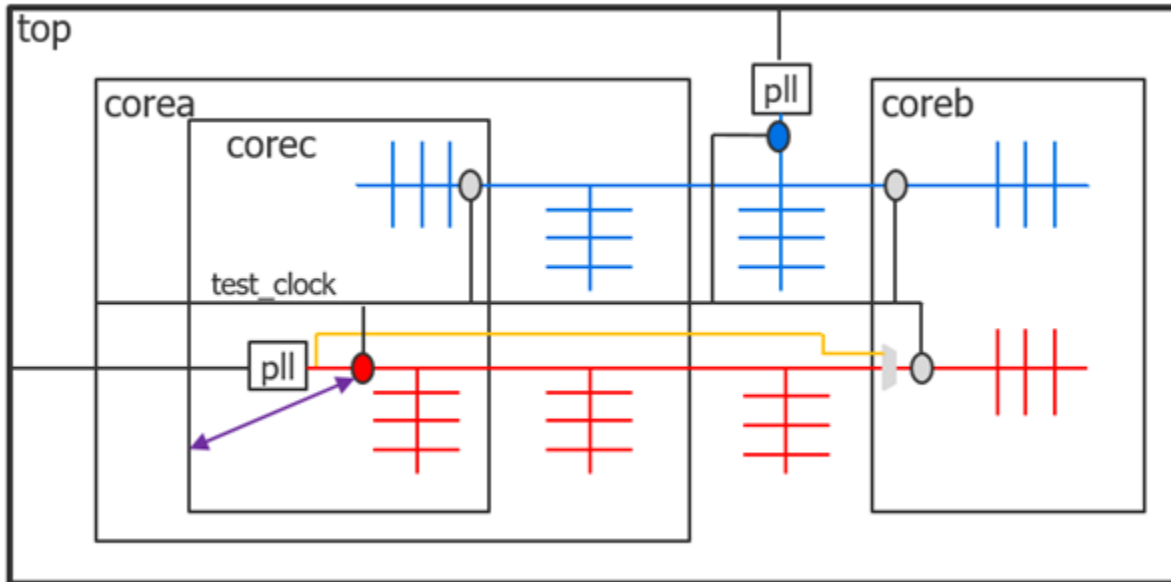
Extra DFT Signals

Figure 9-5 shows the active OCCs when running the logic test mode of the top level. The red OCC inside corec is active, which requires that the scan segment that contains the control flip-flops of the red OCC must be accessible by the scan chains of the top level. This requirement is why Step 1 shows how to add a new DFT signal called “promoted_cells_mode” to use as the enable for that scan chain configuration.

The OCCs within the cores are implemented with the [shift only mode](#) parameter. This has the advantage of keeping the internal core shift timing identical between internal and external modes. The relative shift timing between the many functional clock domains and the EDT clock domain is derived from the common test_clock entering each core. You can close timing during

layout on each core and not have to wait until you run final STA from the top level to know how fast each core shifts in external mode.

Figure 9-5. Active OCCs During Test Modes of Top Level



If the OCCs do not have the shift only mode, the shift clock is injected through the red and the blue OCCs. The relative timing of the shift clock as it arrives at the input of the two clocks inside coreb is not known until the clock trees of the two functional clocks are completed, which does not happen until you complete the layout of the top level.

When you use the `add_dft_signals` command to add the `ext_ltest_en` DFT signal to a given core, the OCCs of that core are automatically equipped with the shift only mode.

New Scan Mode

Step 2 shows how to create the scan mode that contains the control flip-flops of the promoted OCCs. The OCC instances that have their “fast_clock” input controlled by an internal source have an attribute named “wrapper_type_from_clock_source” set on them automatically. The `get_attributed_objects` command is used to find them and make them part of the special scan mode.

Promoted Scan Chains

When you get to the corea level, you need to promote the special scan chain on corec into the wrapper chain of corea, such that the control flip-flops of the red OCC are under ATPG control from the top level. This task is described in step 3.

Step 4 provides instructions for when corea is not directly instantiated into the top level, but instead is embedded within another wrapper core. In that case, you need a special scan mode to collect the embedded OCC chains such that they can be included in the wrapper chains of the

parent wrapped core. You follow step 3 on the parent wrapped core to include the promoted scan mode of corea within its wrapper chain.

The purple line in [Figure 9-5](#) represents the promoted scan chain that contains the control flip-flops of the red OCC. This scan segment is inserted in the “ext_mode” of corea in step 3 such that the OCC is part of the scan chains of the top level.

Related Topics

[On-Chip Clock Controller Design Description](#)

[OCC](#)

How to Design Capture Windows for Hybrid TK/LBIST

A capture window is a group of capture cycles defined by one or more clocks. Test logic can be configured to run a selected number of patterns for each capture window. This approach gives full control over the patterns to optimize test coverage and test power consumption.

Problem

The fundamental objectives for LBIST are increased test coverage, decreased test time, and lower power consumption for the test run. In a typical flow, the full set of data needed to perform optimal insertion to meet these objectives is not available until the gate-level netlist is ready. In practice, test circuitry is closely integrated into the design and the design suffers when test is treated as an ad-hoc component or inserted later in the gate-level netlist.

To address these issues and achieve a high level of test coverage and performance, the Hybrid TK/LBIST flow enables you to insert DFT logic at the RTL level, before the gate-level netlist is ready. Capture windows enable the flexibility to add test logic in the RTL and test accurately.

Solution

The solution is provided in two parts:

- Define capture windows (which clocks and how many pulses from each clock).
- Determine how many patterns to run per capture domain to achieve the targeted coverage.

Define Capture Windows

If the clocking structure is known and determined during the insertion of the IP, define capture windows for this flow by following the examples under “[NCP Index Decoder](#)” in the *Hybrid TK/LBIST Flow User's Manual*.

If the clocking structure is not yet defined or finalized, use the following procedure:

1. Create a gate-level netlist using quick synthesis.
2. Run ATPG with the following constraints:
 - a. Use the same setup and constraints used for LBIST.
 - b. Set the number of random patterns:

```
set_random_pattern integer
```

where the integer argument is the number of patterns that are planned for use in LBIST.

- c. Run pattern simulation:

```
simulate_patterns -source random -store all
```

- d. Report the test coverage per clock domain:

```
report_statistics -clock all
```

- e. Report the generated patterns:

```
report_patterns
```

- f. Design your capture windows using information from the ATPG run and the report_statistics command.

- The ATPG run helps you identify the test clock domains in the design.
- The report_statistics command helps you identify the number of clock pulses per clock domain

- g. Determine how many patterns to run for each capture domain to achieve the targeted coverage.

In the following example, the design has three clock domains, with possible interaction between CLK2 and CLK3. The highest faults per domain is at CLK2 at 78%, followed by CLK3 at 43%.

Clock Domain Summary	% faults (total)	Test Coverage (total relevant)
CLK3_OCC	43.07%	99.13%
CLK2_OCC	78.80%	99.08%
CLK1_OCC	22.98%	98.17%

Patterns to Run per Capture Domain

For stuck-at-fault, try to use as many clock domains in the capture window as possible. This saves test time. Complete the steps under Define Capture Windows in the preceding to select the number of patterns per capture window.

In the following example, to achieve the best coverage with the lowest number of patterns, use a two-cycle capture window for 20% of the patterns and use a single CLK2_OCC/CLK1_OCC pulse capture window for 80% of the patterns.

patt. #	pre-filtering pattern #	type	cycles	loads	capture_clock_sequence
0	0	basic	1	1	[CLK1_OCC, *]
1	1	basic	1	1	[CLK2_OCC, *]
...			
1	400	basic	1	1	[CLK2_OCC, *]
2	401	clock_sequential	2	1	[CLK2_OCC, *] [CLK3_OCC, *]
3	405	clock_sequential	2	1	[CLK2_OCC, *] [CLK3_OCC, *]
3	500	clock_sequential	2	1	[CLK2_OCC, *] [CLK3_OCC, *]

Note: [*] = "SHIFT_CLOCK", which is a pulse-in-capture clock.

How to Use Boundary Scan in a Wrapped Core

You must perform a specific procedure if a wrapped core has embedded pads that require boundary scan. Care must be taken during the insertion of the boundary scan and during ATPG.

Problem

Pads embedded inside a wrapped core must be identified such that boundary scan cells can be added. The resulting boundary scan cells must also be made visible to the tool in order for them to be included in scan chains and to apply scan patterns.

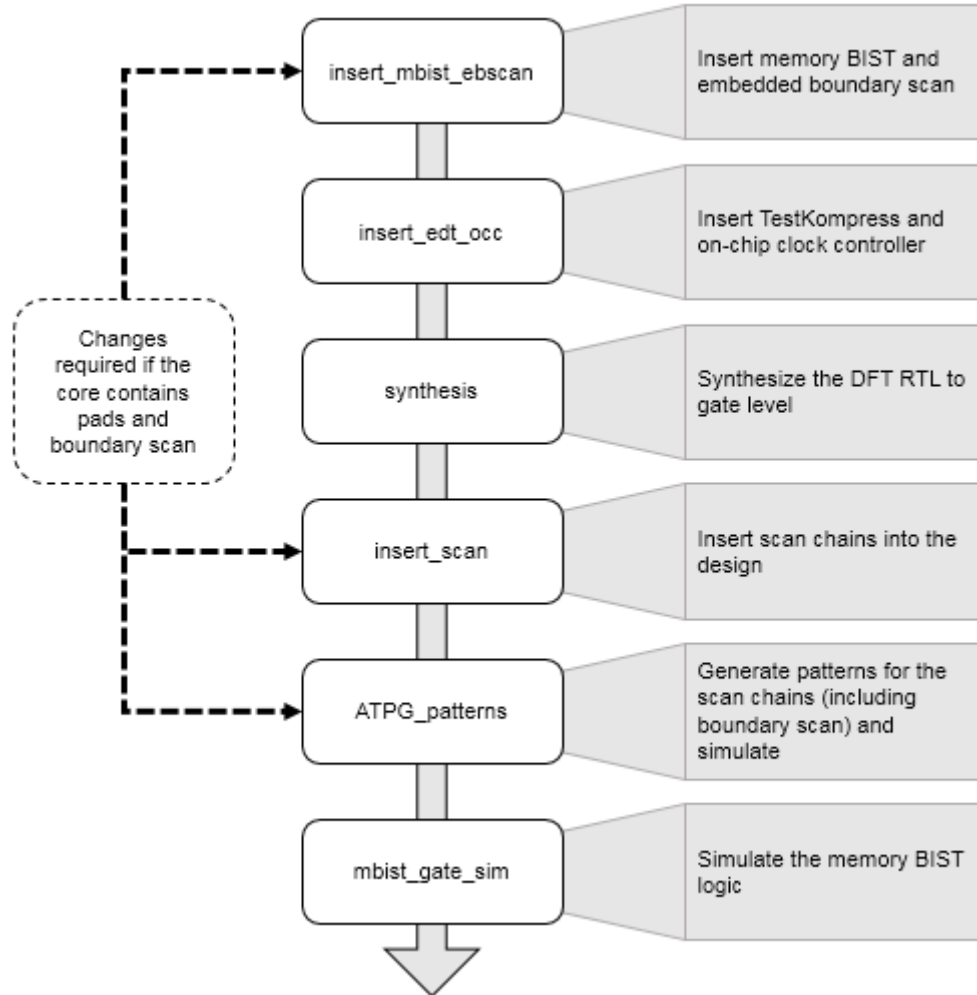
Solution

The solutions are given for wrapped core and chip-level flows.

Wrapped Core Flow

The following figure shows the standard command flow for creating a wrapped core that includes embedded boundary scan:

Figure 9-6. Wrapped Core Boundary Scan Flow



If the core contains pads and boundary scan, you must include the following commands in the wrapped core flow shown in [Figure 9-6](#) on page 503:

- Insert memory BIST and embedded boundary scan using during the `insert_mbist_ebscan` step as follows:
 - a. Specify DFT requirements to insert memory test and embedded boundary scan:

```
set_dft_specification_requirements -memory_test on \
  -boundary_scan on
```

- b. Insert embedded boundary scan cells and identify the pads:

```
set_boundary_scan_port_options \
  -pad_io_ports [list taclk ta_cci0a ta_cci0b ta_cci1a ta_cci1b
    ta_cci2a ta_cci2b ta_out0 ta_out1 ta_out2]
```

Note

The order specified is the order in which the boundary scan cells are inserted.

- Specify not to add any dedicated wrapper cells on the embedded pad I/O during the insert_scan step:

```
set_dedicated_wrapper_cell_options off \
  -ports {ta_out0 ta_out1 ta_out2}

set_dedicated_wrapper_cell_options off \
  -ports {ta_cci0a taclk ta_cci0b ta_cci1a ta_cci1b ta_cci2a
    ta_cci2b}
```

- Preserve the boundary scan instances in the graybox during the ATPG_patterns (graybox generation) step.

```
set_preserve_bscan {}

set_preserve_bscan \
  [get_instances -hier *_tessent_bscan_logical_group_*]
analyze_graybox -preserve_instances $preserve_bscan
write_design -tsdb -graybox -verbose
```

Chip-Level Flow

For boundary scan at the chip-level, follow the standard chip-level flow by inserting boundary scan as the first step in the flow. There are no specific additions for embedded boundary scan at this phase of the flow. The boundary scan segments from the wrapped core are picked up automatically during chip-level boundary scan insertion.

How to Use an Older Core TSDB With Newly-Inserted DFT Cores

When you have existing cores with Tessent DFT inserted, you can use them with automation through the Tessent Shell database (TSDB).

Prerequisites

- Legacy core TSDB file.

Procedure

- Open the TSDB of the legacy core:

```
open_tsdb <tsdb_directory>/legacy_core.tsdb
```


Tessent Shell provides automation through the TSDB directory and TCD files. From the TSDB, the tool finds all available TCD files that apply to the design.

2. Load the current design:

```
read_design <core_name> -design_id <final_design_id>
```

3. Prepare your legacy cores by configuring them into the correct mode.

- a. If scan insertion was performed using Tessent Scan, use the [import_scan_mode](#) command during pattern generation to import the EDT and OCC settings.
- b. If the scan insertion was not performed using Tessent Scan, but there are TCD files in the TSDB, use the [add_core_instances](#) command to configure the EDT and OCC.

```
add_core_instances -module <module_name>
```

- c. If scan insertion was not performed using Tessent Scan and there are no TCD files for OCC, manually add clock information using the [add_clocks](#) command and define the clock definition as a third-party OCC.

```
# For OCC clock definition
add_clocks <occ_output_mux/y_pin> -capture_only
# For procedure setup of OCC
set_test_setup_icall "<OCC_name.setup>"
# For OCC clock definition file
read_procfile <occ_control.proc>
```

4. Perform pattern generation with a regular ATPG session.

Results

You have successfully added legacy cores to newly-inserted cores using the TSDB.

TAP Configuration

Tessent Shell typically relies on an IEEE 1149.1-based Test Access Port (TAP) as the primary access mechanism to the DFT it inserts. When boundary scan is implemented, the Tessent TAP fully complies with IEEE 1149.1 standard requirements, however many other possible configurations are possible.

This section provides a quick reference to the various TAP insertion styles that are supported, how to specify them, and what to expect from such implementations.

Insert a Stand-Alone TAP in a Design	506
Insert a TAP with an IJTAG Host Scan Interface	508
Insert a Compliance Enable TAP with an IJTAG Interface	509
Insert a Daisychained TAP	511
Insert a Primary TAP	512
Insert a Secondary TAP	514
Connecting to a Third-Party TAP	517

Insert a Stand-Alone TAP in a Design

Insert a stand-alone TAP in a design. The TAP generated in this example can be further enhanced with additional IR opcodes, IJTAG host scan interfaces, and decoded outputs to enable downstream logic, such as another TAP.

Solution

1. Set the design level to “chip.”
2. Use the following dofile example to insert the TAP:

```
set DFT [create_dft_specification]
read_config_data -in $DFT -from_string {
  IjtagNetwork {
    HostScanInterface(ijtag) {
      Interface {
        tck : tck;
      }
      Tap(single) {
      }
    }
  }
}
process_dft_specification
```

The generated TAP connects to the trst, tck, tms, tdi, and tdo pads that were present in the pre-DFT design.

Discussion

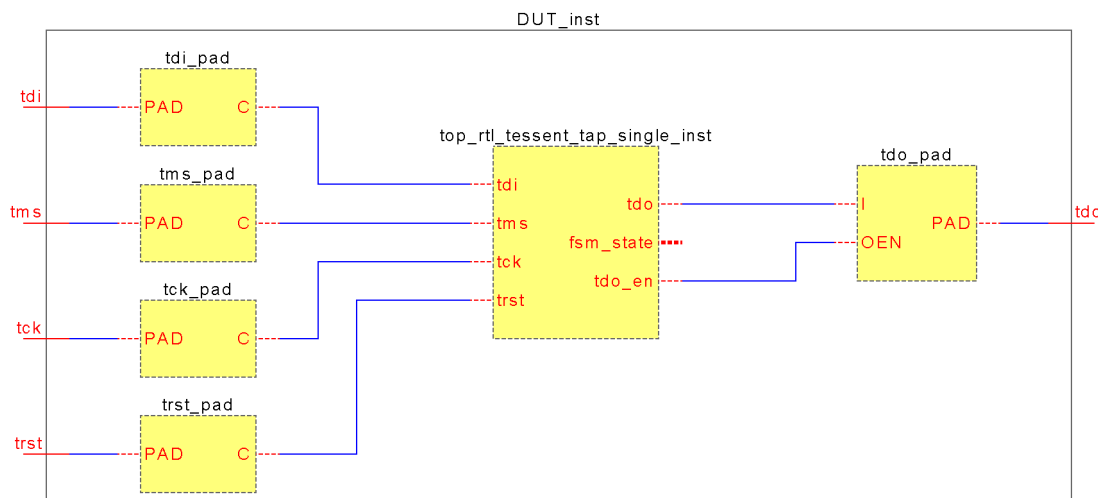
If the TAP pins in the current design do not use the default names (trst, tck, tms, tdi, or tdo), then they can be mapped using one of the following:

- The `DftSpecification::IjtagNetwork::HostScanInterface::Interface` wrapper.
- The following dofile command:

```
set_attribute_value portname -name function \  
-value [trst | tck | tms | tdi | tdo]
```

The tool issues errors if the TAP pads are not yet present in the current design. If the design is only temporary, you can specify the “`set_design_level sub_block`” command instead. Many TAP-specific DRCs are not run in such a case and other side effects may result. You should read an updated design that includes TAP pads as soon as possible.

The following is a schematic representation of this example:



Related Topics

[Insert a TAP with an IJTAG Host Scan Interface](#)

[Insert a Compliance Enable TAP with an IJTAG Interface](#)

[Insert a Daisychained TAP](#)

[Insert a Primary TAP](#)

[Insert a Secondary TAP](#)

[Connecting to a Third-Party TAP](#)

Insert a TAP with an IJTAG Host Scan Interface

Insert a TAP equipped with an IJTAG host scan interface. An IJTAG network can then be connected to the TAP in a subsequent test insertion phase.

Solution

1. Set the design level to “chip.”
2. Use the following dofile example to insert the TAP:

```
set DFT [create_dft_specification]
read_config_data -in $DFT -from_string {
  IjtagNetwork {
    HostScanInterface(ijtag) {
      Interface {
        tck : tck;
      }
      Tap(single) {
        HostIjtag(1) {
        }
      }
    }
  }
}
process_dft_specification
```

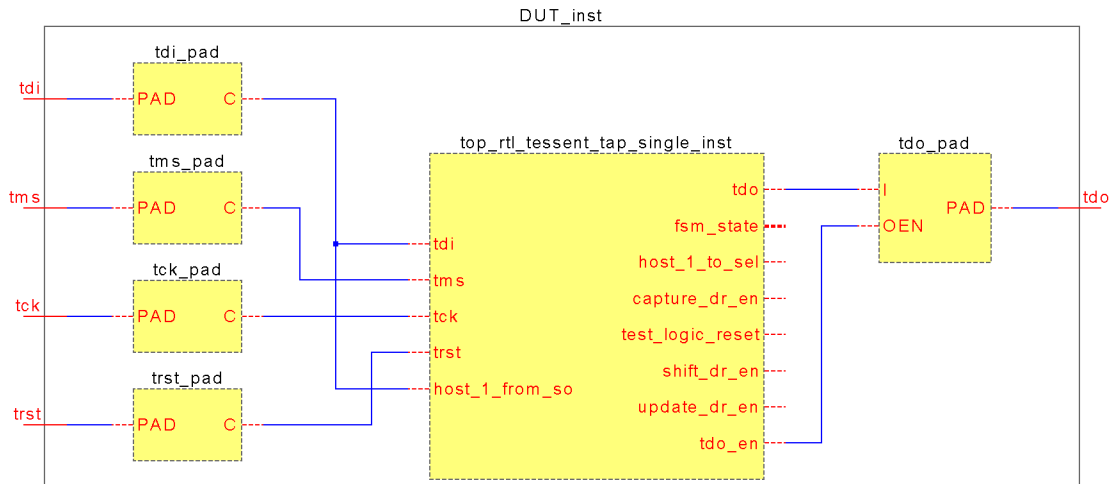
Discussion

The host scan interface on the generated TAP can be used to control any type of IJTAG-compliant network.

The host scan interface provides a `test_logic_reset` output to reset downstream logic; it asserts the `host_<hostname>_to_sel` output to 1 when accessing the client IJTAG network.

The `capture_dr_en`, `shift_dr_en`, and `update_dr_en` outputs are consumed by the client IJTAG network or instruments after qualification with the `host_<hostname>_to_sel` port.

The following is a schematic representation of this example:



Related Topics

[Insert a Stand-Alone TAP in a Design](#)

Insert a Compliance Enable TAP with an IJTAG Interface

Insert a TAP in a design with compliance enable (CE) decoding logic. The CE decoding logic ensures that the TAP can only be accessed if all of the specified values are present on the primary inputs. The same scheme is also used when the TAP pins are shared with functional pins. The CE pins ensure that the TAP is only activated during intended test modes and not accidentally, for example, while the functional pins toggle during normal operation.

Solution

1. Set the design level to “chip.”
2. Ensure that at least one primary input pad is set to 0 or 1 to enable the TAP logic.
3. Use the following dofile example to insert the TAP:

```
set DFT [ create_dft_specification \
    -active_low_compliance_enables {tap_en0} \
    -active_high_compliance_enables {tap_en1} ]
read_config_data -in $DFT/IjtagNetwork/HostScanInterface(tap) \
    -from_string {
        Tap(CE_decoded) {
            HostIjtag(1) {
            }
        }
    }
process_dft_specification
```

Discussion

Note how the CE pins are specified as options to the `create_dft_specification` command.

The CE decoding module relies on the CE pin values to select the active TDO and TDO_EN signals and route both to the primary TDO output pad.

The same CE module also gates the TMS signal such that when the proper values are not present, the TMS input on the TAP is kept to 0. This normally has the effect of “parking” the TAP in one of the stable FSM states (refer to the IEEE 1149.1 FSM state diagram for details).

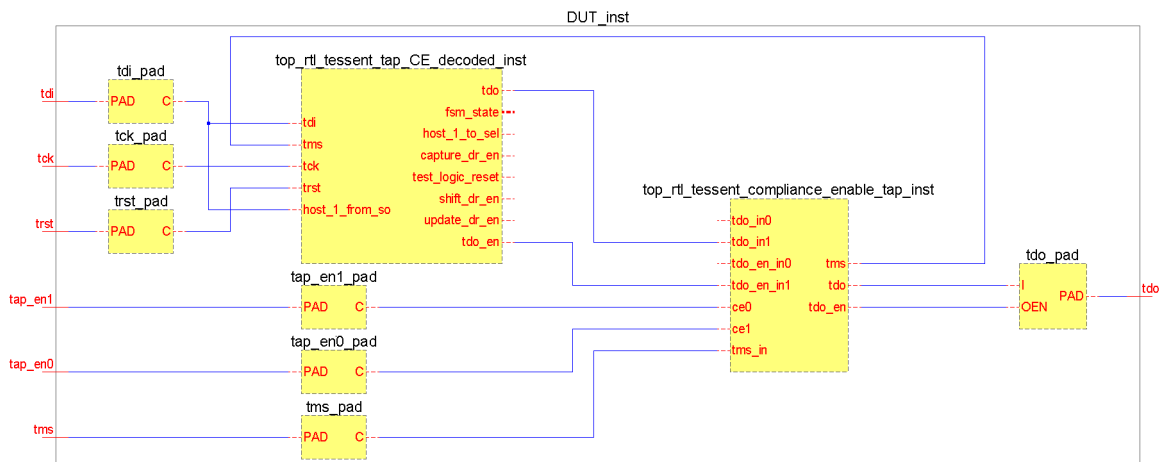
Caution

Do not confuse the CE decoding module input port names with expected CE values! For instance, in the preceding diagram the “ce0” input is actually sourced by the CE pin driven at 1; the “ce1” input is connected to the CE pin driven at 0.

The general rule is that if n CE inputs are specified, the tool creates CE decoding logic with input ports ranging from `ce<n-1>` down to `ce0`.

If internal CE nodes have to be used (that is, when the pre-DFT design already contains decoding logic and hookup points to connect the new TAP to), declare those hierarchical nodes in a new `DftSpecification::IjtagNetwork::HostScanInterface::Interface` wrapper. The `Tap<name>` wrapper then has to be put within the very same interface wrapper.

The following is a schematic representation of this example:



Related Topics

[Insert a Stand-Alone TAP in a Design](#)

Insert a Daisychained TAP

This example inserts a second TAP in series with an existing one. The TRST, TCK, and TMS signals are shared between both TAPs, which implies that the complete JTAG chain (for example, top-level TDI to TDO) always shifts through both TAPs.

Solution

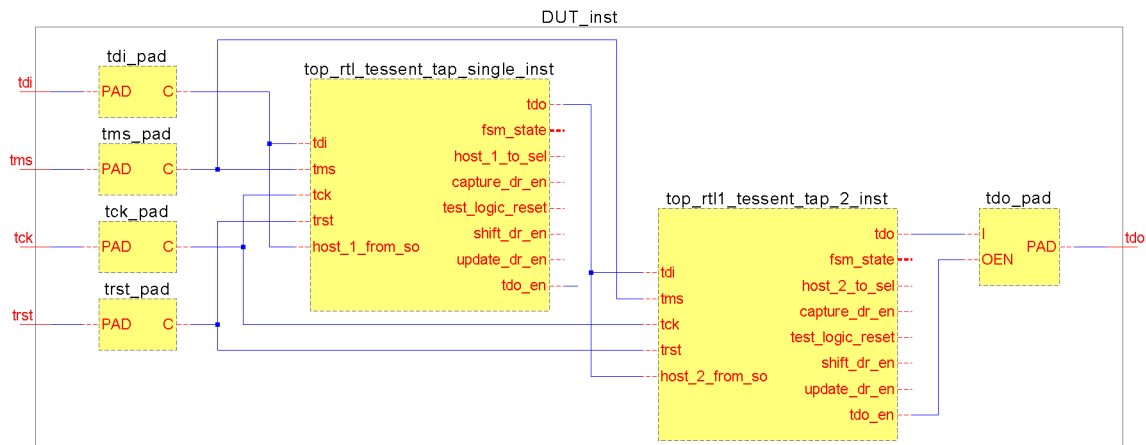
1. Set the design level to “chip.”
2. Use the following dofile example to insert the TAP:

```
set DFT [create_dft_specification]
read_config_data -in $DFT -from_string {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck;
        daisy_chain_with_existing_client : on;
      }
      Tap(2) {
        HostIjtag(2) {
        }
      }
    }
  }
}
process_dft_specification
```

Discussion

The first TAP in this example has a name that starts with “top_rtl_”, while the second TAP begins with “top_rtl1_”. These names are used because this step is typically done as a subsequent insertion pass, such that the current design already contains at least one valid TAP.

The following is a schematic representation of this example:



Related Topics

[Insert a Stand-Alone TAP in a Design](#)

Insert a Primary TAP

This example inserts a TAP into a design that does not contain one. The generated TAP is a primary TAP. Primary TAPs provide outputs to enable secondary TAPs, which are inserted later.

Solution

1. Set the design level to “chip.”

2. In the `IjtagNetwork::HostScanInterface::Tap` wrapper, create one or several `DataOut` output ports on the primary TAP. These enable a secondary TAP when the corresponding `DataOut` port is asserted:

```
set DFT [create_dft_specification]
read_config_data -in $DFT -from_string {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck;
      }
      Tap(primary) {
        HostIjtag(1) {
        }
        DataOutPorts {
          count : 1;
        }
      }
    }
  }
}
process_dft_specification
```

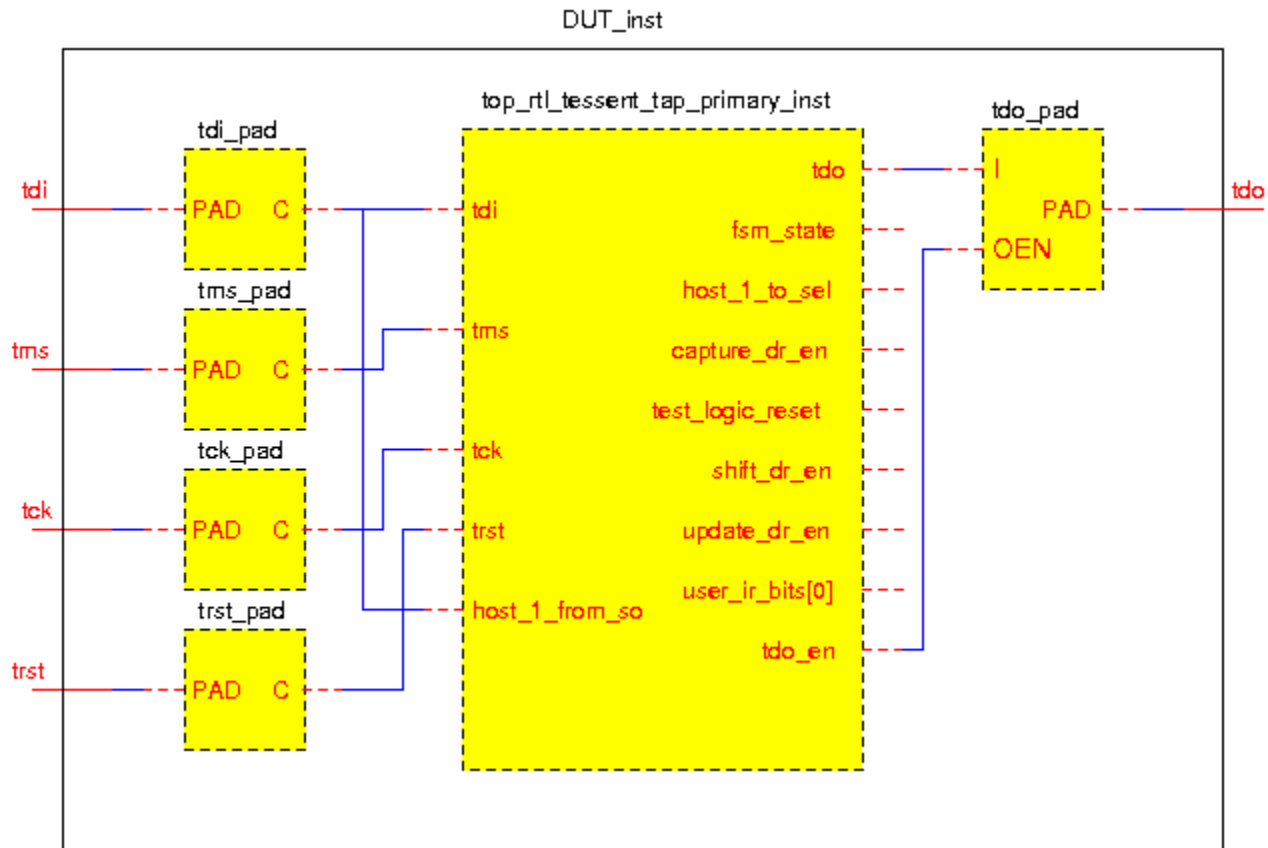
Discussion

Except for the new `user_ir_bits[0]` output port created using the `DataOutPorts` wrapper, this TAP is functionally identical to a stand-alone TAP with a host scan interface.

The added `DataOutPort` ports are TAP IR bits. If you need to create these bits as DR bits, insert a TDR on the existing host scan interface using the following command:

```
create_dft_specification -existing_ijtag_host_scan_in hierarchical_pin
```

The following is a schematic representation of this example:



Related Topics

[Insert a Stand-Alone TAP in a Design](#)

[Insert a Secondary TAP](#)

Insert a Secondary TAP

This example generates a primary TAP, a 2-to-1 scan mux, and a secondary TAP. The primary TAP is enabled by default.

Solution

1. Set the design level to “chip.”

2. Use the following dofile example to insert the TAP:

```
set DFT [create_dft_specification]
read_config_data -in $DFT -from_string {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck;
      }
      Tap(primary) {
        HostIjtag(0) {
        }
        DataOutPorts {
          count : 1 ;
        }
      }
      ScanMux(1) {
        select : Tap(primary)/DataOut(0) ;
        Input(1) {
          Tap(secondary) {
            HostIjtag(1) {
            }
          }
        }
      }
    }
  }
}
process_dft_specification
```

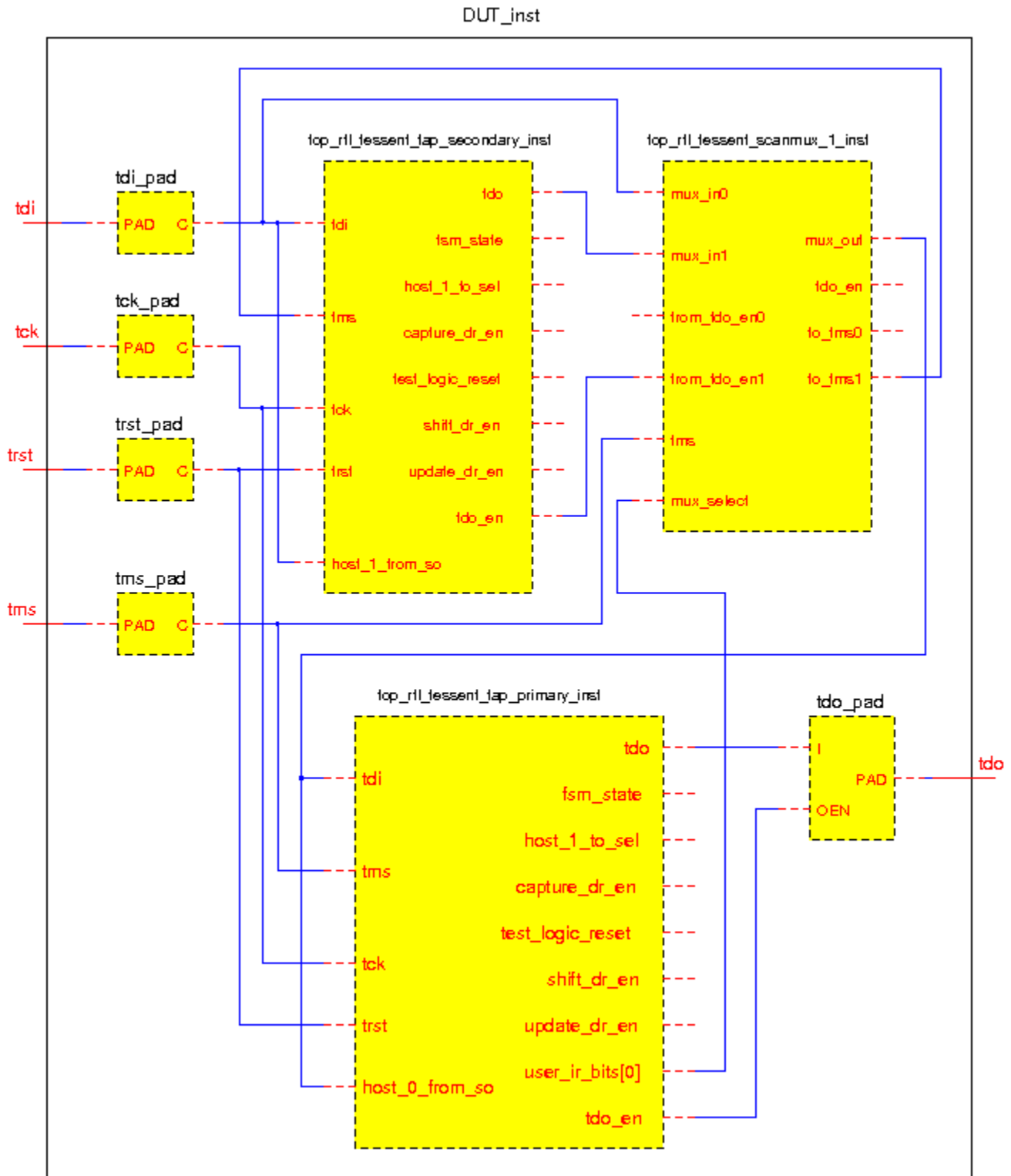
Discussion

To trace through the TAP secondary implementations, begin with the TDO output and trace back toward the primary TDI input.

The inserted ScanMux selects the TDI input by default and routes it to its own mux_out output.

The mux_select input comes from the primary TAP's user_ir_bits[0] output. When this output transitions to 1, the secondary TAP is enabled and the complete JTAG scan path goes through both TAPs.

The following is a schematic representation of this example:



Related Topics

[Insert a Stand-Alone TAP in a Design](#)

[Insert a Primary TAP](#)

Connecting to a Third-Party TAP

Connecting an IJTAG network or instruments to a third-party TAP requires reading the Verilog module of that TAP along with its ICL before setting the current design in Tessent Shell.

Solution

When creating the DFT specification, specify the `-existing_ijtag_host_scan_in` port option and point to the ScanInPort on the third-party TAP that receives the scanned-out data from the inserted IJTAG network or instruments. The DFT specification is then created accordingly.

Related Topics

[Insert a Stand-Alone TAP in a Design](#)

How to Set Up Third-Party Synthesis

You must synthesize DFT logic generated in the Tessent Shell flow in order to perform scan insertion, test pattern generation, and other processes.

Solution

You must define constraints in synthesis tools to enable accurate gate level representations of the RTL. These constraints are primarily made up of clock definitions and timing constraints. Use the following procedures to set up constraints for Synopsys and Cadence synthesis tools within the Tessent Shell flow.

Synopsys

The following procedure provides a high-level overview of the synthesis flow for Synopsys. For complete details, refer to “[Example Design Compiler Synthesis Script](#)” on page 760 for an example.

1. Source the SDC file generated by the tool.

The generated SDC file is located in the TSDB of the current design under the design ID to which the DFT was inserted and the ICL was extracted.

2. Set and redefine the Tessent Tcl variables.

The SDC file includes variables that need to be set in order for the script to operate correctly. For example, the period of the TCK clock needs to be set. Depending on your

DFT flow, you may need to define additional variables. The variables that you must set are identified in the SDC script file.

3. Verify the declaration of the functional clocks.


You must define all functional clocks. The tool automatically generates clock definitions based on the information from your DFT insertion script. You must review these definitions for accuracy.

4. Redefine other Tessent Tcl variables.

You must review and customize any additional variables. For example, a variable must be set that defines the input delay for the primary pins. This setting is based on the pre-defined TCK period and a custom multiplier value. The following is an example:

```
set tessent_input_delay [expr 0.3 * $tessent_tck_period]
```

Note

 The value of `tessent_tck_period` might depend on the maximum tck clock frequency that can be applied to the circuit. See [“IJTAG Network Performance Optimization”](#) in the *Tessent IJTAG User’s Manual* showing how to maximize the frequency of the IJTAG network test clock.

5. Load the design into your synthesis tool.

The Tessent Shell tool automatically creates a script to import your design. The tool then sources the generated script to elaborate the design.

6. Apply the SDC constraints.

At this stage of the flow, the SDC constraints for the DFT logic are applied by sourcing the appropriate procedure. The procedures are described under the [“extract_sdc”](#) command of the *Tessent Shell Reference Manual*.

7. Prepare the DFT logic for synthesis.

You must set up the DFT logic by configuring synthesis tool settings to ensure proper synthesis. For example, some instances need to be preserved during synthesis to ensure that the gate-level analysis functions correctly. The following statements are examples of configuration settings:

```
set_app_var  
compile_enable_constant_propagation_with_no_boundary_opt false  
set preserve_instances [tessent_get_preserve_instances icl_extract]  
set_boundary_optimization $preserve_instances false  
set_ungroup $preserve_instances false  
set_app_var compile_seqmap_propagate_high_effort false  
set_app_var compile_delete_unloaded_sequential_cells false  
set_boundary_optimization [tessent_get_optimize_instances] true  
set_size_only -all_instances [tessent_get_size_only_instances]
```

8. Synthesize your design.

The synthesis script compiles the design to create a gate-level representation of the RTL.

9. Write out the SDC and the final gate-level netlist.

The final SDC is a combination of the functional and the DFT constraints.

Genus

For synthesis with the Genus tool, follow the procedure for the Synopsys tool, but ensure that you adjust steps 3 through 9 to match the script shown in “[Example Genus Synthesis Script](#)” on page 761. These steps are different because the tools use different commands to run synthesis. Otherwise, the flow and results are the same.

How to Set Up Support for Third-Party OCCs

Tessent tools provide automation that enables you to insert Tessent OCCs as well as define and configure them for ATPG. If the design contains third-party OCCs, Tessent tools can set up and operate the OCC throughout the flow by reading and processing files that describe the operation of the OCC.

How to Configure Files for Third-Party OCCs	520
Test Logic Insertion	521
Configuration for Scan Insertion	523
Pattern Generation and Simulation	523

How to Configure Files for Third-Party OCCs

In order to automate the flow as much as possible, the following files set up and control how the tool interacts with the OCC.

Solution

You must place the ICL and PDL files in the same directory as the Verilog of the OCC using the same file base name. For this example, the OCC Verilog module is in a file named *third_party_occ.v*.

Moving the ICL and PDL files into the same directory as the Verilog enables the tool to automatically load them and carry the information throughout the flow.

You must prepare the following files and definitions as described:

- **ICL** — Provide an ICL file based on the IEEE 1687 IJTAG standard to describe the ports on the OCC that need to be controlled by IJTAG during test. Name the file *third_party_occ.icl* and place it in the same directory as the Verilog file.
- **PDL** — Provide a PDL file based on the IEEE 1687 IJTAG standard to describe the procedures that configure the third-party OCC. Name the file *third_party_occ.pdl* and place it in the same directory as the Verilog file.
- **TCD Scan** — Provide a TCD Scan file based on the Tessent Core description language to describe the OCC's programming shift register (chain segment) that needs to be connected to the design's scan chains during scan insertion.

The following is an example TCD Scan file for a third-party OCC. The sub-chain port information must be included in the ScanChain wrapper and in the Clock and ScanEn wrappers to define the polarity of each port.


```

Core(third_party_occ) {
  Scan {
    module_type           : occ;
    allow_scan_out_retiming : 0;
    is_hard_module        : 1;
    traceable             : 0;
    pre_scan_drc_on_boundary_only : 1;
    Clock(slow_clock) {
      off_state : 1'b0;
    }
    ClockOut(clock_out_mux/y) {
      slow_clock_input : slow_clock;
      fast_clock_input : fast_clock;
    }
    ScanEn(scan_en) {
      active_polarity : 1'b1;
    }
    ScanChain {
      length       : 4;
      scan_in_port  : scan_in;
      scan_out_port : scan_out;
      scan_in_clock : slow_clock;
      scan_out_clock : ~slow_clock;
    }
  }
}

```

- **Clock Control Definitions** — Provide a test procedure file that contains Clock Control Definitions (CCDs) for the OCC instances in the design. For details on the format of CCDs, refer to “[Clock Control Definition](#)” on page 567.

Test Logic Insertion

During test logic insertion (for example, EDT), the tool reads and processes the ICL and PDL files and includes this information in design files stored in the TSDB. Additionally, any OCC ports described in the ICL file that need to be controlled by IJTAG are connected to the generated IJTAG network such that the OCCs can be easily configured in subsequent steps.

See “[How to Configure Files for Third-Party OCCs](#)” on page 520 for instructions on setting up the ICL and PDL files.

Solution

EDT Example

The following EDT example uses a post-insertion procedure to connect the `slow_clock` port of the OCC to the newly-generated `shift_capture_clock` DFT signal. This enables the same clock source to be used for the OCC and EDT logic. The post-insertion script is executed after the `process_dft_specification` command creates all other logic defined in the DFT Specification. For detailed information on how to create and use a post-insertion procedure, refer to “[process_dft_specification.post_insertion](#)” in the *Tessent Shell Reference Manual*.

```
set_context dft -rtl
set_tsdb_output_directory ../tsdb_outdir

# Read the design and OCC module. The ICL and PDL files with the
# same base name are automatically read from the same directory
read_verilog ../rtl/RDS_process_with_occ.v
read_verilog ../rtl/third_party_occ.v

set_current_design RDS_process
set_design_level physical_block

# Define DFT Signals for EDT and scan test.
add_dft_signals scan_en edt_update test_clock -source_node \
  {scan_en_r edt_update test_clock_r}
add_dft_signals edt_clock shift_capture_clock -create_from_other_signals
set_dft_specification_requirements -logic_test on
add_clocks clock1 -period 3ns
add_clocks clock2 -period 4ns

check_design_rules

# Create the DFT Spec for adding EDT
set_spec [create_dft_specification -sri_sib_list {edt} ]
read_config_data -in $spec -from_string {
  EDT {
    ijtag_host_interface : Sib(edt);
    Controller (c1) {
      longest_chain_range : 50, 65;
      scan_chain_count : 60;
      input_channel_count : 2;
      output_channel_count : 2;
    }
  }
}

# Post-insertion procedure to change the existing connection to
# third-party OCC's slow_clock port and connect to the shift_capture_clock
# DFT Signal
proc process_dft_specification.post_insertion {RDS_process args} {
  set occ_insts [get_instances -of_module third_party_occ -silent]
  if {[sizeof_collection $occ_insts] > 0} {
    set slow_clock_pins [get_pins slow_clock -of_instances $occ_insts]
    foreach_in_collection occ_pin $slow_clock_pins {
      delete_connection $occ_pin
      create_connection [get_dft_signal shift_capture_clock] $occ_pin
    }
  }
}
process_dft_specification
extract_icl

# Create a synthesis script of all RTL (original and newly created)
write_design_import_script use_in_synthesis.tcl -use_relative_path_to . \
-replace
exit
```

Configuration for Scan Insertion

You must configure the tool to read the OCC files before running scan insertion. The Tessent OCC provides independent control of each clock domain in your design in the context of DFT.

See “[How to Configure Files for Third-Party OCCs](#)” on page 520 for instructions on setting up the OCC files.

Solution

When inserting scan into the post-synthesis netlist, you must specify the location of the TCD Scan file using the `set_design_sources` command. The tool uses the description of the OCCs’ scan segment to stitch them into the design’s scan chains.

The following is an example:

```
set_context dft -scan
set_tsdb_output_directory ../tsdb_outdir

# Specify the location of the TCD Scan file that describes the OCC's scan
# segment
set_design_sources -format tcd_scan -Y ../rtl -extension tcd_scan

# Read the cell library and synthesized netlist from DC Shell
read_cell_library ../library/tessent/adk.tcelllib
read_verilog ../2.synthesis/RDS_process_synthesized.vg

# Use read_design to read in information (DFT signals, etc.) performed in
# previous pass
read_design RDS_process -design_identifier rtl -no_hdl
set_current_design RDS_process
set_system_mode analysis

# Add a scan mode and specify EDT instances to which scan chains should be
# connected
set edt_instance [get_instances -of_icl_instances [get_icl_instances \
    -filter tessent_instrument_type==mentor::edt]]
add_scan_mode edt -edt_instance $edt_instance
analyze_scan_chains
insert_test_logic
exit
```

Pattern Generation and Simulation

You must configure the tool to read the OCC files before generating patterns. The Tessent OCC provides independent control of each clock domain in your design in the context of DFT.

See “[How to Configure Files for Third-Party OCCs](#)” on page 520 for instructions on setting up the OCC files.

Solution

Pattern Configuration

For stuck-at ATPG, much of the setup is automatically imported using the `import_scan_mode` command. Additional clock definitions and settings for the OCC should be provided for the OCC's asynchronous source clocks as well as the clocks on the output of the OCC instances.

```
set_context patterns -scan
set_tsdb_output_directory ../tsdb_outdir

# Read the cell library and design
read_cell_library ../library/tessent/adk.tcelllib
read_design RDS_process -design_id gate
set_current_design RDS_process

# Specify a test mode for stuck-at ATPG
set_current_mode edt_stuck

# Import scan, EDT, and clock configuration for ATPG
import_scan_mode edt

# Define external and OCC clocks
set_clock_options test_clock_r -pulse_in_capture on
set_occ_insts [get_instances -of_module third_party_occ* -silent]
if {[sizeof_collection $occ_insts] > 0} {
    foreach_in_collection occ_inst $occ_insts {
        set inst_name [get_single_name $occ_inst]
        add_clocks [get_pins ${inst_name}/clock_out_mux/y] -capture_only
    }
}
# Execute iProc for third_party OCC to set default values (test_mode = 1)
set_test_setup_icall "${inst_name}.setup" -append
}
set_system_mode analysis

# Specify a procedure file containing clock control definition for the OCC
# instances
read_procfiler third_party_occ_clock_controls.proc
create_patterns
write_tsdb_data -replace
write_patterns patterns/RDS_process_stuck_parallel.v -verilog -parallel \
    -replace -parameter_list {SIM_KEEP_PATH 1}
set_pattern_filtering -sample_per_type 2
write_patterns patterns/RDS_process_stuck_serial.v -verilog -serial \
    -replace -parameter_list {SIM_KEEP_PATH 1}
```

In the ANALYSIS mode, specify a procedure file that contains the clock control definitions for the OCC instances.

For transition fault ATPG, a number of changes to the dofile are highlighted in the following example. Define any additional internal clocks that are needed for the third-party OCC, similar to those defined in the example (for example, if the OCC internally gates the fast clock during transition test).

Additionally, specify any parameters to the OCC's iCalls that must be set to configure the OCC for transition/fast-capture test.

You must constrain the design's I/Os that are not used for transition test.

```
set_context patterns -scan
set_tsdb_output_directory ../tsdb_outdir

# Read the cell library and design
read_cell_library ../library/tessent/adk.tcelllib
read_design RDS_process -design_id gate
set_current_design RDS_process

# Specify a test mode for transition ATPG
set_current_mode edt_transition

# Import scan, EDT, and clock configuration for ATPG
import_scan_mode edt
import_clocks

# Define external and OCC clocks
set_clock_options test_clock_r -pulse_in_capture on
set_occ_insts [get_instances -of_module third_party_occ* -silent]
if {[sizeof_collection $occ_insts] > 0} {
    foreach_in_collection occ_inst $occ_insts {
        set_inst_name [get_single_name $occ_inst]
        add_clocks [get_pins ${inst_name}/clock_out_mux/y] -capture_only
        add_clocks \
            [get_pins ${inst_name}/occ_control/cgc_SHIFT_REG_CLK/clkg] \
            -pulse_in_capture
    }
}
# Execute iProc for third_party OCC and enable fast capture mode for OCCs
set_test_setup_icall "${inst_name}.setup fast_capture_mode 1" \
    -append
}

add_input_constraints -hold
add_output_masks -all

set_system_mode analysis

# Specify a procedure file containing clock control definition for the OCC
# instances
read_procf file third_party_occ_clock_controls.proc

set_fault_type transition
set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]

create_patterns
write_tsdb_data -replace
write_patterns patterns/RDS_process_transition_parallel.v -verilog \
    -parallel -replace -parameter_list {SIM_KEEP_PATH 1}
set_pattern_filtering -sample_per_type 2
write_patterns patterns/RDS_process_transition_serial.v -verilog \
    -serial -replace -parameter_list {SIM_KEEP_PATH 1}
```

Pattern Simulation

You must run a Verilog simulation of the generated patterns to ensure no mismatches are reported and that the patterns function as expected during the tester application.

For parallel load patterns specified by the “write_patterns -parallel” command, you simulate all the patterns.

For serial load patterns, a handful of patterns are sufficient because the run time for simulating serial load patterns can be significant.

Post-Synthesis Update

Mapping complex ports during synthesis may cause name and footprint changes in the design.

ICL and TCD Post-Synthesis Update	527
Limitations Related to SystemVerilog Interface Arrays	528
Updating ICL Attributes From the Design	529
Matching Requirements for Port Names in Post-Synthesis Update	530
Design Name Mapping Commands	531

ICL and TCD Post-Synthesis Update

After logic synthesis with Design Compiler, Genus, or Oasys, ports may be flattened or bit-blasted, causing design footprint changes. Post-synthesis names differ from those in the initial RTL.

Tessent in no RTL mode uses the post-synthesis netlist when you run `set_current_design`. The ICL and TCD of the current design are updated automatically to match the new design objects loaded from the post-synthesis netlist.

The following ICL model attributes are updated to allow smooth binding of the ICL objects and design objects later in the flow:

- `tessent_design_instance` (for the instance path)
- `tessent_design_gate_ports` (for ports)
- `tessent_design_rtl_ports` (for ports)

The `tessent_design_instance` attribute of the ICL instance object is automatically updated when any of the following commands are invoked:

- `set_current_design`
- `write_design`
- `update_icl_attributes_from_design`

The `tessent_design_gate_ports` attribute of the ICL port object is automatically updated when any of the following commands are invoked:

- `set_current_design`
- `write_design`
- `get_ports -of_icl_ports`
- `get_pins -of_icl_pins`

- `get_pins -of_icl_ports`
- `update_icl_attributes_from_design`

For ICL ports and instances, the ICL matching performed by `set_current_design` is insufficient if the current ICL model does not include the complete ICL of child instances under the physical blocks. In these cases, the ICL matching is completed later in the flow using the following commands:

- `get_ports -of_icl_ports`
- `get_pins -of_icl_pins`
- `get_pins -of_icl_ports`
- `update_icl_attributes_from_design`

Use the `-use_path_matching_options` option with `get_pins`, `get_ports`, or `get_instances` to match a name pattern to the ports and pins in the current design when the TCD of instruments dumped during RTL mode is loaded in no RTL flow.

Limitations Related to SystemVerilog Interface Arrays

Designs using SystemVerilog interface arrays synthesized with Design Compiler require special handling.

If your design contains ports declared using SystemVerilog interface arrays, and your netlist is created with Synopsys Design Compiler, the syntax for these declarations should be restricted as follows:

Use the following syntax to declare the SystemVerilog interface ports array when creating your RTL design. The packed range must be from 0 to a positive right index. Without this syntax, Tessent Shell cannot match and automatically update ICL objects and TCD models post-synthesis with Design Compiler.

```
<SV interface type> <port name> [0:<positive right index>]
```

For example, if your design has an interface type named `intf1` and a port named `p1`, the port declaration in the design file should be written so that the range is restricted:

```
intf1 p1 [0:<positive_index>];
```

The left index must be 0 and the right index must be positive.

Updating ICL Attributes From the Design

The `update_icl_attributes_from_design` command updates ICL port, instance, and module attributes for the gate views of a design. All ICL ports, instances, and modules in the current design are updated.

The following attributes are updated:

- **tessent_design_gate_ports** — This attribute refers to the design port corresponding to the ICL port. The attribute value is the name of the design port as it appears in the netlist.
- **tessent_design_instance** — This attribute refers to the design instance corresponding to the ICL instance. The value is the hierarchical name relative to the parent ICL instance.
- **tessent_design_rtl_gate_port_mapping** — This attribute maps the RTL name to the netlist name for those ports which appear in the following attribute name lists:
 - `forced_high_input_port_list`
 - `forced_low_input_port_list`
 - `forced_high_output_port_list`
 - `forced_low_output_port_list`
 - `forced_high_internal_input_port_list`
 - `forced_low_internal_input_port_list`
 - `tessent_ground_port_list`
 - `tessent_power_port_list`
 - `tessent_clock_domain_labels`

The attribute value is a list of names. The names at the even index positions in the list are the RTL names of the ports in the above name lists, and the names at the odd index positions are the netlist names of those ports.

The `update_icl_attributes_from_design` command has one Boolean option: `-verbose`. This option allows warnings to be issued when a port name cannot be found or matched in the netlist. The command is invoked automatically from the `write_design -tsdb` command. It is also invoked automatically as part of the `insert_test_logic` command just after insertion.

Related Topics

[update_icl_attributes_from_design](#)

Matching Requirements for Port Names in Post-Synthesis Update

The following matching rules are supported when looking up a port name in a netlist:

1. Support the transformation performed by dc_shell with “change_names -rules verilog” to remove the escaping and replace special characters with underscores (“_”). A single underscore matches multiple underscores.
2. Support the transformation performed by Genus to get the gate name from the RTL name. All strings and numerical indices in the RTL name are preserved. Only the delimiters are changed.
3. Support the Genus transformation described in rule #2 but with escaping removed and special characters replaced with an underscore (“_”) as in rule #1.
4. Support bit-blasted escaped names generated by a layout tool. The bit select is incorporated into the escaped name.
5. Support user specified matching rules.

Design Name Mapping Commands

The `add_rtl_to_gates_mapping`, `report_rtl_to_gates_mapping`, and `delete_rtl_to_gates_mapping` commands have several features to help you control design name mapping.

Design Name Mapping	531
Default Matching Rules for the <code>get_pins</code>, <code>get_ports</code>, and <code>get_instances -match_rtl_reg</code> Commands	531

Design Name Mapping

Use the `add_rtl_to_gates_mapping`, `report_rtl_to_gates_mapping`, and `delete_rtl_to_gates_mapping` commands to control design name mapping.

You define custom name mapping rules for RTL to post-synthesis or post-layout name mapping with the `add_rtl_to_gates_mapping` command. You can specify substrings, prefixes, and suffixes in addition to basic mapping rules. Use this command if the matching rules described in “Default Matching Rules for the `get_pins`, `get_ports`, and `get_instances -match_rtl_reg` Commands” on page 531 are not sufficient for your application.

Related Topics

[report_rtl_to_gates_mapping](#)

[delete_rtl_to_gates_mapping](#)


Default Matching Rules for the `get_pins`, `get_ports`, and `get_instances -match_rtl_reg` Commands

The tool provides several default rules.

- **Component Names** — All strings between non-escaped delimiters (component names) in the name to be looked up (lookup name) must match the corresponding strings in a netlist name unless they contain special characters. The recognized delimiters are periods (“.”), forward slashes (“/”), brackets (“[]”), and underscores (“_”).

There are two types of component names: subnames and indices. Any of the above characters can delimit subnames. Indices can be delimited only by brackets or underscores. A right bracket must follow a component name that is preceded by a left bracket.

Note

 Negative indices are not supported.

An example lookup name:

```
abc.def
```

The “abc” string in the name to be looked up must exactly match the string before the first delimiter in a netlist name. The “def” string in the name to be looked up must exactly match the string after the last delimiter in the same netlist name.

- **Escaping** — Escape characters are not matched. They are only used to direct the matching procedure.
- **Delimiters** — Delimiters in the lookup name are used only to extract the component names.

All component names after the first are assumed to have a leading delimiter and optionally a trailing delimiter in a netlist name. There are two cases to consider when matching the delimiters for a component name in the netlist:

a. The port name in the netlist is escaped.

Possible matches for component name delimiters are as follows:

- Leading and trailing delimiters of brackets (“[]”).
- Leading delimiter of underscore (“_”) and trailing delimiter of underscore or null.
- Leading delimiter of period (“.”) or slash (“/”) and trailing delimiter of null.


b. The port name in the netlist is not escaped.

Possible matches for component name delimiters are as follows:

- Leading delimiter of underscore and trailing delimiter of underscore or null.

- **Special Characters in the Lookup Name** — When matching to a non-escaped name in the netlist, any characters not allowed by the language without escaping in the lookup name are replaced with the underscore character (“_”). Matching allows truncation in the netlist name to two trailing underscores.

Note

 This truncation rule applies only to underscore characters derived from special characters, not delimiters.

- **Bit Select Lookup Name** — A bit select lookup name (last component name is an index) can match a one-dimensional vector with the final bit select applied to the match or match directly to a bit select.

Chapter 10

Test Procedure File

Test procedure files specify how the scan circuitry within a design operates. The scan circuitry operation is specified using previously-defined scan clocks and other control signals. To operate the scan circuitry in your design, you must define the scan circuitry and provide a test procedure file to describe its operation. The design rules checking (DRC) process, which occurs when you exit setup mode, performs extensive checking to ensure the scan circuitry operates correctly.

Test Procedure File Creation	534
Test Procedure File Syntax	534
Test Procedure File Structure	539
#include Statement	539
Set Statement	540
Alias Definition	543
Timing Variables	545
Timeplate Definition	548
Multiple-Pulse Clocks	554
Always Block	556
Procedure Definition	557
Clock Control Definition	567
The Procedures	576
Test_Setup (Optional)	577
Shift (Required)	580
Alternate Shift Procedure (Optional)	582
Load_Unload (Required)	583
Shadow_Control (Optional)	586
Master_Observe (Sometimes Required)	588
Shadow_Observe (Optional)	588
Skew_Load (Optional)	589
Clock_run (Optional)	591
Capture Procedures (Optional)	593
Clock_po (Optional)	598
Clock_sequential (Optional)	599
Init_force (Optional)	599
Test_end (Optional, all ATPG tools)	600
Sub_procedure	602
Additional Support for Test Procedure Files	604
Creating Test Procedure Files for End Measure Mode	605
Serial Register Load and Unload for LogicBIST and ATPG	609
Register Load and Unload Use Models	609

Static Versus Dynamic Register Variables	609
Test Procedure File Modifications	610
Dofile Modifications	613
Serial Load and Unload DRC Rules	616
Notes About Using the stil2mgc Tool	622
Extraction of Strobe Timing Information from STIL (SPF)	622
The STIL ClockStructures Block	622
Test Procedure File Commands and Output Formats	623

Test Procedure File Creation

You insert scan circuitry and create test procedure files for ATPG operations using the “patterns -scan” context of Tessent Shell. If your design already contains scan circuitry, you need to create a test procedure file to describe its operation either by hand or with Tessent Shell.

You can specify a test procedure file in setup mode using the `add_scan_groups` command. The tools can also read in procedure files by using the `read_procfile` command or the `write_patterns` command when not in setup mode. When you load more than one test procedure file, the tool merges the timing and procedure data.

You can also have the `stil2mgc` tool translate a STIL Procedure File (SPF) into a dofile and test procedure file. This tool produces a dofile that defines clocks, scan chains, scan groups, and pin constraints. This tool also creates test procedure files with a timeplate and the following standard scan procedures: `test_setup`, `load_unload`, and `shift`. For more information about `stil2mgc`, refer to “[Notes About Using the stil2mgc Tool](#)” on page 622.

The following subsections describe the syntax and rules of test procedure files, give examples for the various types of scan architectures, and outline the checking that determines whether the circuitry is operating correctly.

Test Procedure File Syntax

The test procedure file uses common syntactical conventions such as bold and italic fonts, and reserved characters. The file supports Tcl conditional statements.

Syntax Conventions

The following syntax conventions are used in this chapter:

- **Bold** — Indicates a keyword. Enter the keyword exactly as shown.
- *Italic* — Indicates lexical elements such as identifiers, strings, or numbers. Replace the italicized word with the appropriate name or integer.

- | — A vertical bar or pipe character indicates a logical “OR” as in “select foo OR foo_not”.
- [] — Square brackets indicate optional elements. Do not include the brackets.
- ... — An ellipsis indicates a repeatable item or set.

Reserved Characters

If you have a pin or pathname that uses a reserved punctuation character, you must enclose that name in double quotes. See [Table 10-1](#) for a list of reserved punctuation characters.

For example, the following statement is illegal because it uses the exclamation point outside of double quotes.

```
force /inst_my_adder_1/xclk_header!x1!x1/op1[9] 1
```

The signal name contains a reserved punctuation character, the exclamation point (!), so it must be enclosed inside double quotes. The correct syntax would be:

```
force "/inst_my_adder_1/xclk_header!x1!x1/op1[9]" 1
```

Table 10-1. Reserved Punctuation Characters

Name	Character
Ampersand/AND	&
Caret/Circumflex/XOR	^
Comma	,
Equals	=
Exclamation mark	!
Left/Opening brace	{
Left/Opening parenthesis	(
Right/Closing brace	}
Right/Closing parenthesis)
Semicolon	;
Vertical bar/OR	

Throughout this chapter, value = 0, 1, X, or Z.

Using Tcl in the Test Procedure File

Procedure files support the Tcl conditional statements “if”, “else”, and “elseif” using the following syntax:

```
if { tcl_expr } {  
    procedure file statements  
}  
elseif { tcl_expr } {  
    procedure file statements  
}  
else {  
    procedure file statements  
}
```

Where a “tcl_expr” is any Boolean Tcl expression that uses Tcl variables, dofile variables, or environment variables. Just as when doing variable substitution in the procedure file, other Tcl statements and defining Tcl variables are not supported. All variables must be defined in the dofile or from the shell as environment variables.

The body of these Tcl conditional statements should contain only legal procedure file syntax, not any other Tcl statements. The Tcl conditional statements are treated as preprocessor statements in the procedure file parser. They are not preserved in the tool after parsing is finished; only the procedure file code selected by the evaluation of the Tcl “if” expression is stored in the tool. Therefore, when using `write_profile` to write out the procedure file, none of the Tcl conditional statements are present, and the procedure file code not used is also not present. For more information refer to “[The Tessent Tcl Interface](#)” on page 771.

Introductory Test Procedure File Example

The following is an example of a test procedure file.


```

// Comments use "//" characters
//
// Set the base time increment for use in all timeplates
set time scale 1.0 ns;

// Define the strobe time for the measure statements
set strobe_window time 1;

// This design uses a single timeplate, named "tp1", for all
// vectors.

timeplate tp1 =
    force_pi 0;
    measure_po 1;
    pulse CLK0_7 2 1;
    pulse CLK8_15 2 1;
    period 4;
end;

// The shift and load_unload procedures define how the design
// must be configured to allow shifting data through the scan
// chains. The procedures define the timeplate to use
// and the scan group that it references.

procedure shift =
    scan_group grp1;
    timeplate tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse CLK8_15;
        pulse CLK0_7;
    end;
end;

procedure load_unload =
    scan_group grp1;
    timeplate tp1;
    cycle=
        force CLEAR 0;
        force CLK0_7 0;
        force CLK8_15 0;
        force scen1 1;
    end;
    apply shift 8;
end;

// The capture procedure is a "non-scan" procedure. This
// procedure describes the timeplate to use for the
// capture cycle. It also defines the number of cycles
// to use in the capture cycle. In this example there is
// just one cycle.

procedure capture =
    timeplate tp1;
    cycle =
        force_pi;
        measure_po;

```

```
        pulse_capture_clock;  
    end;  
end;
```

Test Procedure File Structure

The test procedure file consists of many structural elements that display in a specific order, starting with #include statements. Some of these elements are required and others are optional.

```
#include "<file_name>";
[set_statement ...]
[alias_definition ...]
[timing_variables ...]
timeplate_definition           // includes pulse clock statements
always_block
procedure_definition
clock control definition
```

#include Statement	539
Set Statement	540
Alias Definition	543
Timing Variables	545
Timeplate Definition	548
Multiple-Pulse Clocks	554
Always Block	556
Procedure Definition	557
Clock Control Definition	567

#include Statement

The “#include” statement specifies that the tool read test procedure data from a specified file.

The following rules apply to #include statements and files:

- The “#include” statement can occur anywhere in the file, and multiple “#include” statements can occur in one file. For example:

```
#include "foo.proc";
```
- The filename to be included must be enclosed in double quotes, and the statement must be followed by a semicolon.
- All timeplates and procedure rules apply to the statements placed in #include files.
- Included files can use the “#include” statement to include other files, up to a maximum include depth of 512. If you later use the write_procfile command write out procedure data, the “#include” statements are not preserved, and the tool writes all procedure data to a single file.

Set Statement

The Set statements define specific parameters used throughout the test procedure file.

The following statements are available:

```
set time scale tscale;  
set strobe_window time window_width;  
set default_timeplate timeplate_name;  
set autoforce off;
```

set time scale Statement

Defines the time scale and unit. The “set time scale” statement must be at the beginning of the procedure file, before any timeplate or procedure definition. If you do not specify the time scale, the default value is 1 ns.

The tool applies the time scale and unit to the test procedure file and timeplates. The *tscale* you specify can be any real number. Time values in the timeplate, however, must be integers, representing whole time scale units. If you find you are specifying fractional times in the timeplate, you must reduce the time scale unit so you can specify integer time values in the timeplate. For example, the following would result in a syntax error:

```
set time scale 1 ns ;  
set strobe_window time 1 ;  
  
timeplate fast_clk_tp =  
    force_pi 0 ;  
    measure_po 0.500 ;  
    pulse CLKA 0.750 1.50 ;  
    pulse CLKB 0.750 1.50 ;  
    period 3.000 ;  
end ;
```

To correct the syntax, you could change the time scale to picoseconds, and adjust the time value to meet the scale as follows:

```
set time scale 10 ps ;  
set strobe_window time 1 ;  
  
timeplate fast_clk_tp =  
    force_pi 0 ;  
    measure_po 50 ;  
    pulse CLKA 75 150 ;  
    pulse CLKB 75 150 ;  
    period 300 ;  
end ;
```

The units supported are ms, us, ns, ps, and fs.

The tool translates the time scale in the procedure file into a Verilog ‘timescale directive in the Verilog test bench when writing patterns in Verilog format.

If the time scale number you specify in the test procedure file is 1 or larger, the resulting Verilog `'timescale` directive has the same time unit (resolution) and time precision. For example, “set time scale 1 ns ;” would result in this Verilog directive:

```
'timescale 1ns / 1ns
```

If you want the test bench to have smaller precision than resolution, there are several ways to designate this:

- Specify a time scale number of less than 1 in the procedure file. For example, “set time scale 0.5 ns ;” produces this Verilog directive:

```
'timescale 1ns / 100ps
```

- Add non-zero significant bits to the time scale in the procedure file. For example, “set time scale 10.05 ns ;” produces this Verilog directive:

```
'timescale 1ns / 10ps
```

- Add trailing zeros as significant bits for an asynchronous clock period or `pattern_set` period (when creating IJTAG patterns). For example, an “add_clocks -period 10.00ns” command produces this Verilog directive:

```
'timescale 1ns / 10ps
```

- Use the `SIM_PRECISION` parameter file keyword. For example, “SIM_PRECISION 0.5ns;” produces this Verilog directive:

```
'timescale 1ns / 100ps
```

The precision in the Verilog test bench can originate from any of the previous sources, and the tool uses the smallest specified precision when writing out the Verilog test bench.

The resolution in the Verilog test bench originates from the procedure file. When you use multiple procedure files, the various “set time scale” statements can specify different values, and the tool uses the smallest specified resolution when writing the Verilog test bench.

set_strobe_window_time Statement

Defines the strobe-window width. The “set_strobe_window_time” statement must be at the beginning of the procedure file, before any timeplate or procedure definition. If you do not specify the `strobe_window_time`, it defaults to the maximum allowable size. For example, if you look at a timeplate, and if there is a 10 ns window between the `measure_po` event and the next event (or end of timeplate), then that is the size of the strobe window. If there are multiple timeplates, then the smallest strobe window from the timeplates is the maximum allowable strobe window.

Some tester formats measure primary outputs (POs) at the exact time that you specify with the `measure_po` statement in the timeplate. However, other tester formats, such as STIL, require

that output measurements occur during a specified window of time (*window_width*). For WGL, this statement changes the strobe window in the output file.

A strobe window can only stretch from the *measure_po* time to the end of the cycle or the next force or pulse event. For example, if you issue a *measure_po* at time 10 and the rising edge of a pulse at time 30, the strobe window can only be a maximum of 20. *Strobe_window* lets you know that, starting at the *measure_po* time, the primary output should be stable for the time specified by the strobe window.

Note



Strobe_window only affects the following formats: STIL, TSTL2, and WGL.

set default_timeplate Statement

Specifies a timeplate that can be used for any procedure definition that does not explicitly specify a timeplate. The referenced *timeplate_name* must be defined prior to the Set Default_timeplate statement in the procedure file.

set autoforce Statement

An optional statement that controls the behavior for automatically adding force events. If included, this statement must appear at the beginning of the procedure file, prior to any procedures.

By default (without this statement), if a constrained pin is not forced to the constrained value in the *test_setup* procedure, a force event is automatically added to the first cycle of the *test_setup* procedure. If the *test_setup* procedure starts with a call to a subprocedure, then the force event is added to the first cycle following the subprocedure. If a force event already exists in the *test_setup* procedure for a constrained pin, then no additional force event is added for that pin.

When you include the “set autoforce off” statement, the tool does not add any force events on the constrained pins to the test setup procedure.

Including the “set autoforce off” statement also prevents the tool from forcing clock pins to the inactive state at the beginning of the *load_unload* procedure. The statement also disables copying the last value forced on an input port in *test_setup* and prevents it from becoming a force at the start of the *load_unload* procedure.

The “set autoforce off;” statement also turns off auto forcing Z values on bidis in the *load_unload* procedure—see [Load_Unload \(Required\)](#).

Alias Definition

The Alias definition groups multiple signal names or cell paths into a single alias name. Signal Alias statements are useful in procedures or timeplates where multiple signals need to be assigned to the same value at the same time.

Cell Alias statements are used to group cell paths into a single alias name. You must define aliases before using them. The definition can occur at any place in the procedure file outside of a timeplate or procedure definition.

Note


 When saving STIL2005, CTL, or Structural_STIL patterns, all aliases defined in the procedure file are defined as SignalGroups in the resulting STIL file.

There is a predefined alias available for specifying all bidirectional pins. The “_ALL_BIDI” keyword may be useful for forcing all bidirectional pins to a specified value without having to identify each individual pin. For example:

```
force _ALL_BIDI Z;
```

In using a cell Alias statement to group cell paths from condition statements into a single alias name, it is possible to override a condition statement in a named capture procedure with a subsequent condition statement that occurs in the same place (global condition, or local to a specific cycle). A condition statement can only override a previous condition if the first condition is specified using an alias name, and if the second condition is specified without using an Alias statement.

Tip

 When using multiple named capture procedures where each procedure requires many condition statements, it is helpful to group cells into a common name and apply the condition statement once to the entire group of cells, and then override specific cells that need a different value than what was applied to the group. This frees you from having to enter numerous condition statements for each named capture procedure, while only a handful of the cells require different values for each procedure.

The Alias definition has the following format:

```
alias alias_name = pin_name [, pin_name ...];
```

or

```
alias alias_name = cell_name [, cell_name ...];
```

- ***alias_name***

A string that specifies the name of the alias.

- *pin_name*
A repeatable string that specifies the pin name to associate with the alias name.
- *cell_name*
A repeatable string that specifies the cell name to associate with the alias name.

Alias Examples

This example groups two signal names into a single alias name.

```
alias my_group = T, U;
```

This next example shows how a named capture procedure should look when using an Alias statement for condition cells. The example sets each of four cells to a value of 0, and then the fourth cell (/inst_3/blockb/reg_2/Q) is overridden with a value of 1.

```
alias cond_cells = "/inst_0/blocka/reg_1/Q", "/inst_1/blocka/reg_1/Q",  
                  "/inst_2/blocka/reg_1/Q", "/inst_3/blockb/reg_2/Q";  
timeplate tp1 =  
    force_pi 0;  
    measure_po 10;  
    pulse ref_clk 50 50;  
    period 100;  
end;  
procedure capture capture1 =  
    timeplate tp1;  
    condition cond_cells 0;  
    condition /inst_3/blockb/reg_2/Q 1;  
                                     // overrides condition in previous statement  
    cycle =  
        force_scan_en 0;  
        force_ctrl_a 1;  
        force_pi;  
        pulse ref_clk;  
    end;  
    cycle =  
        force_pi;  
        measure_po;  
        pulse ref_clk;  
    end;  
end;
```

This example shows how to define a user-defined bus in a procedure file:

```
alias wdata = "D[0]", "D[1]", "D[2]", "D[3]";
```


And this shows how to assign a value to that user-defined bus:

```
procedure sub_procedure subpro1 =
  scan_group grp1 ;
  timeplate shift_tp ;
  cycle =
    force wdata 0010 ;
    pulse ref_clk ;
  end;
end;
```

Timing Variables

Two timing variable block definitions allow a procedure file to express timing using variables and equations, and to have this equation-based timing preserved in the tool and reproduced in the correct syntax in pattern output files.

Test data languages such as WGL and STIL have the ability to express time values in the timing blocks as numerical values or as equations based on variables. Using equation-based timing enables one value to be specified for a global attribute, such as the test cycle period, while other values are derived from this using equations.

The two timing block definitions are called “timing_variables” and “variables”. In the “timing_variables” block, variables can be defined and assigned timing values. These values are expressed in the time scale which is already specified by the Set Time Scale statement. The “timing_variables” block must be defined before the timeplate definitions.

The “variables” block is used to define variables that are not time values and have no units associated with them. These variables can only be assigned integer numbers, and can be used as scaling multipliers in the timing equations.

The variables in the “timing_variables” block can also be assigned timing equations instead of time values. These equations can use either timing values or previously defined variables or timing variables as operands. When using timing equations, you must ensure that the final value has a valid time unit. When the tool parses the procedure file, timing variables that are computed to have an invalid time unit cause a [W42 DRC](#) error.

Note



The event statements in the timeplate definition block accept timing values and timing variables.

When saving patterns in the Verilog, WGL, and STIL supported formats, the waveform tables in these formats are written using the equations and variables, and the variables are defined in the appropriate definition blocks which exist in each format. When saving patterns in formats that don't support equation-based timing, the equations are computed and the timing information is specified as the resulting numeric values in the pattern file. Setting the ALL_FLATTEN_TIMING parameter file keyword to “1” causes Verilog, WGL, and STIL


outputs to compute the timing equations and use only the resulting numeric values in the output files. Any equation that does not compute to an integer value is rounded to the nearest integer value.

The “timing_variables” block has the following syntax:

```
variables =  
    variable_name = integer;  
    [variable_name = integer; ...]  
end;  
  
timing_variables =  
    variable_name = time_or_equation;  
    [variable_name = time_or_equation; ...]  
end;
```

Note that *time_or_equation* can either be an integer time value or a time equation. A time equation is expressed using operators and operands. An operator is one of +, -, *, or /. An operand can be a time value or a variable name (time or scaling variable). The multiplication and division operators (* and /) take precedence over the addition and subtraction operators (+ and -). You can use parenthesis to group operations for precedence.

Note

 In the timeplate definitions, any place where a time value can be used, a timing variable is also valid. A scaling variable from the “variables” block cannot be used in a timeplate definition. These can only be used in time equations.

Variable names can be any identifier except for reserved keywords used in the procedure file syntax (such as “period” and “force_pi”). The variable names must conform to the rules that apply to all identifiers used in the procedure file (alpha numeric string, starting with an alpha character, and no reserved punctuation marks). If reserved characters or reserved words are used in a variable name, the name must be enclosed in quotes.

Equation-Based Timing Example

The following is a partial example of using equation-based timing.

```

set time scale 1.0 ns;
variables =
    v_scale = 1;
end;

timing_variables =
    t_period = 100;
    t_force = 0;
    t_meas = ((t_period * 0.1) * v_scale );
    t_rise = ((t_period / 2) * v_scale );
    t_width = ((t_period * 0.2) * v_scale);
end;

timeplate tp1 =
    force_pi t_force;
    measure_po t_meas;
    pulse ref_clk t_rise t_width;
    period t_period;
end;

```

This is how the preceding timing example would be represented in the STIL output:

```

Spec STUCK_spec {
    Category STUCK_cat {
        v_scale = '1';
        t_period = '100ns';
        t_force = '0ns';
        t_meas = '(t_period*0.1)*v_scale';
        t_rise = '(t_period/2)*v_scale';
        t_width = '(t_period*0.2)*v_scale';
    }
}

Timing STUCK_timing {
    WaveformTable tset_tp1 {
        Period 't_period';
        Waveforms {
            input_time_gen_0 { 01 { 't_force' D/U; }}
            input_time_gen_1 { 01 { '0ns' D; 't_rise' D/U;
                                   't_rise+t_width' D;}}
            _po_ { LHX { '0ns' X; 't_meas' l/h/x; 't_rise' X;}}
        }
    }
}

```

This is how the timing example would be represented in the WGL output:

```
equationsheet STUCK_sheet
  exprset STUCK_set
    v_scale := 1.0;
    t_period := 100nS;
    t_force := 0nS;
    t_meas := (t_period * 0.1) * v_scale;
    t_rise := (t_period / 2) * v_scale;
    t_width := (t_period * 0.2) * v_scale;
    _tp1_fall_1 := t_rise + t_width;
  end
end

timeplate tp1 period t_period
  "input_a" := input [t_force:S];
  ...
  "output_z" := output [0nS:X, t_meas:Q, t_rise:X];
  ...
  "refclk" := input [0nS:D, t_rise:S, _tp1_fall_1:D];
end
```

Timeplate Definition

The timeplate definition describes a single tester cycle and specifies where in that cycle all event edges are placed.

You must define all timeplates before they are referenced. A procedure file must have at least one timeplate definition. All clocks must be defined in the timeplate definition. The timeplate definition has the following format:


```
timeplate timeplate_name =
  timeplate_statement
  [timeplate_statement ...]
  period time;
end;
```

The following list contains available timeplate_statement statements. The timeplate definition should contain at least the force_pi and measure_po statements. You are not required to include pulse statements for the clocks. But if you do not “pulse” any of the clocks, the tool uses two cycles to pulse a clock, resulting in larger patterns.

The tool uses the pulse_clock statement rather than individual pulse statements when generating default procedures.

```
timeplate_statement:  
  offstate pin_name off_state;  
  force_pi time;  
  bidi_force_pi time;  
  measure_po time;  
  bidi_measure_po time;  
  force pin_name time;  
  measure pin_name time;  
  pulse pin_name time width [, time width];  
  pulse_clock time width [, time width];
```

Note

 In “timeplate_statement” definitions, you can use timing variables instead of time values. For more information, refer to “[Timing Variables](#)” on page 545.

The following is a list of statements in the timeplate definition:

- ***timeplate_name***


A string that specifies the name of the timeplate.

- ***offstate pin_name off_state***

A literal and double string that specifies the inactive, off-state value (0 or 1) for a specific named primary input pin that is pulsed within this timeplate but is not defined as a clock pin by the add_clocks command. The complex timeplates are most useful in the shift procedure where a non-clock pin must be pulsed while still maintaining a single cycle in the shift procedure.

This statement must occur before all other timeplate_statement statements. This statement is required for any pin that is not defined as a clock pin by the add_clocks command but is pulsed within this timeplate.

Note

 An “offstate” statement does not automatically force *pin_name* to its off state at time 0. For that to occur, you must force or pulse *pin_name* appropriately in a procedure.

- ***force_pi time***

A literal and integer pair that specifies the force time for all primary inputs.

- ***bidi_force_pi time***

A literal and integer pair that specifies the force time for all bidirectional pins. This statement enables the bidirectional pins to be forced after applying the tri-state control signal, so the system avoids bus contention. This statement overrides “force_pi” and “measure_po”.

- ***measure_po time***

A literal and integer pair that specifies the time at which the tool measures (or strobes) the primary outputs.

- ***bidi_measure_po time***

A literal and integer pair that specifies the time at which the tool measures (or strobes) the bidirectional pins. This statement overrides “force_pi” and “measure_po”.

- ***force pin_name time***

A literal, string, and integer that specifies the force time for a specific named pin.

Note



This force time overrides the force time specified in force_pi for this specific pin.

- ***measure pin_name time***

A literal, string, and integer that specifies the measure time for a specific named pin. You can use a “measure” statement in timeplates only to specify a measure time for a pin.

Note



This measure time overrides the measure time specified in measure_po for this specific pin.

- ***pulse pin_name time width* [, *time width*]**...

A literal, string, and repeatable integer set that specifies the pulse timing for a specific named pin.

pin_name — String that refers to a pin in the design. Valid pins must meet one of the following conditions:

Defined as a clock pin using the add_clocks command.

Not defined as a clock pin, but has a pulse signal and an offstate specified by the “offstate” statement.


time — Integer that defines the offset from time 0 to the leading edge of the pulse.

width — Integer that defines the width of the pulse.

To define a multiple-pulse waveform, include multiple time and width pairs separated by a comma.

All pulses must occur within the tester cycle period. You define this period using the “period” keyword.

Note

 Multiple pulses are only supported for the following output formats: Verilog, WGL, STIL, STIL2005, CTL, FJTDL, MITDL, and TSTL2. Additionally, the TSTL2 output format does not support more than two pulses.

For MITDL format there is restriction that multiple pulse timing must be a cyclical repetition of the first pulse. Consequently, multi-pulse and double-pulse timing in the procedure file only works in the MITDL output without an error if the timing fits the restrictions of the MITDL syntax.

- **pulse_clock *time width* [, *time width*]...**

A literal and integer set that specifies the pulse timing for all signals defined as clocks, unless another statement such as a “force” or “pulse” exists for a particular clock signal. This is similar to the `force_pi` statement, which specifies the timing for ports that are not explicitly overridden by a force statement for those specific ports.

time — Integer that defines the offset from time 0 to the leading edge of the pulse for the first pulse. For subsequent edges in a multi pulse clock, time equals the time of the previous leading edge plus the period of the clock.

width — Integer that defines the width of the pulse.

You can use the `pulse_clock` statement to ensure that any added clocks (especially internal clocks) automatically have defined timing.

You may have multiple `pulse_clock` statements within a timeplate definition, as long as each statement has a different number of offset and width pairs. If two `pulse_clock` statements in the timeplate definition have the same number of offset and width pairs, the tool issues an error.

For more information on multiple-pulse clocks, see “[Multiple-Pulse Clocks.](#)”

All pulses must occur within the tester cycle period. You define this period using the “period” keyword.

For example (1x `pulse_clock`):

```
timing_variables =
  tester_period = 10;
  strobe_1 = (0.96 * tester_period);
  t_time = (0.25 * tester_period);
  t_width = (0.5 * tester_period);
end;

timeplate tesseract_ijtag =
  force_pi 0 ;
  measure_po strobe_1;
  force tck 0;
  pulse_clock t_time t_width;
  period tester_period;
end;
```

- **period *time***

A literal and integer pair that defines the period of a tester cycle. This statement ensures that the cycle contains sufficient time, after the last force event, for the circuit to stabilize. The time you specify should be greater than or equal to the final event time.

Example 1

```
timeplate tp1 =  
  force_pi 0;  
  pulse T 30 30;  
  pulse R 30 30;  
  measure_po 90;  
  period 100;  
end;
```

Example 2

The following example shows a shift procedure that pulses b_clk with an off-state value of 0. The timeplate tp_shift defines the off-state for pin b_clk. The b_clk pin is not declared as a clock in the ATPG tool.

```
timeplate tp_shift =  
  offstate b_clk 0;  
  force_pi 0;  
  measure_po 10;  
  pulse clk 50 30;  
  pulse b_clk 140 50;  
  period 200;  
end;  
  
procedure shift =  
  timeplate tp_shift;  
  cycle =  
    force_sci;  
    measure_sco;  
    pulse clk;  
    pulse b_clk;  
  end;  
end;
```

Example 3

In the following example, the pin b_clk is not declared as a clock in the ATPG tool. However, in the shift procedure, the user needs the pin to be pulsed twice with an offstate of 0.


```
timeplate tp_shift =  
  offstate b_clk 0;  
  force_pi 0;  
  measure_po 10;  
  pulse clk 50 30;  
  pulse b_clk 40 50, 140 50;  
  period 200;  
end;  
  
procedure shift =  
  timeplate tp_shift;  
  cycle =  
    force_sci;  
    measure_sco;  
    pulse clk;  
    pulse b_clk;  
  end;  
end;
```

Multiple-Pulse Clocks

You can use the `pulse_clock` statement to handle multiple-pulse clock timing and still use a single generic timeplate template definition in various flows. The `pulse_clock` statement handles multiple-pulse timing definitions in the same manner as the `pulse` statement.

The <code>pulse_clock</code> Statement	554
Inferred Timing	555
Differences Between Default <code>add_clock</code> and 1x Multiplier Clock.	556

The `pulse_clock` Statement

In a timeplate statement, each `pulse_clock` statement has a different number of integer pairs for the offset and width of the clocks. The different statements represent 1x, 2x, 3x, and so on, pulse timing for clocks. The frequency multiplier that you specify for the clocks identifies the appropriate pulse timing used for them.

When you specify a multiplier for a clock and no `pulse_clock` statement exists for that clock, the tool creates inferred timing for the clock using the timeplate period. The tool issues an error when a port-specific pulse or force statement exists for the clock and the timing in the statement does not specify the correct number of pulse statements to match the frequency multiplier of the clock.

The following example has multiple `pulse_clock` statements. It is a timeplate that includes both a port specific pulse statement for a clock and the generic `pulse_clock` statements for all other clocks. The first `pulse_clock` statement sets the default clock timing for this timeplate and contains a single pair of numbers (offset and width). Clocks other than “SlowClockA” that do not have frequency multipliers use this timing. Clocks with 2x multipliers use the second `pulse_clock` statement, and clocks with 4x multipliers use the third `pulse_clock` statement. The tool uses inferred timing for clocks specified with any other frequency multiplier and issues an error if "SlowClockA" has a frequency multiplier of 2x or more.

```
timeplate tp1 =  
    force_pi 0;  
    measure_po 95;  
    pulse SlowClockA 20 50;  
    pulse_clock 30 30;  
    pulse_clock 13 25, 63 25;  
    pulse_clock 6 12, 31 12, 56 12, 81,12;  
    period 100;  
end;
```

The following example shows a timeplate with one port-specific pulse and one `pulse_clock` statement for 4x multiplier timing. Clocks other than “ClockA” that do not have a frequency multiplier use `force_pi` timing. Inferred timing only occurs when clocks have frequency multipliers, which means other clocks still use NRZ timing when you use multiple cycles to create the clock pulse.

```

timeplate tp1 =
  force_pi 0;
  measure_po 95;
  pulse ClockA 20 50;
  pulse_clock 6 12, 31 12, 56 12, 81, 12;
  period 100;
end;

```

Inferred Timing

Based on the period of the timeplate, the tool creates inferred timing for frequency multiplied clocks when there are no `pulse_clock` statements in the timeplate with the correct number of clock edges for the clock.

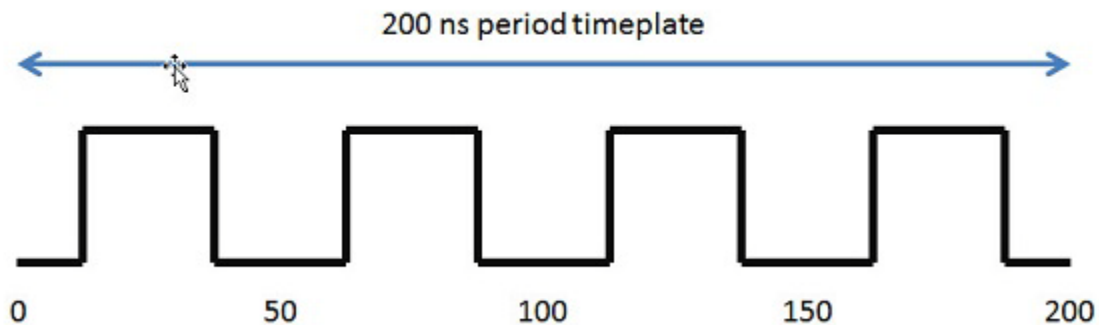
The total period of a clock pulse is the period of the timeplate divided by the frequency multiplier of the clock. The offset for the leading edge of the first clock pulse is one-fourth of the period of the clock. For example, the inferred timing for a 4x clock in a timeplate with a period of 200 is equivalent to the following `pulse_clock` definition:

```

pulse_clock 13 25, 63 25, 113 25, 163 25

```

Figure 10-1. 200ns Timing Waveform



The first edge is one-fourth the period of the clock rounded to the nearest integer, and the width of the pulse is one-half the period of the clock. Each subsequent edge is the previous leading edge plus the period of the clock.

The tool bases the inferred timing for a 1x frequency-multiplied clock on any other clock timing found for a single pulse clock; otherwise, it uses the inferred timing formula.

When you specify a timeplate period and a clock frequency multiplier that combine to create inferred timing that is too small to be integer timing, the tool automatically reduces the procedure file timescale and scales all timing numbers to larger values.

For example, suppose the timescale for the procedure file is 1ns, the period of the timeplate is 5, and the clock frequency multiplier is 10. The tool automatically adjusts the timescale to 100ps and the period of the timeplate becomes 50. It adjusts all of the other numbers accordingly. In this case, the tool multiplies them by 10.

Differences Between Default add_clock and 1x Multiplier Clock

Default add_clock clocks use the time specified by pulse_clock statements with one pulse or by pin-specific pulse statements. However, frequency-multiplied clocks use the timing specified for them only if they match the number of pulses. If this is not the case, the tool uses inferred timing and issues an error if there are mismatches.

Clocks specified with a frequency multiplier of one behave differently than default clocks. For this reason, you must specify the timing for a 1x frequency-multiplied clock with a single clock pulse. The tool does not use NRZ timing with multiple cycles or forced timing edges, and it issues an error if the timeplate specifies a double-pulse or multiple-pulse timing for the 1x clock.

Always Block

This optional block definition specifies events that happen in all cycles of all procedures. Because the always block specifies events for all cycles, it is used with all timeplates and does not require a timeplate to be referenced in the block. Also, any signal that is pulsed in the always block must have a pulse waveform in all timeplate definitions.

If you defined any pulse-always clocks using the add_clocks command, an always block is automatically created in the procedure file, if one does not already exist, and a pulse statement added for each clock. Similarly, if you pulse a clock signal in the always block, the signal is automatically defined as a pulse-always clock. For more information, refer to the [add_clocks](#) description in the *Tessent Shell Reference Manual*.

Note



Pulse-always clocks are not automatically pulsed in a named capture procedure. The clocks must be pulsed explicitly.

All events specified in the always block is subject to rules checks that apply to each procedure. In other words, the events in the always block is added to each cycle of each procedure, and all DRC rules still apply to these events.

When saving patterns that preserve the structure of the procedures as macros (such as the CTL pattern file, or structural STIL pattern file), the events in the always block is placed in the cycles of each procedure. The always block is not present in the structural pattern file as a macro or procedure.

Always Block Syntax

The always block has the following syntax.

```
always =  
  always_statement ;  
  [always_statement ; ... ]  
end ;
```

The `always_statement` is defined as one of the following.

```
pulse pin_name ;  
force pin_name value ;
```

Always Block Example

The following is a partial example of an always block.

```
set time scale 1,0 ns ;  
  
timeplate tp1 =  
  force_pi 0 ;  
  measure_po 10 ;  
  pulse ref_clk 20 20, 60 20 ;  
  pulse shift_clk 50 20 ;  
  period 100 ;  
end ;  
always =  
  pulse ref_clk ;  
end ;  
  
procedure shift =  
  timeplate tp1 ;  
  cycle =  
    force_sci ;  
    measure_sco ;  
    pulse shift_clk ;  
  end ;  
end ;
```

Procedure Definition

The procedure definition is the heart of the procedure file. The procedure defines precisely how the scan circuitry operates.

All procedure definitions contain one or more cycle definitions. Each cycle definition in the procedure specifies a vector; each statement in the cycle specifies which events occur in that vector. The timeplate being used specifies any timing associated with that vector. The following is a list of rules for writing procedure definitions:

- If more than one timeplate is defined, you can assign a specific timeplate for each procedure definition or for each cycle within the procedure definitions. You must assign a timeplate at some point within a procedure definition.
- You must group all procedure statements, except `scan_group`, `timeplate`, `apply`, and `loop`, into cycle statements.

- You cannot specify time values in cycle statements.
- You cannot specify loop statements within cycle statements.
- The order of events within a cycle definition does not matter. The assigned timeplate specifies the order.
- Within a procedure definition, you can specify a scan group.
- Each scan group needs a unique test procedure file. You associate the test procedure file with the scan group when you specify the [add_scan_groups](#) command.
- Text following “//” is a comment and is ignored.
- You can include blank lines.
- You define a procedure type for a particular scan group (with the exception of the seq_transparent and clock procedures) only once in a test procedure file.
- You can only have a single test_setup procedure, even if you define multiple scan groups for your design.

The procedure definition has the following general format, but certain statements are restricted to certain procedures.

```

procedure procedure_type [proc_name] =
  [scan_group scan_group_name;]
  proc_statement [proc_statement ...]
end;

proc_statement:
  [timeplate timeplate_name;]
  cycle =
    cycle_statement [cycle_statement ...]
  end;
  annotate "quoted string";
  apply proc_name #times;
  loop loop_count =
    cycle =
      cycle_statement [ cycle_statement ...]
    end;
  end;

  cycle_statement:
    force_pi;
    bidi_force_pi;
    force_sci;
    force_sci_equiv;
    measure_po;
    bidi_measure_po;
    measure_sco;
    restore_pi;
    restore_bidi;
    bidi_force_off;
    pulse_capture_clock;
    force_capture_clock_on ;
    force_capture_clock_off ;
    pulse_read_clock;
    pulse_write_clock;
    force pin_name value;
    expect pin_name value;
    condition cell_name value;
    measure pin_name;
    initialize instance_name [value];
    pulse pin_name;
    timeplate timeplate_name;
    annotate "quoted string";

```

- ***procedure_type***

A string that specifies the type of procedure that follows. The following list contains valid procedures types:

- test_setup
- shift
- load_unload
- shadow_control
- master_observe
- capture
- clock_po
- ram_sequential
- ram_passthru
- clock_sequential
- seq_transparent
- test_end
- clock
- sequential
- skew_load

- shadow_observe
- init_force
- sub_procedure

For more information, refer to “[The Procedures](#)” on page 576.

- *proc_name*

An optional string that specifies the user-defined name of the procedure. Because you can specify multiple seq_transparent and clock procedures in a test procedure file, these procedure types require explicit procedure names, *proc_name*, for each procedure that you define.

- **scan_group** *scan_group_name*

A literal and string pair that specifies a scan group within a scan procedure. Because some of the scan procedures are scan group specific, you can specify scan groups within scan procedures. This makes it possible to define the scan procedures (shift, load_unload) for multiple scan groups within the same procedure file. You can then specify this file on the [add_scan_groups](#) command for each scan group in this file. If you use the [read_procfile](#) command to read a procedure file, you must include this statement. However, if you use the [add_scan_groups](#) command, this statement is optional because the group is specified on the command line. When the tool writes out a procedure file, it produces the scan_group statement.

Note



The scan_group_name argument is case-sensitive if the netlist used is case-sensitive.

- **timeplate** *timeplate_name*

A literal and string pair that specifies the name of the timeplate the procedure uses.

A timeplate statement at the beginning of the procedure, outside of the cycle definitions, is the timeplate used by the entire procedure, if no other timeplates are referenced.

A timeplate statement within a cycle is the timeplate used for that cycle and all other subsequent cycles until another timeplate statement is encountered. For more information about timeplates, refer to “[Timeplate Definition](#)” on page 548.

- **annotate** “*quoted string*”;

A literal and string pair that reports the Verilog test bench annotations during simulation.

The annotate statement is optional and must always include a quoted string. All procedures can be annotated, including sub-procedures. The annotate statement can occur inside or outside of cycle blocks, including before the first cycle or after the last cycle.

This example shows the annotate statement in the beginning of a cycle along with the cycle timeplate statement, before any event statements:

```
CYCLE =
  [ TIMEPLATE tp_name ; ]
  [ ANNOTATE "quoted string" ; ]
  event_statement;
...
END ;
```

The following is an example of using the annotate statement outside of a cycle block:

```
procedure test_setup =
  timeplate tpl ;
  annotate "Before first cycle" ;
  cycle =
    ...
  end;
  annotate "start sub procedure" ;
  apply mySub 1 ;
  ...
end;
```

The following is an example of an annotate statement used in a test_setup procedure, and how this appear sin a STIL pattern file.

```
Procedure test_setup =
  timeplate tpl;
  cycle =
    annotate "first cycle in test_setup" ;
    force reset 1;
    force clock 0;
  end;
  cycle =
    annotate "next annotation" ;
    force reset 0;
  end;
...

```

This is a segment of the resulting STIL pattern file:

```
W tset_tpl;
V { _pi_ = 0X11XXX;
  _po_ = XXXX;
}
Ann {* Begin chain test *}
Ann {* first cycle in test_setup *}
V { ...
}
Ann {* next annotation *}
V { ...
}
```

- label “*quoted_string*”;

A literal and string pair for including pattern labels in saved patterns. As with the annotation statement, you can have one label statement per cycle in a procedure definition. The `quoted_string` becomes a pattern label for the vector that corresponds to that procedure cycle.

Only the STIL and WGL pattern formats support a pattern label statement. For pattern file formats that don't support a pattern label, the label is present as an annotation statement that has the string "label:" added at the beginning of the label string.

For the simulation test bench, the label is also present as an annotation that has the string "label:" added at the beginning, and the annotation is echoed when the patterns are simulated. You can use the existing parameter file keyword `SIM_ANNOTATE_QUIET` to turn off echoing the annotations and labels while simulating.

Each pattern label is a unique identifier, with its vector count appended to the end of the label string.

This statement can be used at the start of any cycle, just like an annotation statement. A cycle cannot contain both a label and an annotation statement.

The following example shows how to use the label statement within the `test_setup` procedure:

```
procedure test_setup =
  timeplate my_tp;
  cycle =
    force my_sig 0;
  end;
  cycle =
    label "end of test_setup" ;
    force my_sig 1;
  end;
end;
The previous example produces the following STIL vectors:
V { _pi_ = ...;
}
"end of test_setup_1": V { _pi_ = ...;
}
```

- ***apply proc_name #times***

A literal and double-argument string that tells the tool to apply the specified procedure the specified number of times. You must use the apply shift statement at least once in the `load_unload` procedure. For the apply shift statement, you must enter a proper *#times* parameter.

If required, you must enter the `apply shadow_control` statement immediately after the apply shift procedure statement, and you must set the *#times* argument to 1. The apply statement is only valid outside of the cycle blocks because it specifies another group of cycles within another procedure to be added at that point.

- ***loop loop_count***

A literal and integer pair that specifies the loop count and is followed by a block of statements. The loop procedure statement takes the loop count and causes all cycles within the loop block to be repeated by the number of times specified by the count. For example, the following test procedure file excerpt specifies 3 cycles within the loop that are each repeated 20 times:

```
procedure test_setup =
  timeplate tp1 ;
  cycle =
    force_pi;
    measure_po;
  end;
  loop 20 =
    cycle =
    end;
    cycle =
      pulse tck;
    end;
    cycle =
    end;
  end; // end loop
end;
```

Nesting loops within other loops is permitted. For example, the following test procedure file excerpt causes tck to be pulsed 20 times and clk_a to be pulsed 100 times:

```
procedure test_setup =
  timeplate tp1 ;
  cycle =
    force_pi;
    measure_po;
  end;
  loop 20 =
    cycle =
    end;
    cycle =
      pulse tck;
    end;
    loop 5 =
      cycle =
        pulse clk_a;
      end;
    end; // end inside loop
    cycle =
    end;
  end; // end outside loop
end;
```

This statement can be used in procedures but must be specified outside of the cycle statement.

The loop statement is preserved in the flat model when the tool writes the model and is also present in the TCD files.

When writing out the patterns in tester pattern formats, the loops are preserved where possible, and unrolled if the syntax of the pattern file does not support loops. Specifying the [ALL_NO_LOOP](#) parameter keyword unrolls loops in the pattern files in similar fashion to sub procedures that are applied more than once. Using the loop statement to repeat a certain number of cycles N times is exactly equivalent to putting those cycles within a sub procedure, then applying that procedure N times.

- **cycle_statement**

The following list describes valid cycle_statement keywords. Cycle_statements cannot contain time values.

- force_pi

A literal that specifies for the tool to force all primary inputs.

- bidi_force_pi

A literal that specifies for the tool to force all bidirectional pins.

- force_sci

A literal that specifies for the tool, in the shift procedure, to place values on the scan chain inputs, thus implementing scan cell controllability.

- force_sci_equiv

A literal that acts the same as the force_sci statement, except that it also forces all pins equivalent to the scan input pins. Using this statement places the complement value on the associated differential pin of a scan input during scan loading. This statement is necessary because the test procedures do not consider pin equivalence relationships (those specified with add_input_constraints -equivalent).

- measure_po

A literal that specifies for the tool to measure or strobe the primary outputs.

- bidi_measure_po

A literal that specifies for the tool to measure or strobe the bidirectional pins.

- measure_sco

A literal that specifies for the tool, in the shift procedure, to measure scan output values, thus implementing scan cell observability. In End Measure Mode (refer to “[Creating Test Procedure Files for End Measure Mode](#)” on page 605), measure_sco is also used in the load_unload procedure.

- restore_pi

A literal that returns primary inputs to their original states (prior to this procedure’s execution). You use the restore_pi statement at the end of a seq_transparent procedure.

- restore_bidi

A literal that returns bidirectional pins to their original states (prior to this procedure’s execution). You use the restore_bidi statement at the end of a “clock” procedure.
- bidi_force_off

A literal that specifies for the tool to force all unconstrained bidirectional pins off.
- pulse_capture_clock

A literal that specifies for the tool to pulse the capture clock.
- force_capture_clock_on

A literal that specifies the cycle when the capture clock goes active. This statement and the force_capture_clock_off statement can be used in place of the pulse_capture_clock statement.

The “_on” refers to the active state of the clock, which is not necessarily the high binary value. This statement is used only with the non-scan procedures and cannot be mixed with the following statements in the same procedure:

 - pulse_capture_clock
 - pulse_write_clock
 - pulse_read_clock
- force_capture_clock_off

A literal that specifies the cycle when the capture clock goes inactive. This statement and the force_capture_clock_on statement can be used in place of the pulse_capture_clock statement.

The “_off” refers to the inactive state of the clock, which is not necessarily the low binary value. This statement is used only with the non-scan procedures and cannot be mixed with the following statements in the same procedure:

 - pulse_capture_clock
 - pulse_write_clock
 - pulse_read_clock
- pulse_read_clock

A literal that specifies for the tool to pulse the RAM read clock.
- pulse_write_clock

A literal that specifies for the tool to pulse the RAM write clock.
- force pin_name value

A literal and double string that forces the specified value of 0, 1, X, or Z on the specified pin. The pin names you specify must be valid pin pathnames for primary inputs.

- expect name value

A literal and double string that causes the tool to expect the specified value of 0, 1, X, or Z on the specified internal pin or port. The default value is X. You can use “expect” statements only in the test_setup and test_end procedure. Internal pins are checked by DRC and are compared in the test bench. For ports, the tool validates the values in the test bench, the DRCs, and in the tester pattern formats.

- condition cell_name value

A literal and double string that you use at the beginning of a seq_transparent procedure to identify the necessary scan cell states (conditions) to establish transparency in non-scan cells. You identify the scan cell by the pin pathname associated with the output of its state element. The path from the defined pin to the scan cell must only contain buffers and inverters. The value argument sets the value at the specified pin pathname, which may be inverted relative to the associated scan cell value.

- measure pin_name

A literal and string pair that specifies for the tool to measure the value of the named pin. You can use a “measure” statement in the capture procedure only to specify a measure on a pin in a different cycle than the measure_po event.

- initialize instance_name value

A literal and string pair that initializes the named memory element to the value given. This statement is particularly useful for initializing the finite state machine in the TAP controller of boundary scan circuitry, when the TAP does not contain the TRST signal. Once set to a binary state, the TCK and TMS pins can place the finite state machine in a known state. If not set, these pins remain at X.

If you do not specify a value, the tool chooses a random value to assign to all latches and flip-flops with the specified instance name.

- pulse pin_name

A literal and string pair that specifies for the tool to pulse the named clock pin.

- observe_method value

A literal and string pair set to a value of master, slave, or shadow, to specify for a specific observe method to be defined for each named capture procedure.

The following example shows how to use a `cycle_statement` to force scan inputs and measure scan outputs:

```
procedure shift =
  scan_group grp1;
  timeplate tp1;
  cycle =
    force_sci;
    measure_sco;
    pulse T;
  end;
end;
```

Clock Control Definition

You can manually create clock control definitions in the test procedure file.

For complete information about when you use this definition, refer to “[Support for Internal Clock Control](#)” in the *Tessent Scan and ATPG User’s Manual*.

ATPG Restrictions

The following restrictions apply to ATPG when clock control definitions are enabled:

- Clock_PO patterns are disabled.
- In undefined cycles, the internal clock is assumed to be off, even if the source clock pulses.
- Source clocks are pulsed regardless of clock restrictions. Any false paths should be explicitly defined with DC or the [add_false_paths](#) command.
- External clocks without clock control definitions are controlled through top-level pins.
- Clock control definitions applied to a clock defined as equivalent also applies to all associated equivalent clocks.
- Timeplate definitions apply only to external clocks.
- If you use the [set_clock_restriction](#) -same_clocks_between_loads command, you must use one of the following definitions to pulse the controlled clock:
 - {ATPG_SEQUENCE, END}
 - {ATPG_SEQUENCE <N> <M>, END} with N starting from 0. The generated test pattern includes M+1 capture cycles between the scan loading operation and the scan unloading operation.
 - <ATPG_CYCLE <i>, END} with i=0 defined

If none of these statements are present a warning displays during ATPG about clock controls that cannot be applied because of clock restrictions.

Rules for Clock Control Definitions

The following rules apply to the clock control definitions in the test procedure file:

- Global control conditions and source clocks defined for equivalent clocks must be the same.
- When a clock is forced off, it cannot be used as a source in the same definition.
- A FORCE statement cannot force a clock pin to an on state.
- Clock control cannot be applied to an asynchronous free-running clock.
- Condition statements cannot be applied to non-scan state elements.
- You must specify a condition to turn on the internal clock; otherwise, it is assumed to be off.
- When multiple sequence clock control definitions are defined for the same clock, they must use mutually exclusive pulse conditions as follows:
 - The clock control condition to pulse a clock in sequence mode must be mutually exclusive with the clock control condition for the same clock in per-cycle mode.
 - The condition to pulse a clock in sequence mode must be mutually exclusive with the condition to pulse the same clock by using another sequence mode.

Keywords

The following is a list of keywords used in clock control statement:

- **ATPG_CYCLE** *cycle_number*

A literal and integer that specifies a test pattern capture cycle to map the clock control to. The specified capture cycle values start from 0, which corresponds to the first capture cycle after scan loading.

Multiple ATPG_CYCLE definitions can be declared to pulse the internal clock at the same capture cycle with different sets of conditions.

Use a FORCE statement to turn the clock off, or the clock continues to pulse when the conditions are satisfied.

The specified and actual capture cycles may differ—see “[Capture Cycle Determination](#)” in the *Tessent Scan and ATPG User’s Manual*.

- **ATPG_SEQUENCE** *N M*

A literal and an integer pair that specify a range of capture cycles for clock pulsing from N to M consecutively.

Define the condition to pulse the clock continuously from capture cycle N ($N \geq 0$) to capture cycle M ($M \geq N$) right after scan loading. If N is greater than 0, the clock is automatically set to off state from the first capture cycle right after scan loading to the capture cycle N - 1. When the generated test pattern includes more than M capture cycles after scan loading, the clock is set to off state from the M + 1 capture cycle to the last capture cycle.

Multiple ATPG_SEQUENCE definitions can be declared as necessary.

Use a FORCE statement to turn the clock off, or the clock continues to pulse when the conditions are satisfied.

The specified and actual capture cycles may differ—see “[Capture Cycle Determination](#)” in the *Tessent Scan and ATPG User’s Manual*.

- **ATPG_SEQUENCE**

A literal that specifies clock pulsing. When a condition list is provided, the controlled clock pulses in all capture cycles in the pattern when the conditions are met. When checking the off conditions for cycles outside the capture window, the conditions listed in this special atpg_sequence are ignored.

When there is no condition list, the controlled clock pulses unconditionally in every capture cycle between scan loading.

Multiple ATPG_SEQUENCE definitions can be declared as necessary.

Use a FORCE statement to turn the clock off, or the clock continues to pulse when the conditions are satisfied.

The specified and actual capture cycles may differ—see “[Capture Cycle Determination](#)” in the *Tessent Scan and ATPG User’s Manual*.

- **CLOCK_CONTROL *pin_pathname***

A literal and string value that specifies the pin pathname of the PI for the internal clock. The specified pin must be an existing clock. Internal clocks must also be defined with the [add_clocks](#) command.

- **CONDITION *cell_name value***

An optional literal and double string that specifies necessary scan cell states (conditions). The scan cell is specified by the pin pathname associated with the output of its state element. The value argument specifies the value loaded into the scan cell at the end of shift.

The specified and actual capture cycles may differ—see “[Capture Cycle Determination](#)” in the *Tessent Scan and ATPG User’s Manual*.

- **FORCE** *pin_pathname value*

A literal and double string that forces a value of 0, 1, or Z on a specified pin. The specified pin names must be valid pin pathnames for primary inputs. This keyword is used to force necessary pins off during capture cycles when the controlled clock is pulsed.

- **SOURCE_CLOCK** *pin_pathname...*

A literal and repeatable string that specifies one or more source clocks to drive the internal clock logic to pulse in the specified capture cycle(s). If no source clock is specified, the source clock is assumed to be an always-capture clock that pulses in every capture cycle.

- **END**

Required literal the specifies the end of an ATPG_CYCLE or ATPG_SEQUENCE block, or at the end of the clock control definition.

Global Condition Statements

Global conditions are **CONDITION** statements that are accessible in every scope of a clock control definition. You can use global conditions to define default conditions within a clock control definition.

For example, you can specify a **CONDITION** statement for all ATPG_CYCLE/ATPG_SEQUENCE blocks within a definition. Then you can define a **CONDITION** statement within individual ATPG_CYCLE/ATPG_SEQUENCE blocks to override the global **CONDITION** variable when necessary.

The following example uses local conditions (*italicized*) to define some of the control bits necessary for each scan cell to pulse the clock. The global conditions define other conditions that must be satisfied for all clock cycles.

```
CLOCK_CONTROL /clk_ctrl/int_clk1 =
SOURCE_CLOCK ref_clk;
CONDITION /clk_ctrl/enable_1/q 0;
CONDITION /clk_ctrl/enable_2/q 0;
ATPG_CYCLE 0 =
    CONDITION /clk_ctrl/F0/q 1;
END;
ATPG_CYCLE 1 =
    CONDITION /clk_ctrl/F1/q 1;
END;
ATPG_CYCLE 2 =
    CONDITION /clk_ctrl/F2/q 1;
END;
ATPG_CYCLE 3 =
    CONDITION /clk_ctrl/F3/q 1;
END
END;
```

The previous example is equivalent to the following:

```
CLOCK_CONTROL /clk_ctrl/int_clk1 =
SOURCE_CLOCK ref_clk;
ATPG_CYCLE 0 =
    CONDITION /clk_ctrl/F0/q 1;
    CONDITION /clk_ctrl/enable_1/q 0;
    CONDITION /clk_ctrl/enable_2/q 0;
END;
ATPG_CYCLE 1 =
    CONDITION /clk_ctrl/F1/q 1;
    CONDITION /clk_ctrl/enable_1/q 0;
    CONDITION /clk_ctrl/enable_2/q 0;
END;
ATPG_CYCLE 2 =
    CONDITION /clk_ctrl/F2/q 1;
    CONDITION /clk_ctrl/enable_1/q 0;
    CONDITION /clk_ctrl/enable_2/q 0;
END;
ATPG_CYCLE 3 =
    CONDITION /clk_ctrl/F3/q 1;
    CONDITION /clk_ctrl/enable_1/q 0;
    CONDITION /clk_ctrl/enable_2/q 0;
END
END;
```

The previous example demonstrates the importance of ensuring that global conditions do not conflict with local conditions. To further illustrate this point, consider the following incorrect definition of global conditions:

```
// Example of incorrect definition of global conditions
CLOCK_CONTROL /clk_ctrl/int_clk1 =
SOURCE_CLOCK ref_clk;
    CONDITION /clk_ctrl/F0/q 0;
ATPG_CYCLE 0 =
    CONDITION /clk_ctrl/F0/q 1;
END;
ATPG_CYCLE 1 =
    CONDITION /clk_ctrl/F1/q 1;
END;
ATPG_CYCLE 2 =
    CONDITION /clk_ctrl/F2/q 1;
END;
ATPG_CYCLE 3 =
    CONDITION /clk_ctrl/F3/q 1;
END
END;
```

On the surface, it may appear correct that the condition in ATPG_CYCLE 0 overrides the global condition while the other cycles can still be satisfied. However, after the global condition is expanded to all cycles, the clock control definition looks like this:

```
// Example of incorrect definition of global conditions
CLOCK_CONTROL /clk_ctrl/int_clk1 =
SOURCE_CLOCK ref_clk;
  ATPG_CYCLE 0 =
    CONDITION /clk_ctrl/F0/q 1;
  END;
  ATPG_CYCLE 1 =
    CONDITION /clk_ctrl/F1/q 1;
    CONDITION /clk_ctrl/F0/q 0;
  END;
  ATPG_CYCLE 2 =
    CONDITION /clk_ctrl/F2/q 1;
    CONDITION /clk_ctrl/F0/q 0;
  END;
  ATPG_CYCLE 3 =
    CONDITION /clk_ctrl/F3/q 1;
    CONDITION /clk_ctrl/F0/q 0;
  END
END;
```

You can now see that it would not be possible to pulse the clock in ATPG_CYCLE 0 while also pulsing it in any other cycle. The tool can load only one value into /clk_ctrl/F0, so it can either pulse the clock in cycle 0 by loading a 1 or pulse it in another cycle by loading a 0.

Per-Cycle Clock Control Definition Example

The following example defines per-cycle clock control for two internal clocks (/top/core1/clk1 and /top/core1/clk2):

```
CLOCK_CONTROL /top/core1/clk1 =
  ATPG_CYCLE 0 =
    CONDITION /pll_ctl/cell_0/Q 1;
  END;
  ATPG_CYCLE 1 =
    CONDITION /pll_ctl/cell_1/Q 1;
    CONDITION /pll_ctl/cell_4/Q 0;
  //both conditions must be satisfied for clock to pulse in
  //capture cycle 1
  END;
END;
CLOCK_CONTROL /top/core1/clk2 =
  ATPG_CYCLE 0 =
    CONDITION /pll_ctl/cell_2/Q 1;
  END;
  ATPG_CYCLE 1 =
    CONDITION /pll_ctl/cell_3/Q 1;
    CONDITION /pll_ctl/cell_5/Q 1;
  END;
END;
```

Sequence Clock Control Definition Example

The following example defines sequence clock control for two internal clocks (/top/core1/clk1 and /top/core1/clk2) derived from the source clock clk_src:

```
CLOCK_CONTROL /top/core1/clk1 =
  SOURCE_CLOCK clk_src;
  ATPG_SEQUENCE 0 1 =
  // Pulses 2 consecutive cycles if the scan cell
  // is loaded with 1, and the source clock is pulsed.
  CONDITION /pll_ctl/cell_1/Q 1;
  END;
END;
CLOCK_CONTROL /top/core1/clk2 =
  SOURCE_CLOCK clk_src;
  ATPG_SEQUENCE 0 1=
  CONDITION /pll_ctl/cell_2/Q 1;
  END;
END;
```

The following example defines sequence clock control for two internal clocks (/top/core1/clk1 and /top/core1/clk2) derived from the source clock clk_src. The clock pulses in all capture cycles when the conditions are met.

```
CLOCK_CONTROL /top/core1/clk1 =
  SOURCE_CLOCK clk_src;
  ATPG_SEQUENCE =
  // Pulse clock in all capture cycles if the scan cell
  // is loaded with 1, and the source clock is pulsed.
  CONDITION /pll_ctl/cell_1/Q 1;
  END;
END;
CLOCK_CONTROL /top/core1/clk2 =
  SOURCE_CLOCK clk_src;
  ATPG_SEQUENCE =
  CONDITION /pll_ctl/cell_2/Q 1;
  END;
END;
```

The following example pulses clock /top/core1/clk1 unconditionally in every capture cycle between scan loading:

```
CLOCK_CONTROL /top/core1/clk1 =
  ATPG_SEQUENCE =
  // empty body
  END;
END;
```

The following example defines a multi-sequence clock control definition:

```
CLOCK_CONTROL /top/core/clk1_int =
SOURCE_CLOCK /clk1;
ATPG_SEQUENCE 0 2 =
    CONDITION /pll/ctl_1/Q 1;
    FORCE ENABLE_1 1;
END;
ATPG_SEQUENCE 3 4 =
    CONDITION /pll/ctl_1/Q 0;
    FORCE ENABLE_1 1;
END;
END;
```

Exclusive conditions ensure that only one sequence block is applied per capture cycle (otherwise, no sequence is applied). If no cycle numbers are specified for sequence clock control, the clock pulses in every capture cycle when conditions are loaded.

Multiple Sets of Conditions for the Same Cycle Example

The following example shows that if a clock can be pulsed in a particular cycle or sequence of cycles when there are multiple sets of conditions where any one set can activate the clock for that cycle, the same cycle can be defined multiple times:

```
CLOCK_CONTROL /top/core1/clk1 =
ATPG_CYCLE 0 =
    CONDITION /pll_ctl/cell_1/Q 1;
END;
ATPG_CYCLE 0 =
    CONDITION /pll_ctl/cell_2/Q 1;
END;
END;
```

The previous example shows that /top/core1/clk1 can be pulsed in ATPG_CYCLE 0 when any set of the specified conditions are met. This demonstrates the case where loading a '1' into either /pll_ctl/cell_1 or /pll_ctl/cell_2 pulses the clock in cycle 0.

Similarly, the following example defines multiple sets of conditions for the same sequence of cycles, which can overlap. The sequence of cycles must have mutually exclusive conditions to ensure conditions for each ATPG_SEQUENCE can be satisfied without conflicting with other sequences.

```
CLOCK_CONTROL /top/core1/clk1 =
  ATPG_SEQUENCE 0 2 =
    CONDITION /pll_ctl/cell_1/Q 1;
    CONDITION /pll_ctl/cell_2/Q 0;
    CONDITION /pll_ctl/cell_3/Q 0;
  END;
  ATPG_SEQUENCE 0 3 =
    CONDITION /pll_ctl/cell_1/Q 0;
    CONDITION /pll_ctl/cell_2/Q 1;
    CONDITION /pll_ctl/cell_3/Q 0;
  END;
  ATPG_SEQUENCE 1 4 =
    CONDITION /pll_ctl/cell_1/Q 0;
    CONDITION /pll_ctl/cell_2/Q 0;
    CONDITION /pll_ctl/cell_3/Q 1;
  END;
END;
```

Source Clocks with Different Frequencies Example

The following example defines source clocks that have different frequencies when using clock control definitions:

```
timeplate _default_WFT_ =
  force_pi 0 ;
  measure_po 40 ;
  pulse clk1 45 10;
  pulse ref_clock 15 5, 40 5, 65 5, 90 5;
  pulse clocks_02/my_controller/U2/Z 45 10;
  pulse clocks_03/my_controller/U2/Z 45 10;
  pulse clocks_04/my_controller/U2/Z 45 10;
  period 100 ;
end;

procedure capture =
  timeplate _default_WFT_;
  cycle =
    force_pi ;
    measure_po ;
    pulse_capture_clock ;
  end;
end;
```

In this example, for one pulse of clk1, there are 4 pulses of ref_clock, specifically the ref_clock frequency is 4 times the frequency of clk1.

The Procedures

The test procedure file contains scan and clock procedures, and non-scan procedures. The scan and clock-related procedures inform the tool how to operate the scan chain and pulse clocks. The non-scan procedures can represent any type of pattern that the tool produces.

You can use the non-scan procedures to specify in which cycles of the procedure “potential events” happen. A potential event is an event that the ATPG engine may or may not have created to cover a certain fault.

To avoid DRC violations, each non-scan procedure must contain the proper statements in the correct order with the timing from the timeplate. The statements in a non-scan procedure can be spread over any number of cycles using a different timeplate for each cycle if needed.

A basic pattern consists of loading the scan chains, a default capture procedure, followed by unloading the scan chains; however, you do not specify the loading and unloading of scan chains in non-scan procedures. The following shows the basic pattern for non-scan procedures.

Basic Pattern

Force primary inputs
Measure primary outputs
Pulse capture clock

All example procedures shown in this section use one of the following two timeplates, unless otherwise stated:

```
timeplate tp1 =
  force_pi 0;
  measure_po 10;
  pulse scan_clk 30 10;
  pulse sys_clk 30 10;
  period 50;
end;

timeplate tp2 =
  force_pi 0;
  measure_po 10;
  pulse scan_mclk 15 10;
  pulse scan_sclk 30 10;
  period 50;
end;
```

Test_Setup (Optional)	577
Shift (Required)	580
Alternate Shift Procedure (Optional)	582
Load_Unload (Required)	583
Shadow_Control (Optional)	586

Master_Observe (Sometimes Required)	588
Shadow_Observe (Optional)	588
Skew_Load (Optional)	589
Clock_run (Optional)	591
Capture Procedures (Optional)	593
Clock_po (Optional)	598
Clock_sequential (Optional)	599
Init_force (Optional)	599
Test_end (Optional, all ATPG tools)	600
Sub_procedure	602

Test_Setup (Optional)

This optional procedure, which can only contain force, pulse, init, and expect event statements, sets non-scan elements to the desired states for the load_unload procedure. You may use this procedure only once for all scan groups, and it appears only once at the beginning of the test pattern set.

This procedure is particularly useful for initializing boundary scan circuitry. For an example using this procedure to set up boundary scan circuitry, refer to “[Pattern Generation for a Boundary Scan Circuit](#)” in the *Tessent Scan and ATPG User’s Manual*.

IJTAG Embedded Instruments

In the ATPG context patterns -scan, IJTAG is valid in the test_setup and test_end procedures only. It is invalid in any other procedure, as well as in ATPG’s analysis mode. Also, only **iReset** and **iCall** commands are valid in the test procedures.

For detailed information, see “[How to Set Up Embedded Instruments Through Test Procedures](#)” in the *Tessent IJTAG User’s Manual*.

Bidirectional Scan Out Pins

The value of all bidirectional scan out pins must be forced to the Z state (indicating it is operating in “output” mode) to properly sensitize the scan chain. When reading in a test procedure file, the tool automatically adds force events to the beginning of the load_unload procedure to force all bidi pins to Z.

Bidi pins that are clocks or constrained pins are not forced to a Z, as they were already forced to the off-state or the constrained values. Bidi scan-in pins are also not forced to a Z. Any bidi pin already forced later in the load_unload procedure is not be forced to a Z. Any bidi forced to a specific value in the test_setup procedure is instead be forced to this value instead of a Z.

Like previous automatic force values, these can be disabled by putting the “set autoforce off;” statement at the beginning of the procedure file.

Pin Constraints

If you use the `add_input_constraints` command to set pin constraints, be aware this command only forces pins during capture. To constrain these pins during `test_setup`, you should include the same pin constraints in the `test_setup` procedure. This ensures the pins are in the same state for loading the first pattern as for loading all subsequent patterns.

If you do not properly constrain the pins prior to the end of the `test_setup` procedure, the tool automatically constrains them by inserting a cycle statement in the `test_setup` procedure. However, this automatic handling may not insert the events with the timing you want. Also, the automatic handling is not included in DRC.

If you have defined input constraints but have not provided a `test_setup` procedure, the tool automatically generates a `test_setup` procedure to force those pins to their constrained values.

You can use both the [write_procfile](#) and the [report_procedures](#) commands to see the contents of the `test_setup` procedure the tool has generated. The `write_procfile` command writes existing procedure and timing data to a specified file. The `report_procedures` command writes the information to the screen.

Example 1

The following is an example using a `sub_procedure`. In this example, the signal named C retains its value of 1 during the test unless it is forced to a different value in a later cycle, by another procedure, or it is overwritten by WGL patterns.

```
procedure sub_procedure initialize =
  template soc_timeplate ;
  cycle =
    force C 1;
  end;
end;
```

The following example shows how to apply the previous `sub_procedure`. For more information, see “[Sub_procedure](#)” on page 602.

```
procedure test_setup =
  timeplate soc_timeplate;
  cycle =
    force test_en 1; // force test_en 1
    force chip_en 0; // force chip_en to 0
  end;
  apply initialize 10 ; // force C to 1 for 10 cycles
end;
```

Example 2

The following example shows a way to apply initialization cycles to a memory. The RST signal is active for the first 128 cycles, then it is deactivated in the next cycle (cycle 129).

```

procedure sub_procedure reset_mem =
    timeplate soc_timeplate ;
    cycle =
        force RST 1;
    end;
end;
procedure test_setup =
    timeplate soc_timeplate;
    apply reset_mem 128;
    cycle =
        force RST 0; // deactivate RST
    end;
end;

```

Example 3

The following example shows a way to use an expect statement in a test_setup procedure. The output signal (DFT) is expected to 1 in the first cycle and X in the remaining cycle. An “expect” statement does not work the same as a force or pulse statement. When none is present, it is assumed to mean do not measure.

```

procedure test_setup =
    timeplate soc_timeplate;
    cycle =
        expect DFT 1 ;
    end;
end;

```

Example 4

This example shows a way to start pulsing a clock in a test_setup procedure. The SYSCLK starts pulsing at cycle number 2 until the end of test.

```

timeplate soc_timeplate =
    force pi;
    measure_po 90;
    pulse SYSCLK 50 50;
    period 100;
end;

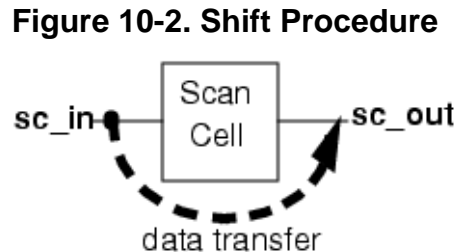
procedure test_setup =
    timeplate soc_timeplate;
    cycle =
        force RST_L 0;
    end;
    cycle =
        pulse SYSCLK;
    end;
end;

```

Shift (Required)

This required procedure describes how to shift data one position down the scan chain by forcing the scan input, toggling the clock(s), and strobing the scan output.

Figure 10-2 shows the data flow process for the shift procedure.



Within this procedure, you must use the `force_sci`, or `force_sci_equiv`, and the `measure_sco` event statements. You can also use the `force` and `pulse` event statements. A shift procedure can contain more than one cycle, although not all pattern formats can support multiple cycles and parallel load. Pattern formats that do not support multiple cycles are any parallel format other than STIL and Verilog. If you use `write_patterns` to write out one of these other parallel formats with a multicycle shift procedure, the command generates an AG11 error.

The times at which the timeplate used by the shift procedure applies the `force_sci` and `measure_sco` commands must be consistent with proper operation of the `load_unload` procedure. The `measure_sco` occurs at the `measure_po` time specified in the timeplate. The `force_sci` occurs at the `force_pi` time specified in the timeplate.

The following are examples of the shift procedure for both mux-DFF and LSSD architectures.

Mux-DFF Example

```
procedure shift =  
    timeplate tpl;  
    cycle =  
        // force scan chain input  
        force_sci;  
        // measure scan chain output  
        measure_sco;  
        // pulse the scan clock  
        pulse scan_clk;  
    end;  
end;
```

LSSD Example

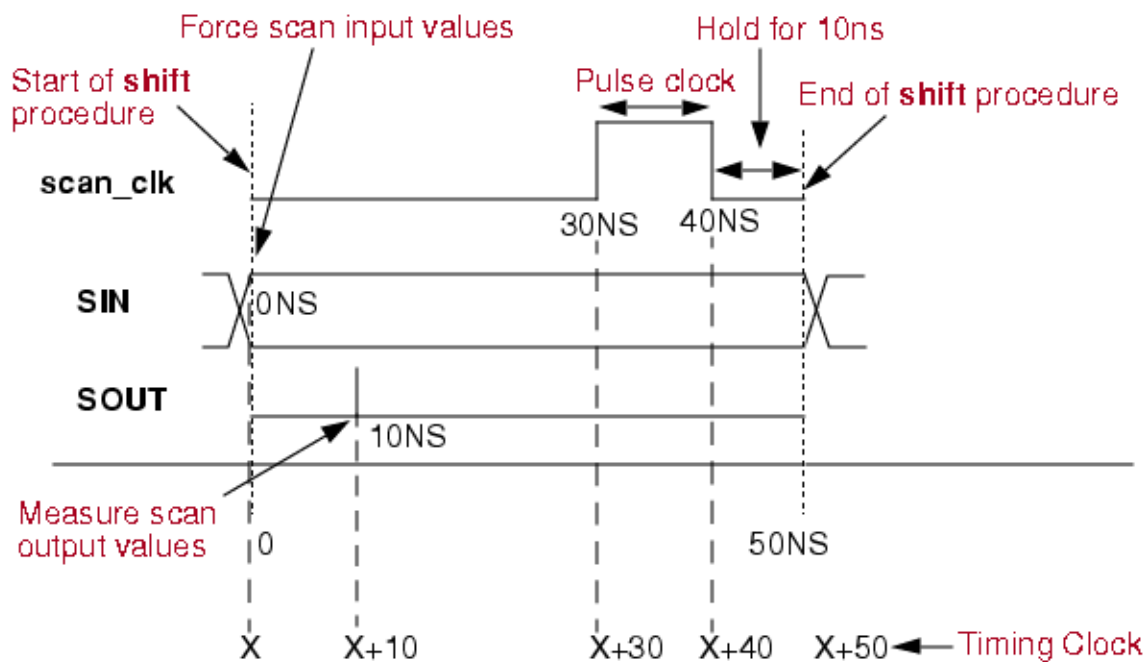
```

procedure shift =
  timeplate tp2;
  cycle =
    // force scan chain input
    force_sci;
    // measure scan chain output
    measure_sco;
    // pulse primary clock
    pulse_scan_pclk;
    // pulse secondary clock
    pulse_scan_sclk;
  end;
end;

```

Figure 10-3 graphically displays the waveforms for the clock pin, the scan-in pin, and the scan-out pin derived from the Mux-DFF shift procedure example. This timing diagram shows one scan chain shift cycle, assuming the time unit is 1ns.

Figure 10-3. Timing Diagram for Shift Procedure



The procedure contains four scan events: forces scan input values at 0ns, strobes (or measures) scan output values at 10ns, pulses the scan clock scan_clk (turning it on at 30ns and off at 40ns), and holds the state of the last event until the procedure finishes at 50ns.

A timing clock monitors when each significant event occurs. If the timing clock is at X when the shift procedure begins, the timing clock assigns those four events with time values X, X+10, X+30, and X+40. When the shift procedure finishes, the timing clock advances to X+50. The shift cycle ending time becomes the starting time for the next shift cycle.

Alternate Shift Procedure (Optional)

When using on-chip clock generators, such as programmable PLLs, it is sometimes necessary to change values on input (control) signals to the clock generator a cycle or two before the change in generated clocking schemes is realized. When the shift clocks for a scan chain are also provided by the on-chip clock generator, it is sometimes not possible to reprogram the clock generator near the end of the scan chain shifting in order to stop the shift clock and prepare for the capture clocks. To accomplish this you might want to use an alternative shift procedure.

Alternate shift procedures have names, as described in the following paragraph. Alternate shift procedures can only be used for single shifts (a pre shift or a post shift), and there must be one un-named normal shift as the main shift in the required load_unload procedure. See [Load_Unload \(Required\)](#).

The shift procedure enables an optional name following the shift procedure type. For each scan group, one shift procedure must be defined that has the default name of shift. For each scan group, additional alternate shift procedures can be defined as long as each has a unique name.

Each shift procedure is required to contain a force_sci or force_sci_equiv statement and a measure_sco statement.

Syntax

```
procedure shift [ procedure_name ] =  
  ...  
end ;
```

Example

The following is a partial example of how the alternate shift procedure might be used in a procedure file for a scan chain with a length of 100.

```
timeplate tp1 =
  force_pi 0;
  measure_po 10;
  pulse ref_clk 50 50;
  period 100;
end;
procedure shift =
  timeplate tp1;
  scan_group grp1;
  cycle =
    force ctrl_a 1;
    force_sci;
    measure_sco;
    pulse ref_clk;
  end;
end;
procedure shift shift_last =
  timeplate tp1;
  scan_group grp1;
  cycle =
    force ctrl_a 0;
    force_sci;
    measure_sco;
    pulse ref_clk;
  end;
end;
procedure load_unload =
  timeplate tp1;
  scan_group grp1;
  cycle =
    force ref_clk 0;
    force scan_en 1;
    force ctrl_a 1;
  end;
  apply shift 98;
  apply shift_last 1;
  apply shift_last 1;
end;
```

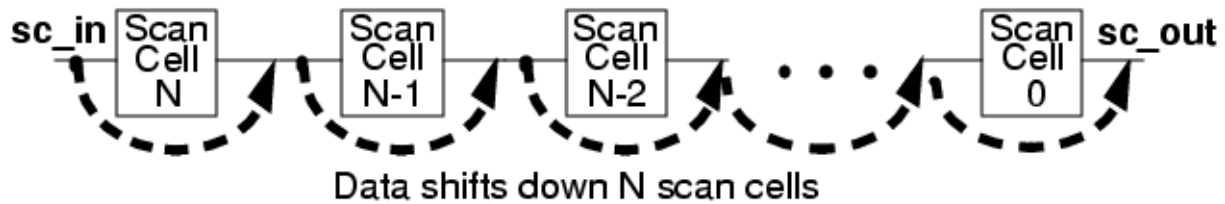
Load_Unload (Required)

This required procedure describes how to load and unload the scan chains in the scan group. To load the scan chain, you must force the circuit into the appropriate state for the start of the shift sequence. This includes forcing clocks, resets, RAM write control signals, and any other signals that need to be at their off states for scan chain loading. Also, if a reset signal is defined as a clock, and pin constrained to its off state in the dofile, it needs to again be forced to its off state in the load_unload and named capture procedures in order to avoid a P34 DRC.

Offstate for clock pins, constrained pin values, and other pins that have values forced in the test_setup procedure are automatically added as force statements to the beginning of the load_unload procedure (if not present); this helps reduce DRC failures.

Figure 10-4 shows the data flow for the load_unload procedure.

Figure 10-4. Load_Unload Procedure



If the scan out pin is bidirectional, you must force its value to the Z state (indicating it is operating in “output” mode) to properly sensitize the scan chain. If there is a scan enable signal, you must force it on to enable the scan chain prior to the shift. You then use the apply shift statement to specify the number of shift cycles (which equals the number of scan elements in the chain). If you have optionally included the shadow_control procedure (which if used, immediately follows the shift procedure), you must also include the apply command.

The following list includes the basic statements in the load_unload procedure:

Mux-DFF Example

```
procedure load_unload =  
  timeplate tp1;  
  cycle =  
    // force clocks off  
    force RST 0;  
    force CLK 0;  
    // activate scanning mode  
    force scan_en 1;  
  end;  
  // shift data thru each of 7 cells  
  apply shift 7;  
end;
```

LSSD Example

```
procedure load_unload =  
  timeplate tp2;  
  cycle =  
    // force all clocks off  
    force RST 0;  
    force CLK 0;  
    force scan_sclk 0;  
    force scan_mclk 0;  
  end;  
  // apply shift procedure 7 times  
  apply shift 7;  
end;
```

The timing for the load_unload procedure is generally straightforward. The load_unload procedure contains the apply statement. Therefore, the total time for a load_unload procedure

includes the time specified by the timeplate being used plus the time required to execute the apply cycles.

For example, examine the following load_unload procedure, using the example shift procedure in the previous section.

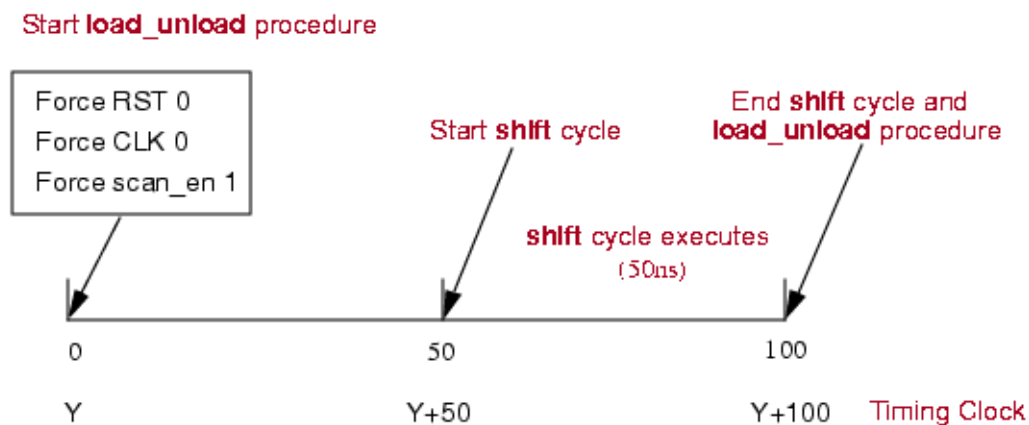
```

procedure load_unload =
  timeplate tp1;
  cycle =
    force RST 0;
    force CLK 0;
    force scan_en 1;
  end;
  apply shift 1;
end;

```

The timeplate of the load_unload procedure specifies the period is 50ns. However, the load_unload procedure includes an apply statement that executes one shift procedure. The shift procedure requires an additional 50ns. Thus, the load_unload procedure actually requires a total time of 100ns, as shown in [Figure 10-5](#).

Figure 10-5. Timing Diagram for Load_Unload Procedure



Within the load_unload procedure, after the completion of the cycle block, the shift procedure starts at 50ns, executes for 50ns, and ends at 100ns. Thus, the load_unload procedure also ends at 100ns.


As with the shift procedure, the timing clock determines the event times for the load_unload procedure. If the timing clock is at Y when the load_unload procedure begins, the first three events happen at time Y. When the apply cycle executes, the timing clock advances to Y+50, which is when the shift procedure begins. As mentioned previously, the shift procedure requires 50 time units. Therefore, when the apply cycle finishes, the timing clock reads Y+100.

Because it is the last event in the load_unload procedure, the end of the apply cycle determines the end of the load_unload procedure.

Shadow_Control (Optional)

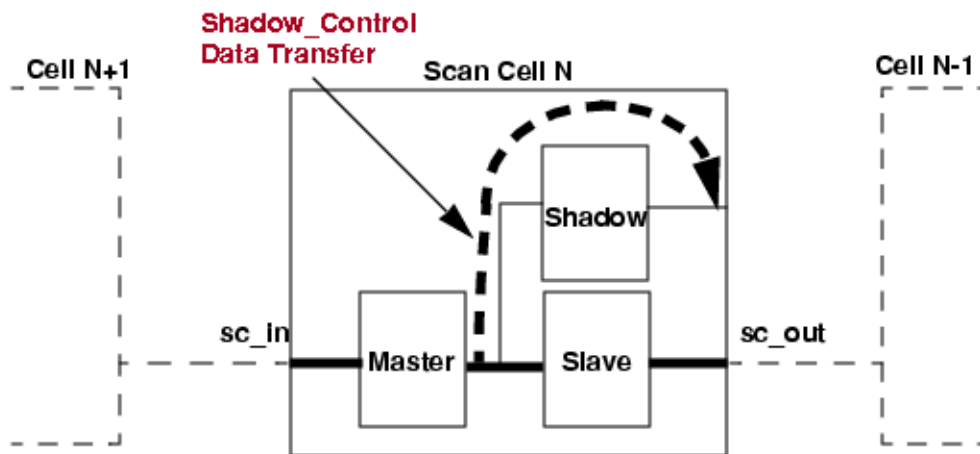
The optional shadow_control procedure, which may only contain force and pulse event statements, describes how to load the contents of a scan cell into the associated shadow.

Note

 This procedure is not supported when using SSN.

If you use this procedure, you must also apply the shadow_control command in the load_unload procedure. This procedure must not disturb the contents of any of the scan cells. [Figure 10-6](#) shows the data flow for the shadow_control procedure.

Figure 10-6. Shadow_Control Procedure



The following procedure file example demonstrates the syntax for applying a shadow_control procedure within a load_unload procedure:

```

proc shift =
    force_sci          0;
    measure_sco       0;
    force SK2         1 1;
    force SK2         0 2;
    force SK1         1 3;
    force SK1         0 4;
end;

proc load_unload =
    force WE          0 0;
    force ABC         1 0;
    force COMB_B     1 0;
    force SK2         0 0;
    force CLK         0 0;
    force SK1         0 0;
    force sh_clk     0 0;
    apply shift      5 1;
    apply shadow_control 1 2;
end;

proc shadow_control =
    force sh_clk      1 1;
    force sh_clk      0 2;
end;

proc master_observe =
    force WE          0 0;
    force SK2         0 0;
    force CLK         0 0;
    force SK1         0 0;
    force sh_clk     0 0;
    force SK1         1 1;
    force SK1         0 2;
end;

proc shadow_observe =
    force WE          0 0;
    force EN          1 0;
    force SK2         0 0;
    force CLK         0 0;
    force SK1         0 0;
    force sh_clk     0 0;
    force CLK         1 1;
    force CLK         0 2;
    force SK1         1 3;
    force SK1         0 4;
end;

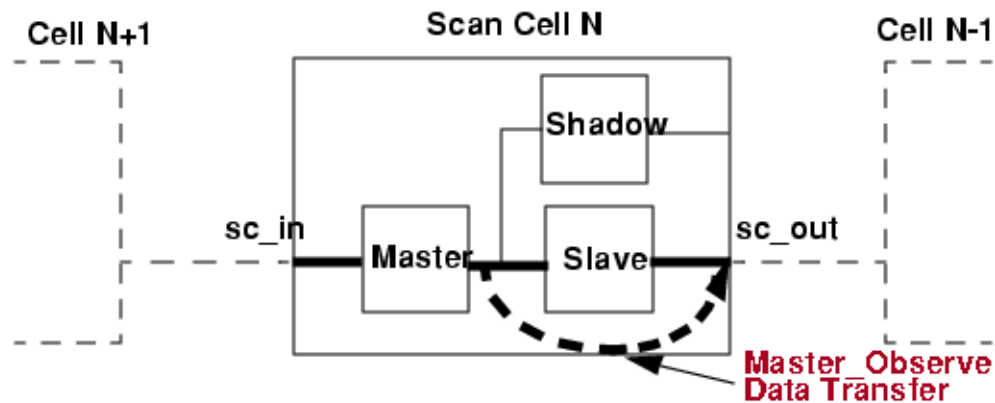
```

Master_Observe (Sometimes Required)

The master_observe procedure, which may only contain force and pulse event statements, describes how to place the contents of a master into the output of its scan cell, where you can observe it by using the unload operation.

Figure 10-7 shows the data flow for the master_observe procedure.

Figure 10-7. Master_Observe Procedure



You do not need to use this procedure if the master element's output is the output of the scan cell. The D1 rule ensures this procedure does not disturb master memory element's contents. You can override this requirement by changing the D1 rule handling. The following example shows a master_observe procedure for the LSSD architecture:

```
// LSSD architecture example
procedure master_observe =
  timeplate tp1;
  cycle =
    // Force all clocks off
    force scan_sclk 0;
    force scan_mclk 0;
    force rst      0;
    force clk      0;
    // Pulse the slave clock
    pulse scan_sclk;
  end;
end;
```

Shadow_Observe (Optional)

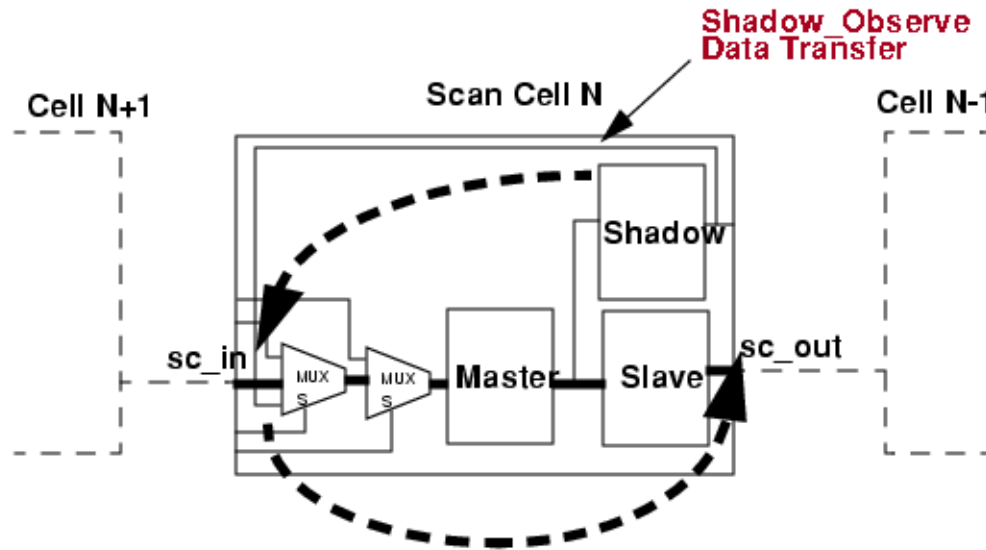
The optional shadow_observe procedure, which may only contain force and pulse event statements, describes how to place the contents of a shadow into the output of its scan cell, assuming that data can be transferred in this way in the scan cell. Once the data is at the scan cell output, you can observe it by applying the unload command. This procedure allows the shadow to be used as an observation point in the design.

Note

This procedure is not supported when using SSN.

Figure 10-8 shows the data flow of the shadow_observe procedure.

Figure 10-8. Shadow_Observe Procedure



Skew_Load (Optional)

The optional skew_load procedure propagates the output value of the preceding scan cell into the master memory element of the current cell without changing the slave, for all scan cells. Using only force and pulse event statements, this procedure defines how to apply an additional pulse of the master shift clock after the scan chains are loaded.

Figure 10-9 shows the data flow of the skew_load procedure.

Figure 10-9. Skew_Load Procedure

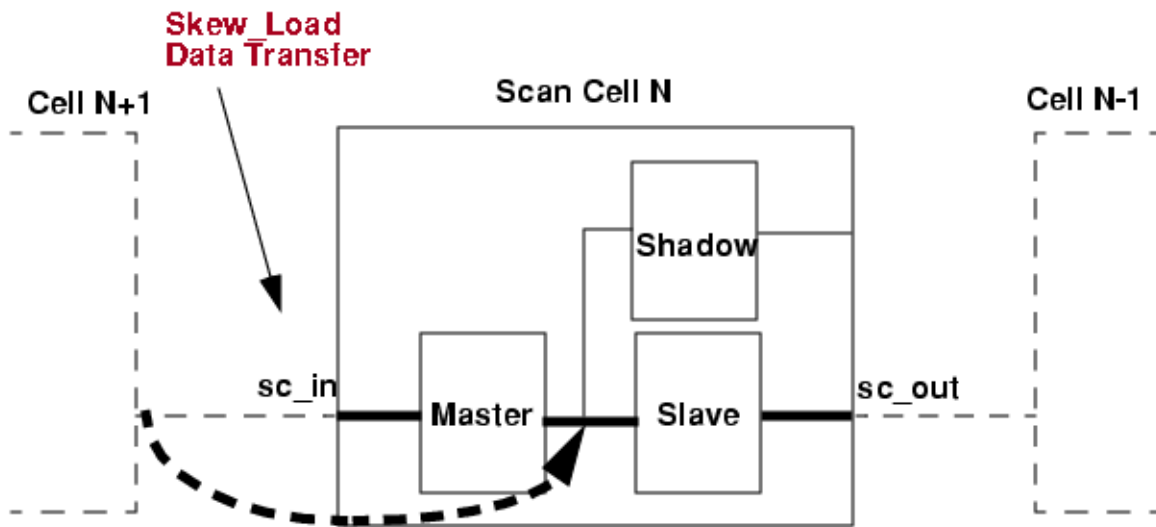
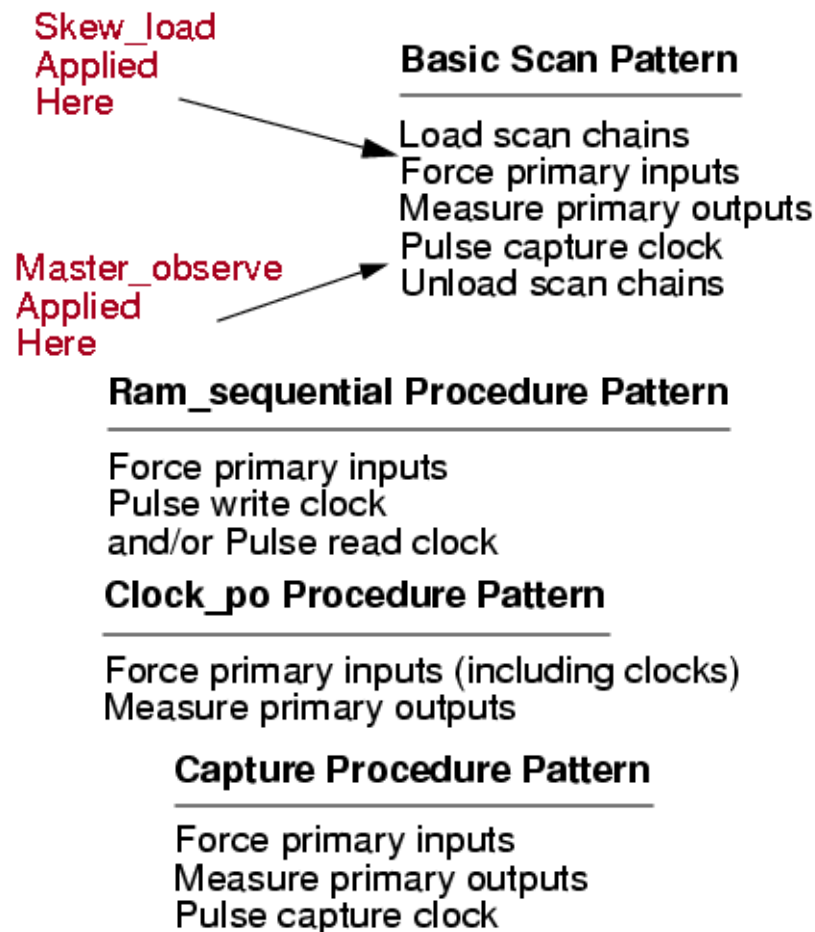


Figure 10-10 shows where you apply the skew_load procedure and the master_observe procedure within the basic scan pattern events.

Figure 10-10. Skew_load applied within Pattern



Clock_run (Optional)

For every controller, or concurrent controller group, you can write a `clock_run` procedure, if needed. The `clock_run` procedure has both an internal mode as well as an external mode.

You can specify only one `clock_run` procedure per controller or concurrent group; however, you do not need to specify a separate procedure for each controller instance. The same procedure can be used for multiple controllers. You need to specify a separate procedure for a controller instance only if it maps to a different set of internal clocks.

In case of controllers running concurrently, and some of these controllers clocks are driven by PLL internal clocks, the `clock_run` procedure is required per concurrent group. It is not required for every BIST controller participating in the group to have its clock driven by a PLL internal clock. For some controllers, their clocks can be driven by a PLL reference clock or even by a system clock.

The tool relies on you to control the PLL control signal. This can be achieved by forcing the PLL control signal to a proper value in a test_setup procedure and in external mode of clock_run procedure as well (it depends on the PLL model behavior).

A clock_run procedure has to have a N-to-1 or 1-to-N ratio between internal and external cycles; that is, either the internal mode has to have only one cycle, or the external mode has to have only one cycle. You cannot have, for example, two external cycles and three internal cycles.

Capture Procedures (Optional)

There are three types of capture procedures. These procedures are optional in the “patterns -scan” context.

- The default capture procedure is an optional capture procedure, without a name, that provides information on how the series of capture events are broken into cycles and which timeplates these cycles use. The default capture procedure is defined in the profile as part of the scan group definition or internally derived by the tool when you do not define one. If you need to create or edit a default capture procedure, see “[Clock_po \(Optional\)](#)” in this chapter.
- The named capture procedure is an optional capture procedure, with a name, that is used to define explicit clock cycles. You can create multiple named capture procedures, each with a unique name, using the [create_capture_procedures](#) command. If you need to manually create or edit named capture procedures, see “[Rules for Creating and Editing Named Capture Procedures](#)” in this chapter. For information on using named capture procedures to create at-speed test patterns, see “[At-Speed Test With Named Capture Procedures](#)” in the *Tessent Scan and ATPG User’s Manual*.
- The external_capture procedure is an optional capture procedure used for all capture cycles between each scan load, even when the pattern is a multi-load pattern. External_capture procedures are used by the “[set_external_capture_options -capture_procedure](#)” command. External_capture procedures that are used with this command have several restrictions:
 - The procedure can only have one force_pi statement and no measure_po statements. This is because to use the “[set_external_capture_options -capture_procedure](#)” switch, the patterns to be saved must be hold_pi and mask_po patterns. The statements in the capture procedure must match up to this.
 - Unlike named capture procedures, the external_capture procedure cannot have any load_cycles as it is meant to be used between each scan load of a pattern. The external_capture cannot contain any events on internal signals.
 - When using the -capture_procedure switch with set_external_capture_options, all clocks in the design that are internally connected (don't trace to just a cut point), must be controlled. In addition, the clocks must either be constrained, always-capture, controlled by a clock_control definition, or the clock must have an event in the external_capture procedure.

Rules for Creating and Editing a Default Capture Procedure	594
Rules for Creating and Editing Named Capture Procedures	594
Slow and Load Types in the Cycle Statement	596
launch_capture_pair Statement	597

Rules for Creating and Editing a Default Capture Procedure

There are several issues to consider when working with default capture procedures.

- The default procedure may only contain `force_pi`, `measure_po`, `pulse_capture_clock`, `bidir_force`, `bidir_force_pi`, `bidir_force_off`, and `bidir_measure_po` event statements that represent the non-scan activity for a normal pattern. There is no overlap between the capture procedure and the existing clock procedure.
- Use the `pulse_capture_clock` statement in the default capture procedure to indicate in which cycle one or more capture clocks should be pulsed.
- Do not specify any complex clocking that needs to be described for capture clocks or other clocks in the default capture procedure; specify it in the clock procedure or by using a named capture procedure.
- Do not specify any type of pin or ATPG constraint in the default capture procedure. For example, specifying that a certain pin is to be held at a certain state in the default capture procedure does not restrict the ATPG engine from applying different values to that pin. However, you can use the `bidir_force` and `bidir_force_pi` statements in the default capture procedure to force all bidirectional pins off in one cycle and force the ATPG values on the bidirectional pins in the next cycle.

Rules for Creating and Editing Named Capture Procedures

There are several issues to consider when working with named capture procedures.


- A named capture procedure may only contain `force_pi`, `measure_po`, `observe_method`, `pulse` (named clock), and `condition` statements.
- If you use mode definitions, all cycles in a procedure must be defined within mode definitions. Use the keyword “mode” with two mode blocks: “internal” and “external”. Use the `mode_internal` definition to describe what happens on the internal side of the on-chip PLL. Use the `mode_external` definition to describe what happens on the external side of the on-chip PLL.
- All events in a named capture procedure that use modes must be duplicated in both modes. The only difference is that the internal mode uses only internal clocks and the external mode uses only external clocks. The number of cycles and timeplates used can be different as long as the total period of both modes is the same.
- Signal events used in both internal and external modes must happen at the same time. Examples of these events are `force_pi`, `measure_po`, and other signal forces, but also include clocks that can be used in both modes.

- If a `measure_po` statement is used, it can only appear in the last cycle of the internal mode and must occur before the last clock pulse. If no `measure_po` statement is used, the tool issues a warning that the primary outputs cannot be observed.
- The cumulative time from the start of the first cycle to the `measure_po` must be the same in both modes.
- The external mode cannot pulse any internal clocks or force any internal control signals.
- A `force_pi` statement needs to appear in the first cycle of both modes and occur before the first pulse of a clock.
- If an external clock goes to the PLL and to other internal circuitry, a C2 DRC violation is issued.
- At-speed cycles need to be continuous; that is, a named capture procedure cannot have more than one at-speed clocking subsequence.
- All defined real clocks (excluding internal clocks) must be forced to off state first in the `mode_internal` definition.

For more information, see “[Internal and External Modes Definition](#)” in the *Tessent Scan and ATPG User’s Manual*.

- Do not use the `pulse_capture_clock` statement in a named capture procedure. The clocks used are explicitly pulsed.
- If you want to specify the internal conditions that need to be met at certain scan cells in order to enable a clock sequence, use the `condition` statement at the beginning of the cycle statement in the named capture procedure.
- If you want to define a specific observe method for each named capture procedure, use the `observe_method` statement in the named capture procedure; otherwise, the ATPG engine automatically selects master, slave, or shadow observation.

Note

 The `write_patterns` command enables you to save internal or external clock patterns. Internal clock patterns can be used to simulate the DUT without having the PLL modeled, while the external patterns only exercise the PLL external clocks and control signals. Internal patterns are the default for ASCII and binary formats, and external patterns are the default for tester formats.

- If you generate patterns using a named capture procedure that has both internal and external modes and you save them in STIL or WGL format, you must use the `write_patterns` command’s “internal” option to read them back into the tool (for example, to use in diagnosis). For more information and for information about special considerations that apply to LBIST mode in the TK/LBIST Hybrid flow, refer to the `-Mode_internal` and `-Mode_external` switches for the `write_patterns` command in the *Tessent Shell Reference Manual*.

DRC rules W20 through W36 check named capture procedures. If a DRC error prevents use of a capture procedure, the run aborts.

Slow and Load Types in the Cycle Statement


Optionally, you can add a “slow” or a “load” type to the cycle definition.

For example:

```
cycle slow =  
...  
end;
```


- The slow cycle indicates that at-speed faults cannot be launched or captured. The tool must know which at-speed cycles are slow to get accurate at-speed fault coverage simulation numbers; therefore, be sure to include “slow” when defining cycles that are not at-speed cycles in an at-speed capture procedure.

Note

 At-speed cycles need to be continuous; that is, a named capture procedure cannot have more than one at-speed clocking subsequence.

- The load cycle indicates that the cycle is always preceded by an extra scan load. The first cycle in a named capture procedure is always a load (with or without the load type designation), so you typically apply “load” to subsequent cycles. An at-speed launch cycle can be a load cycle; however, none of the cycles that follow in the at-speed sequence, up to and including the capture cycle, can be load cycles.

Note

 To get extra loads, you must enable the tool’s multiple load and clock sequential capabilities by issuing the [set_pattern_type](#) command with “-multiple load on” and “-sequential <2 or greater>”. For more information, see “[Multiple Load Patterns](#)” in the *Tessent Scan and ATPG User’s Manual*.

The following example illustrates the “slow” and “load” attributes:

```
procedure capture multi_load_example =
  timeplate tp1;
  // first cycle is always a load, with or without load type designation
  cycle slow =
    force_pi;
    force wr_enable 1;
    pulse int_clk1;
  end;
  cycle slow load =
    pulse int_clk1;
  end;
  cycle =
    force re_enable 1;
    pulse int_clk1; // launch clock
  end;
  cycle =
    pulse int_clk1; // capture clock
  end;
end; // end of capture procedure
```

launch_capture_pair Statement

Optionally, you can add one or more “launch_capture_pair” statements to the beginning of a named capture procedure. This statement defines legal at-speed launch and capture points in non-adjacent cycles. If you do not use the launch_capture_pair statement, the tool launches and capture only in adjacent cycles. If at least one launch and capture clock pair is defined, the launch and capture points are derived from the defined launch and capture clock pairs.

Note



This statement is only supported when using a named capture procedure to perform test generation.

The syntax of the launch_capture_pair statement is as follows:

```
launch_capture_pair <launch_clock_pin_name> <capture_clock_pin_name>;
```

Where:

- launch_clock_pin_name is the clock used to launch the transition.
- capture_clock_pin_name is the clock used to capture the transition.

The launch clock cycle is used to check the transition condition. The capture clock cycle is used to capture the transition fault effect. The cycles between the launch clock and capture clock must be at-speed cycles. They cannot include any slow cycles between them. The faults to be tested by the named capture procedure with the defined launch and capture clock pair are the faults that can be launched by the launch clock and captured by the capture clock defined in the launch_capture_pair statement.

The following is an example of the `launch_capture_pair` statement:

```
procedure capture c1_c1 =
  launch_capture_pair c1 c1;

  cycle = // cycle 1
    force_pi;
    force c1 0;
    force c2 0;
    force c3 0;
    pulse c1;
  end;

  cycle = // cycle 2
    pulse c2;
  end;

  cycle = // cycle 3
    pulse c1;
    pulse c3;
  end;
end; // end of capture procedure
```

In this example, a valid launch can happen in cycle 1. A valid capture can happen in cycle 3 only with `c1` as the capture clock. A launch in cycle 1 and a capture in cycle 2 is not used for fault detection. The faults to be tested by this named capture procedure are the faults that can be launched and captured by clock `c1`.

Clock_po (Optional)

The `clock_po` procedure (optional in “patterns -scan” context), which can contain only `force_pi`, `measure_po`, `bidi_force_pi`, and `bidi_force_off` event statements, represents the non-scan activity for a clock PO pattern. Use this procedure instead of the capture procedure.

Note



This procedure is not supported when using SSN.

With this procedure, you must use a timeplate that does not pulse the clocks.

The following shows the pattern for the `clock_po` procedure pattern.

Clock_po Procedure Pattern

Force primary inputs (including clocks)
Measure primary outputs

Clock_sequential (Optional)

The clock_sequential procedure (optional for “patterns -scan” context), which may only contain force_pi, pulse_write_clock, pulse_read_clock, pulse_capture_clock, bidi_force_pi, and bidi_force_off event statements, represents the clock sequential events in a clock sequential pattern. Use this procedure with the capture procedure.

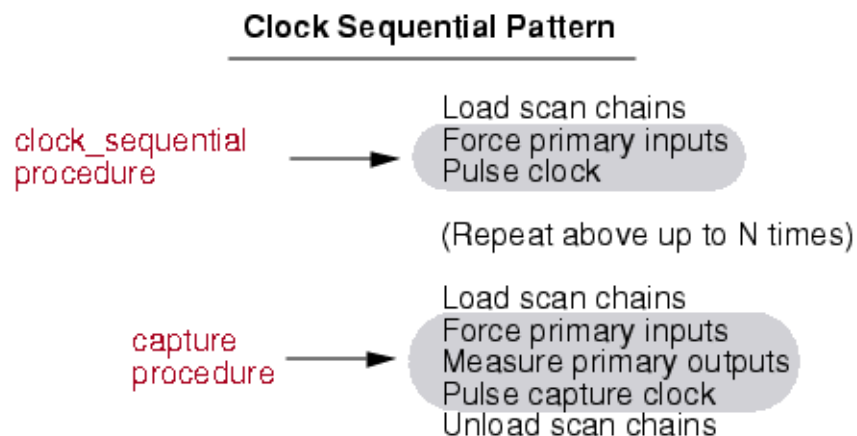
The following shows the clock_sequential procedure pattern.

Clock_sequential Procedure Pattern

**Force primary inputs
Pulse write clock
and/or Pulse read clock
and/or Pulse capture clock**

Figure 10-11 shows an entire clock sequential pattern, which illustrates where the clock_sequential and capture procedures are used.

Figure 10-11. Full Clock Sequential Pattern



Init_force (Optional)

The init_force procedure (optional for “patterns -scan” context), which may only contain force_pi event statements, represents the force cycle that is used in an ATPG pattern that targets a transition fault. The transition must be launched off of the last scan chain shift. This procedure is used when the fault type is set to transition fault and either the depth is set to 2 or less or the ATPG engines fail to find a sequential pattern that can cover this transition fault. Use this procedure with the capture procedure.

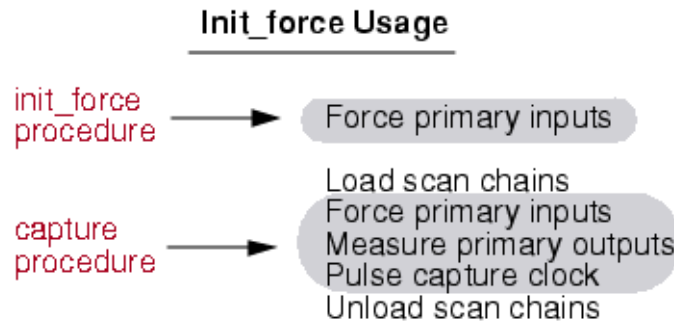
The following illustrates the format of the `init_force` procedure pattern.

Init_force Procedure Pattern

Force primary inputs

Figure 10-12 shows the pattern which uses the `init_force` procedure.

Figure 10-12. Init_force Procedure Usage



Test_end (Optional, all ATPG tools)

The optional `test_end` procedure is used to add a sequence of events to the end of a test pattern set.

The `test_end` procedure may only contain force and pulse event statements (see the following exception), and can only be defined once for all scan groups. When saving patterns, the `test_end` procedure is applied to the end of each pattern set saved.

The following shows the general pattern for the `test_end` procedure pattern.

Test_end Procedure Pattern

Force primary inputs
Pulse clocks

IJTAG Embedded Instruments

In the ATPG context patterns -scan, IJTAG is valid in the `test_setup` and `test_end` procedures only. It is invalid in any other procedure, as well as in ATPG's analysis mode. Also, only `iReset` and `iCall` commands are valid in the test procedures.

For detailed information, see “[How to Set Up Embedded Instruments Through Test Procedures](#)” in the *Tessent IJTAG User's Manual*.

Example 1: Using test_end in a Procedure File

The following is a partial example of how the test_end procedure might be used in a procedure file:

```
timeplate tp1 =  
  force_pi 0;  
  measure_po 10;  
  pulse ref_clk 50 50;  
  period 100;  
end;  
  
procedure test_end =  
  timeplate tp1;  
  cycle =  
    force ctrl_a 1;  
    force tms 0;  
    pulse ref_clk;  
  end;  
end;
```

Example 2: Test_end Procedure with Timeplate

The following is an example test_end procedure with its corresponding timeplate:

```
timeplate tp4 =
    force_pi 0;
    pulse TCK 10 10;
    measure_po 30;
    period 40;
end;

procedure test_end =
    timeplate tp4;
    cycle =
        // TMS = 1, change to select-DR state
        force TDI 1;
        force TMS 1;
        pulse TCK;
    end;

    cycle =
        // TMS = 0, change to capture-DR state

    ...
    cycle =
        // Scan out signature (MISR has length of 4)
        force TDI 1;
        force TMS 0;
        pulse TCK;
    end;
    cycle =
        force TDI 1;
        force TMS 0;
        pulse TCK ;
    end;
    ...
end;
```

Sub_procedure

The sub_procedure procedure eliminates the need to insert duplicate actions within a procedure. Once you have defined a sub_procedure, you can specify this procedure within other procedures using the apply statement.

You can also set the tool to reissue the sub_procedure as many times as needed by specifying the repeat_count. Because the repeat_count is required when using apply sub_procedure, you must enter a minimum of 1 for this parameter.

Sub_procedure Looping

Sub_procedure looping is used to reduce the size of pattern files. The default behavior of the sub_procedure is to use “loops” or “repeats” in all applicable pattern formats to repeat the contents of the sub_procedure N times, where N is greater than 1.

Disabling Sub_procedure Looping

If you want the event data in a sub_procedure to be expanded and represented as N sets of vectors in the pattern file, where N is the number of times the sub_procedure is applied, use the “[ALL_NO_LOOP 1](#)” parameter file keyword to disable the use of “loop” or “repeat” statements.

For example, if the test_setup procedure has the following statement:

```
apply pulse_bclock 1000;
```

The vector data for the sub_procedure “pulse_bclock” would be expanded to be 1000 vectors. The default for the ALL_NO_LOOP keyword is off (0).

Sub_procedure Definition Format

The sub_procedure definition has the following format.


```
procedure sub_procedure my_subprocedure =  
  timeplate tp1;  
  cycle =  
    force_pi;  
    measure_po;  
  end;  
end;
```

Using the Sub_procedure in a Procedure

The following is an example of how to use the sub_procedure in a procedure.

```
procedure shift =  
  scan_group grp1;  
  timeplate tp1;  
  apply my_subprocedure 4;  
  cycle =  
    force_sci;  
    measure_sco;  
    pulse T;  
  end;  
end;
```

Note

 You must first define a sub_procedure before using it in a procedure. Next, you can apply a sub_procedure within any procedure type. Also, you cannot use a sub_procedure within the “cycle =” and “end;” statements.

Additional Support for Test Procedure Files

Tessent Shell provides additional support so that you can use environmental variable, merge, default template capabilities with your test procedure files.

Tcl Variables Used for Substitution and Conditional Selection

The procfile can include Tcl variables to provide parameterized values within the procfile statements. The Tcl variables must be set in the global Tcl namespace. If you are setting the variables within a proc, make sure to use `::` as a prefix to the variable name such that it is set in the global namespace. For example, use `"set ::my_period 4"` such that `$my_period` exists when processing the proc files.

The Tcl variable can also be used to evaluate the condition of a Tcl if command located inside the procfile. The element of a list of ports in the procfile must be separated by commas. If the port names are escaped identifiers, they must be enclosed in quotes. Use the method shown in the following example to convert a list of port names into a quoted comma-separated list needed in the proc file:

Command invoked in the dofile:

```
set ::add_clocks_timing 1
set ::edges "25 75"
set ::port_list [get_name_list [get_clocks -type sync_source]]
set ::all_clocks [string cat {"} [join $port_list {" , "}] {"}]
set _procfile_name ./myproc
```

Given the content of the `./myproc` is:

```
alias all_clocks = $all_clocks;
timeplate mytimeplate =
  if {$add_clocks_timing} {
    pulse all_clocks $edges;
  }
end;
```

The `report_procedures` command displays the following:

```
alias all_clocks = "clk1", "clk2", clk3";
timeplate mytimeplate =
  pulse all_clocks 25 75;
end;
```

Merging Procedure Files

It is possible to specify more than one procedure file for a design. You can specify a procedure file with the `add_scan_groups` command or with the `read_procfile` command. You need to supply (to the ATPG tool) a minimum set of information in the procedure file with the `add_scan_groups` command. You must supply all event information for the scan procedures.

However, after leaving setup mode, it is possible to specify non-scan procedures, timeplates, and new timing for the scan procedures by reading in an additional procedure file with the `read_procf` command. Specifying new information for the same design, from more than one procedure file, is known as “merging the procedure files.” To properly merge the information from multiple procedure files, the Vector Interfaces code follows these rules:

- All scan procedures that you use must be specified in the procedure file that you load with the `add_scan_groups` command.
- If you load a procedure that contains nothing but the procedure name, a timeplate name, and an optional scan group, it is a template procedure. If a procedure already exists by that name for that scan group (if it is a group-specific procedure), then the timeplate is mapped onto the existing procedure. If no procedure already exists with that name, the tool stores the template procedure for future use.
- If you load a new complete procedure (not a template) and a procedure already exists by that name for the specified scan group (if applicable), the new procedure overwrites the existing one.
- In both cases, when a procedure overwrites an existing one, or if a new timeplate is mapped to an old procedure, the tool checks the procedures to make sure that the sequence of events in the new procedure does not differ from the old procedure.

Default Information Provided by the Tool

When you issue the `write_patterns` command, the tool checks to make sure that all procedures and timeplates needed to save the patterns in the specified format are present.

If there are any missing non-scan procedures, the tool creates default procedures and issues a warning. For example, in cases where there are `ram_sequential` patterns that need to be saved and no `ram_sequential` procedure was supplied, the tool automatically creates a default procedure.

For any procedures that are created or that do not have a timeplate specified, the default timeplate is mapped to these procedures, if it is set. You can set the default timeplate by using the `set default_timeplate` statement previously described in the “[Set Statement](#)” section. If you use this statement, the timeplate specified when creating default procedures is used. If the default procedure needs to be created and no default timeplate has been set, then the first timeplate specified is used. If no timeplates are specified, a default timeplate is created as well.

Creating Test Procedure Files for End Measure Mode

You can create test procedure files that enable end measure mode. End measure mode refers to the special handling that the Vector Interfaces code needs to move the measure to the end of the shift and capture cycle.

Prerequisites

- A test procedure file.

Procedure

1. Create a new timeplate that measures the outputs after the clock pulse.
2. Change the timeplate for the shift and load_unload to point to the new timeplate.
3. Add the measure_sco statement to the load_unload procedure.
4. Make sure all shift procedures have the measure_sco statement after the shift clock. When end measure mode is enabled, the measure_sco statement measures the next value from the output of the scan chain. The very first value for the output of the scan chain is measured by a measure_sco statement in the load_unload procedure.
5. Change the timeplate for the capture cycle by breaking it into two cycles. Move the capture clock to the second cycle of the capture procedure to allow the measure at the end. In the first cycle, the force_pi and measure_po are performed. In the second cycle, the capture clock is pulsed. When using end measure mode, a measure cannot be performed after the capture clock.

Examples

```

set time scale 1.000000 ns ;
set strobe_window time 10 ;
timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 10 ;
    pulse clk 20 10;
    pulse edt_clock 20 10;
    pulse ramclk 20 10;
    period 40 ;
end;
// CREATE A NEW TIMEPLATE THAT MEASURES AFTER THE CLOCK PULSE
timeplate gen_tp2 =
    force_pi 0 ;
    //      measure_po 10 ;
    pulse clk 20 10;
    pulse edt_clock 20 10;
    pulse ramclk 20 10;
    measure_po 35 ;          // <<== NEW MEASURE STATEMENT
    period 40 ;
end;
// FOR CAPTURE SPLIT INTO TWO CYCLES
procedure capture =
    timeplate gen_tp1 ;
    cycle =
        force_pi ;
        measure_po ;
    end ;
    cycle =
        pulse_capture_clock ;
    end;
end;
// FOR THE SHIFT AND LOAD_UNLOAD, USE THE NEW TIMEPLATE
procedure shift =
    scan_group grp1 ;
    timeplate gen_tp2 ; //<<=== NEW TIMEPLATE
    cycle =
        force_sci ;
        force edt_update 0 ;
        measure_sco ;
        pulse clk ;
        pulse edt_clock ;
    end;
end;
// ADD A MEASURE_SCO TO THE LOAD_UNLOAD PROCEDURE
procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp2 ; //<<=== NEW TIMEPLATE
    cycle =
        force clk 0 ;
        force edt_bypass 0 ;
        force edt_clock 0 ;
        force edt_update 1 ;
        force ramclk 0 ;
        force scan_en 1 ;
        pulse edt_clock ;
    measure_sco;          //<<=== NEW MEASURE STATEMENT

```

```
    end ;  
    apply shift 39;  
end;  
procedure test_setup =  
    timeplate gen_tpl ;  
    cycle =  
        force edt_clock 0 ;  
    end;  
end;
```


Serial Register Load and Unload for LogicBIST and ATPG

You can use the `test_setup` or `test_end` test procedure file procedures to serially load or unload control registers in the circuit under test. Additionally, this section discusses the commands you must add to your dofile to use this functionality.

Serial register load and unload targets the following flows:

- LogicBIST — Used to serially unload certain registers (for example, MISRs) using the `test_end` procedure.
- Low pin count test — Used to serially load registers with test information through a serial access port such as a JTAG TAP controller.

Register Load and Unload Use Models	609
Static Versus Dynamic Register Variables	609
Test Procedure File Modifications	610
Dofile Modifications	613
Serial Load and Unload DRC Rules	616

Register Load and Unload Use Models

Using serial register load and unload enables you to identify a load/unload register variable value in the test procedure file and define this variable, either static or dynamic, using commands in your Tessent dofile. By using this functionality, you can load control registers by using a type of load procedure where the data value for the register can be passed as a string of values, and the load register uses a shift mechanism to show how this string of data values is shifted into the register.

To serially load/unload register value variables, you must make modifications to your test procedure file and your Tessent dofile.

The register value variable is used in the test procedure file to load the value into a register through the data input pin, or unload a value from a register through the data output pin. The load/unload procedures support pin inversions, but you must explicitly specify this.

The registers to be loaded/unloaded can be cascaded such that one load or unload operation loads or unloads multiple values, for example PRPG/MISR values.

Static Versus Dynamic Register Variables

You can define both static and dynamic register value variables.

- Static variable — A specific register value variable string you specify during setup mode. Once set, you cannot change this variable.
- Dynamic variable — A register value variable string that you can define or change later, and can use tool-specific switches.

Static Variable

A static register variable is one where the value is assigned to the variable when the variable is defined (using the [add_register_value](#)), and this value then cannot be changed. This static value is used by DRC when simulating the procedures. A static variable cannot be set using tool specific switches to link the variable to tool computed values, as these may change.

Dynamic Variable

A dynamic register value variable uses tool-specific switches and arguments to link the variable to a value computed by the tool. If you use a dynamic variable, then you must specify the register's width unless the tool knows this value at the time DRC is run (for example, PRPG size).

This method is used for defining a variable that has a tool-specific computed value that is available when patterns are saved and needs to be loaded or unloaded by the `test_setup` or `test_end` procedures. The value defaults to all X bits, and you can specify the actual value in non-Setup mode by using the `set_register_value` command.

If no value is specified the first time DRC is invoked after adding a register value variable, the tool considers the variable to be dynamic for DRC and any future invocation of DRC.

Test Procedure File Modifications

In the test procedure file, the `load_unload_registers` Procedure and `shift` Keyword are used.

load_unload_registers Procedure

The `load_unload_registers` procedure is a named procedure; consequently, you can have multiple occurrences of this procedure, corresponding to multiple groups of registers that need to be loaded or unloaded. The `load_unload_registers` procedure can only be called from the following procedures:

- `test_setup`
- `test_end`

The `load_unload_registers` procedure can load, unload, or both. Additionally, one application of the procedure can load/unload multiple cascaded registers.

When the test procedure file explicitly calls the `load_unload_registers` procedure from either the `test_setup` or `test_end` procedures, the values to load or unload are passed to the procedure using the string of binary values (value hard-coded in the procedure application) or the register value variables.

shift Keyword

The `load_unload_registers` procedure uses the `shift` keyword to define a block of events that result in one shift operation. This is similar to the IEEE STIL syntax where the `shift` keyword defines a shift block within a load or unload procedure. It is analogous to embedding the shift procedure into the `load_unload` procedure.

Apply Statement Extension

The `apply` statement in the procedure file syntax is extended to accept a statement that associates a set of string values or register value variables with a particular input pin, output pin, or alias. This pin or alias must then be used within the shift statement and have a “#” character associated with it. This character is a placeholder for the values that is loaded or unloaded using the string of values passed to the procedure, or the internally generated values associated with the value name.

Test Procedure File Syntax

The following illustrates using the `load_unload_registers` procedure and `shift` keyword:

```

procedure load_unload_registers procedure_name =
    ...
    [ cycle blocks ]
    shift =
        cycle blocks
    end ;
    [ cycle blocks ]
end ;

```

The `load_unload_registers` procedure must have a shift block defined, which has one or more cycles used to shift the data into the data input pin or the data out of the data output pin. The procedure can also optionally have cycles which precede or follow the shift block, to be used to put the circuit into shift mode or finish the shift mode when done, similar to how a `load_unload` procedure is used.

Event Statements

Within a shift block, at least one event statements in a `load_unload_registers` procedure must use the “#” character to denote where the shift data passed into the procedure is used.

The event statement must be either a “force” event or an “expect” event. An event statement with the “#” character can also occur in the cycles preceding or following the “shift” block to

express pre-shifts and post-shifts for loading the register. This event statement has the following syntax:

```
<force | expect > pin_or_alias_name # ;
```

The apply statement which is currently used to call shift and sub_procedure procedures and also calls the load_unload_registers procedure. When calling these procedures, however, the number of times argument in the apply statement is replaced with one or more value assignments to the data in or data out pins.

```
apply procedure_name [ #times | shift_data_assignment  
[ , shift_data_assignment ...] ] ;
```

A shift_data_assignment has the following syntax:

```
identifier = < value_string | register_value_variable >  
[<value_string | register_value_variable>...]
```

where identifier is either a pin name or an alias name.

The identifier used in the shift_data_assignment must match an identifier used within the procedure in one of the event statements with the “#” character.

Register Value Strings

A register value string (value_string) is one of the following:

- A binary string of 0's, 1's or X's, where the length of the string determines the number of shifts to load the register.
- A string of a different radix as long as the Verilog syntax of identifying the radix and width are used, such as “32'h” for a 32 bit hexadecimal value.

If a register_value_variable is used in the shift_data_assignment, then a value computed by the tool for that register_value_variable (as bound within the dofile) or hard-coded in the dofile is loaded or unloaded at that time and the shift length is also provided by the tool. It is possible for more than one value_string or register_value_variable to be assigned to an identifier in one shift_data_assignment. In this case, the extra values are separated by spaces. This enables multiple shorter values to be shifted into one register group.

The typical usage is that each register_value_variable corresponds to one register being loaded, and the specification of multiple variables is used when those registers are cascaded and loaded/unload on after another.

Alias Names

It is possible to use an alias name in the procedure type for loading shift data, even if that alias name refers to multiple pins. If this is the case, the number of bits assigned by the “#” character

for each shift is equal to the width of the alias being used. The length of the value string being passed to the procedure must be a multiple of the width of the alias.

Loading or unloading an alias can be used if performing parallel load/unload and no shift is required. For example, if each MISR bit is connected to a separate primary output. Even if no shifting is required, the functionality is still useful to bind the expected values to the signature computed by the tool.

If multiple `shift_data_assignments` are passed to a procedure, then all of them must have the same shift length such that each pin being loaded/unloaded requires the same number of cycles to load/unload all the data. No padding is performed by the tool.

If a “measure” event is used in one of these procedures to unload a register, then these measure values can be compared in the final patterns and the Verilog test bench.

Dofile Modifications

You define register value variables using commands you issue to the tool interactively or in a dofile. The value variables are subsequently referenced in the procedure file.

You use the following commands to perform these operations:

- `add_register_value`
- `set_register_value`
- `delete_register_value`
- `report_register_value`

Addition of a Register Value Variable

The `add_register_value` command defines the register value variables. You can define the variables as either static value variables or dynamic value variables.

You can only use this command in setup mode. You must define the register value variables in the dofile before the variables are used in a test procedure file to load values into the registers.

Static Value Variables

You specify a static value variable by stating a specific value string. This value variable then has this value string as a constant value.

You must specify this value in setup mode; the value is considered to be static by default, and the value is present when simulating procedures in DRC.

```
add_register_value value_name {value_string [-Radix {Binary | Decimal |  
Octal | hexadecimal} -Width integer] optional_arguments
```

The *value_name* is a user-specified identifier, and the *value_string* is a state string in a particular radix. The default is binary radix.

If the `-Radix` switch is used to change this to a different radix, then a register width must also be specified using the `-Width` switch.

Dynamic Value Variables

You specify a dynamic value variable using the following variation of the command:

```
add_register_value value_name value_string -Dynamic -Width integer  
optional_arguments
```

The *value_string* is optional and defaults to all X bits if not specified.

You can subsequently enter the actual value once you have exited setup mode using the [set_register_value](#) command. If no value is specified the first time DRC is invoked after adding a register value variable, it is considered as dynamic in this and any future invocation of DRC.

Data Pin Inversions

When using either a static or dynamic variable, you can optionally specify inversions on the input and output data pins by using the `-INput_pin_inversion` and `-OUtput_pin_inversion` switches, respectively.

These are optional switches you use to specify that the data value in this variable should be inverted before it is loaded into the register through the `load_unload_register` procedure.

Definition of a Dynamic Register Value Variable

You can define a dynamic register value variable by using tool-specific switches and arguments to link the variable to a value computed by the tool. These variables are dynamic, and the width must be specified unless the width is known to the tool at the time DRC is run.

This method is used for defining a variable that has a tool-specific computed value that is available when patterns are saved and needs to be loaded or unloaded by the `test_setup` or `test_end` procedures.

In this case, you use the `add_register_value` command with the `-Width` switch and omit the value while in setup mode. Once you have exited setup mode, you can use the [set_register_value](#) command to set the variable:

```
set_register_value value_name value_string [-RADix {BInary | HEx | OCtal |  
Decimal}]
```

The name of the register value and its width are known prior to parsing the procedure file and running DRCs. The value is all X bits for DRC.

This command can only be used for a register value that was defined without a value string, and the width of the value string must match the width specified in the [add_register_value](#) command.

If the value overflows the width specified, an error is issued.

By default, the bits extracted by force/measure “#” in the procedure are from MSB to LSB. If the value specified should be shifted in/out in the opposite order, with the LSB bits applied/measured first, use the “-LSB_shifted_first” optional switch.

Deletion of a Register Value Variable

The [delete_register_value](#) command deletes all or a specified register value variable. You can only use this command in setup mode.

Register Value Variable Reports

The [report_register_value](#) command provides a detailed report of the register value variables to stdout or, optionally, a file.

Serial Load and Unload DRC Rules

The P13 and P54, P66, and W5 DRC rules are applied to the procedures and syntax in the section “Procedure Examples.”

The P1 “syntax error” message is used to catch many potential issues, such as specifying a “#times” value for applying a `load_unload_registers` procedure instead of the `shift_data_assignment`.

P13 and P54	616
P66	616
W5	617
Procedure Examples	618

P13 and P54

The P13 and P54 rules are used to check the shift assignment statements when calling a `load_unload_registers` procedure.

If the shift assignment uses a value string that is not properly formatted or contains illegal characters for that radix, then a P13 is issued.

```
Error: Invalid state value state string. (P13)
```

If the shift assignment references an undefined register value variable name, a P54 is issued.

```
Error: Undefined identifier identifier_string referenced by signal_name.  
(P54)
```

P66

The P66 rule is used to check for missing statements within a `load_unload_registers` procedure.

For example, if no Shift block is specified, or if an event statement using the “#” character is not present for each `shift_assignment` passed to the `load_unload_registers` procedure, then a P66 is issued.

```
Error: Procedure procedure_name is missing required statement_string  
statement. (P66)
```

The following examples illustrate the types of P66 DRC errors you could encounter.

Example 1

In the following example, the tool issues this error if there is no shift block within the `load_unload_registers` procedure:

```
Error: Procedure procedure_name is missing required shift statement.  
(P66)
```

Example 2

In the following example, the tool issues this error if there are no events in the `load_unload_register` procedure that use the “#” substitute character.

```
Error: Procedure procedure_name is missing required substitute event  
statement. (P66)
```

Example 3

In the following example, the tool issues this error if an apply statement that uses a `load_unload_registers` procedure has a shift data assignment that uses a signal that does not appear in the `load_unload_registers` procedure with the substitute character “#”.

```
Error: Procedure procedure_name is missing event using shift assign  
signal_name statement. (P66)
```

W5

The W5 DRC error is used to flag any extra events or statements in a `load_unload_registers` procedure, or any events that are not legal.

For example, if an event type other than Force or Expect is used with the “#” substitute character, then a W05 rule is issued. If the `load_unload_registers` procedure contains more than one shift block, this rule is issued. If there is a Force or Expect statement using a “#” character for a signal name that is not being passed to the procedure as a shift assignment, this rule is issued. The W05 rule is used when shift assignments for a particular Apply statement do not all have the same length.

```
Error: Procedure procedure_name has an illegal event statement or event  
order (event_statement_string) (W5)
```

Example 1

The following error is issued if an event in the `load_unload_register` procedure uses the “#” substitute character, but no shift data for this signal is passed into the procedure when it is called “extra shift block”:

```
<force | measure> signal_name # without matching shift assign data
```

Example 2

The following error is issued if there is more than one shift block in the `load_unload_registers` procedure.

```
"extra shift block"
```

Example 3

The following error is issued if more than one event of the same type in the shift block uses the same signal name with the “#” substitute character.

```
"too many events using shift assign signal_name"
```

Example 4

The following error is issued for an apply statement that uses a `load_unload_registers` procedure but has no shift data assignments in the apply statement.

```
"apply with no shift data assignment"
```

Example 5

The following error is issued for an apply statement that uses a `load_unload_registers` procedure and has more than one shift assignment, however, the target of the shift assignments are aliases and they are not the same width.

```
"unmatched shift assign signal width"
```

Example 6

This is issued for an apply statement that uses a `load_unload_registers` procedure and has more than one shift assignment and the assignments have different shift lengths.

```
"unmatched shift lengths"
```

Procedure Examples

The definition of the `load_unload_registers` procedure would look as follows. Notice how this procedure uses the “`shift =`” statement and the “`force tdi #`” statement to denote the shifting and where the string of data is applied, one bit at a time.

```

procedure load_unload_registers load_prpg1 =
  timeplate tp1 ;
  cycle =
    force prpg_select 1 ;
    ... // setup tap controller to load prpg
  end;
  shift =
    cycle =
      force tdi # ;
      pulse tck ;
    end;
  end;
end;

```

This procedure could then be applied in the test_setup procedure to initialize the PRPG, specifying the string of bits to apply to “tdi” during shifting.

```

procedure test_setup =
  timeplate tp1 ;
  cycle =
    force clk1 0 ;
    force clk2 1 ;
    force tck 0 ;
    ...
  end;
  apply load_prpg1 tdi = 00000000000000000000000000000001 ;
  cycle =
    ...
  end;
end;

```

For loading a register with the shift length during test_setup, using the values computed by the tool, the procedure definition would look the same, but how the procedure is called is slightly different. The following example both loads and unloads the register, as this same procedure could be used in a test_end procedure to unload the shift length value. The dofile for this example contains the following command:

add_register_value length_val -shift_length -width 16

The procedure file would contain the following:

```
procedure load_unload_registers load_unload_length =
  timeplate tp1 ;
  cycle =
    force clk1 0 ;
    ...
  end;
  shift =
    cycle =
      force tdi # ;
      expect tdo # ;
      pulse tck ;
    end;
  end;
end;
procedure test_setup =
  timeplate tp1 ;
  cycle =
    ...
  end;
  apply load_unload_length tdi = length_val, tdo = XXXXXXXXXXXXXXXXXXXX;
  cycle =
    ...
  end;
end;
```

This next example uses an alias to group three tdi signals into one alias, and also adds a post shift cycle to the load_unload_registers procedure. When this procedure is called, the data passed to it is consumed three bits at a time, with each bit being shifted into tdi1, tdi2, and tdi3 in order. The total number of shifts applied in the “shift” block is the total length of the value string divided by three, and then minus one for the post shift. Each shift consumes three bits, and the final cycle adds a post shift which assigns the last three bits. The length of the value string being passed to this procedure must be a multiple of three.

```
alias TDI_GRP = tdi1, tdi2, tdi3 ;
procedure load_unload_registers load_reg1 =
  timeplate tp1 ;
  cycle =
    force clk1 0 ;
    ...
  end;
  shift =
    cycle =
      force TDI_GRP # ;
      pulse tck ;
    end;
  end;
  cycle =
    force TDI_GRP # ;
    pulse tck;
  end;
end;
```

This final example shows how to use a dofile commands to set up a user-defined register value that is used to store total number of scan patterns when the final patterns are saved.

```
add_register_value scan_pat_count -pattern_count scan_test -width 24
```

Notes About Using the stil2mgc Tool

You can use the stil2mgc tool to create a dofile and procedure file. The stil2mgc tool reads a STIL Procedure File (SPF) and then creates a dofile and test procedure file.

The dofile defines clocks, scan chains, scan groups, and pin constraints. The test procedure file contains a timeplate and the following standard scan procedures: test_setup, load_unload, and shift. For more information about this tool, refer to the [stil2mgc](#) command description in the *Tessent Shell Reference Manual*.

Extraction of Strobe Timing Information from STIL (SPF)..... 622

The STIL ClockStructures Block..... 622

Extraction of Strobe Timing Information from STIL (SPF)

In the STIL WaveformTable, strobe window and edge strobe are indicated by which event characters are used for the measure timing. Using the ‘H’ and ‘L’ characters indicates that the measure is an edge strobe, while using the ‘h’ and ‘l’ character indicates a window strobe with the length of the strobe window being determined by the following X event in the event list for that WaveformCharacter.

If the window strobe events (‘h’ or ‘l’) are used for a measure event, then the strobe window is calculated taking into account the strobe window indicated by these events. The shortest strobe window calculated is the one used for the procedure file.

If “-edge_strobe_processing on” is specified to the tool, and the edge strobe events (‘H’ or ‘L’) are used for a measure event, then the strobe window is set to 0 in the procedure file to indicate edge strobe timing.

If the “-edge_strobe_processing” option is not used or is set to off, and edge strobe events (‘H’ and ‘L’) are used for a measure event, then the existing behavior is maintained where the strobe window is calculated based on the next force event or the period of the Waveform table. The strobe window is not set to 0.

The STIL ClockStructures Block

When parsing STIL Procedure Files (SPF), stil2mgc recognizes the Synopsys-defined ClockStructures block and uses this information to create clock_control definitions. The Latency statement within the ClockStructures block is used to control the “set_external_capture_options” statement in the generated dofile.

Test Procedure File Commands and Output Formats

The test procedure file is supported by a set of commands and a set of output formats.

Test Procedure File Tool Commands

The following table provides a summary of the tool commands that support the use of test procedure files. For a detailed description of each command, refer to the corresponding command reference page in the *Tessent Shell Reference Manual*.

Table 10-2. Procedure File Tool Command Summary

Command	Description
add_scan_groups	Adds a scan group using the scan procedures in the named procedure file.
read_procfile	Reads a new procedure file in non-setup mode. Merges new procedure and timing data with existing data loaded from previous procedure files.
write_procfile	Writes out existing procedure and timing data as the named procedure file.
write_patterns	Loads a cycle before saving patterns and merges the new data with the existing data.
report_procedures	Reports (displays) a named procedure to the screen. The -All switch displays all procedures to the screen.
report_timeplates	Reports (displays) a named timeplate to the screen. The -All switch displays all timeplates to the screen.

Test Procedure File Output Formats

The test procedure file format supports the following output formats:

- Fujitsu TDL
- Mitsubishi TDL
- STIL
- TI TDL
- WGL
- TSTL2
- VERILOG

The `-PROfile` switch causes the `write_patterns` command to get its timing information from the procedure file. For more information, refer to the [write_patterns](#) command in the *Tessent Shell Reference Manual*.

Chapter 11

Tessent Visualizer

Tessent Visualizer is a graphical user interface for Tessent Shell. This chapter describes the various features of Tessent Visualizer and how to work with those features. With Tessent Visualizer, you can use schematics, browsers, tables, and other reports to analyze and understand DFT issues. By cross-referencing between these different viewpoints of your data and performing various analyses on them, you can more quickly arrive at solutions to those issues.

Invoking Tessent Visualizer	626
Framework Overview	627
Tessent Visualizer Components and Preferences	630
Tables	631
Schematics	639
Tessent Visualizer Preferences	671
Macros	672
Tooltips	673
Gate Report Settings	674
Saving and Restoring the Session State	675
Window Title Prefixes	675
Tessent Visualizer GUI Reference	677
Hierarchical Schematic	677
Flat Schematic	682
Instance Browser	684
Wave Generator	686
Cell Library Browser	688
DRC Browser	689
Pin Data	691
Transcript	692
Text/HDL Viewer	693
Diagnosis Report Viewer	695
Search Features	697
Using Tessent Visualizer to Debug Design Issues	699
Accessing Tutorials for Tessent Visualizer	703
Accessing Videos for Tessent Visualizer	704

Invoking Tessent Visualizer

You can invoke Tessent Visualizer in different ways, depending on the task you want to perform. Tessent Visualizer can be launched directly from the Tessent Shell application or remotely from another machine or a different terminal on the same machine.

Prerequisites

- Comply with the hardware and operating systems requirements listed under “[Supported Hardware and Operating Systems](#)” in the *Managing Tessent Software* manual.


The requirements also apply to the machine providing the X Window desktop (as defined by the DISPLAY environment variable).

- Set the DISPLAY environment variable correctly.

Procedure

1. Set a context using the `set_context` command.
2. Load a design using the `read_verilog` command.
3. Read a library using the `read_cell_library` command.
4. Set the current design using the `set_current_design` command.

Note

 You can invoke Tessent Visualizer without issuing these commands, but most of the user interface is disabled until you load a design and any required libraries, and you specify the current design. You can also use these commands in the **Transcript** tab after you invoke Tessent Visualizer.

5. Invoke Tessent Visualizer using one of the following methods:

- Invoke explicitly on the same machine Tessent Shell is running on:

```
ANALYSIS> open_visualizer
// Note: Tessent Visualizer client successfully started and
connected to the server.
```

- Invoke explicitly on a different machine.

Tessent Visualizer is a client-server application. Tessent Shell is the server, and Tessent Visualizer is a client of that server.

To start the server in Tessent Shell, invoke the `open_visualizer` command with the `-server_only` switch. Optionally, specify the TCP/IP port using the `-tcp_port` switch. Valid port numbers are from 1024 to 65535.

```
ANALYSIS> open_visualizer -server_only
// server: started
// - hostname:          server01.example.com
// - TCP/IP port:      43527
// - authorization code: 821703
// client: not connected
// Note: To connect Tessent Visualizer client to the server,
// issue the following command in a Linux console:
// -----
// /<path>/tessent -visualizer -server server01.example.com \
// -tcp_port 43527 -authorization_code 821703
// -----
```

To start the client, use the **tessent -visualizer** command:

```
$TESSENT_HOME/bin/tessent -visualizer
```

This opens a remote connection dialog box that prompts for the hostname, port, and authorization code.

The six-digit authorization code is a one-time password that is cleared upon successful connection to a server or after five invalid attempts. To reconnect to the same server, generate a new authorization code by invoking `open_visualizer` on that server with the `-authorization_code` switch.

For more information, see the “[tessent](#)” command description in the *Tessent Shell Reference Manual*.

Note



The IP address and port of the server must be accessible from the client machine.

- Invoke implicitly using one of the following commands:
 - [add_schematic_callout](#)
 - [add_schematic_objects](#)
 - [add_schematic_path](#)
 - [add_schematic_connections](#)
 - [analyze_drc_violation](#)
 - [display_diagnosis_report](#)

Framework Overview

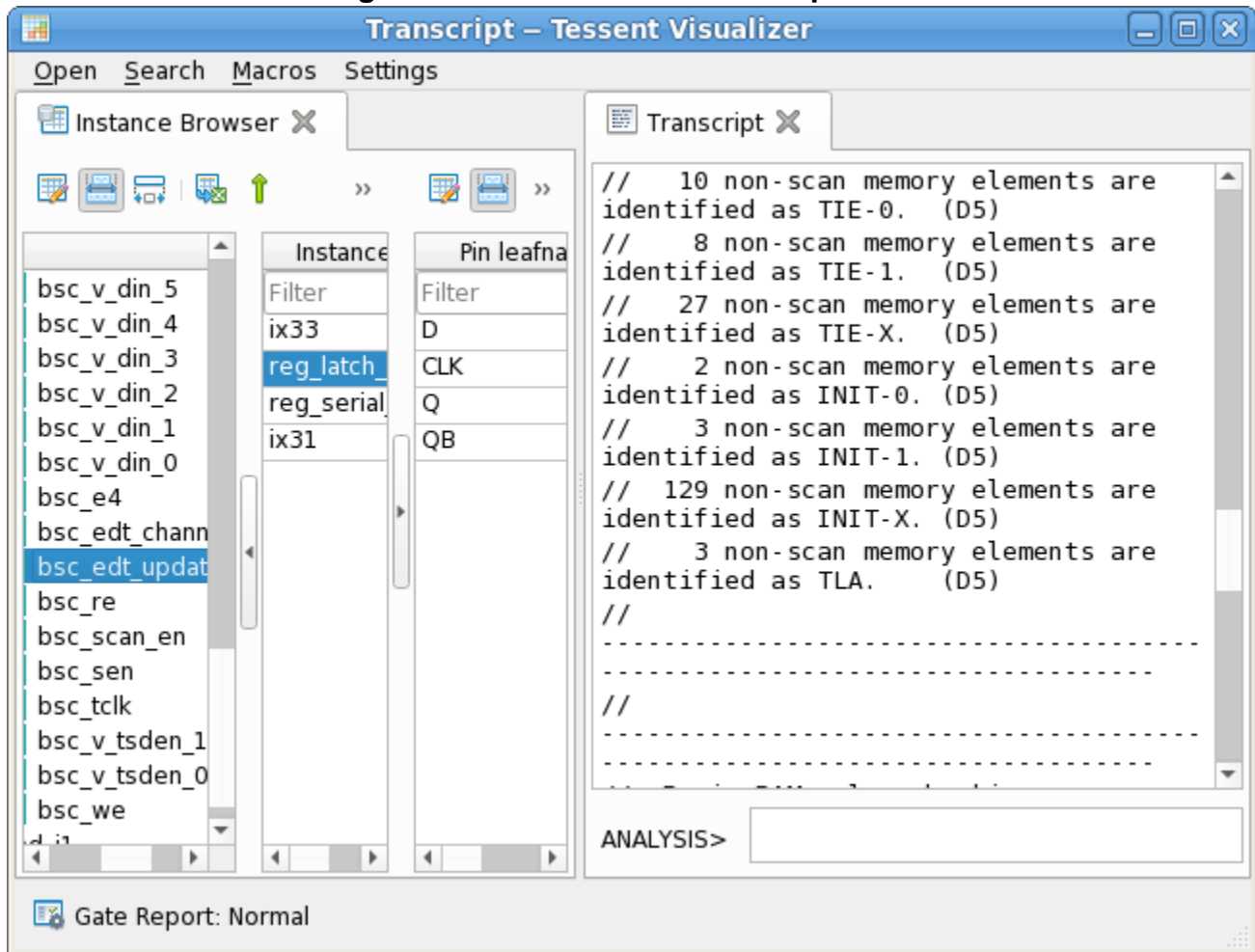
Tessent Visualizer provides multiple tabs for viewing and debugging design and simulation data in specialized windows.

The Tessent Visualizer framework consists of the following:

- **Windows** — Windows contain one or two panes, a global top menu, and a status bar. The menu and status bar are identical across multiple windows; only the pane content differs.
- **Panes** — There are at most two panes per window. By default, one pane provides multiple tabs. Windows can be split into two panes by dragging and dropping. Each pane has its own tabs.
- **Tabs** — You can view and select multiple tabs within a pane. Each functional task in Tessent Visualizer is placed as a tab. You can reorder the tabs by dragging and dropping, or cycle through them with Ctrl+Tab (next tab) or Ctrl+Shift+Tab (previous tab). See [“Tessent Visualizer Keyboard Shortcuts”](#) on page 791 for more ways to interact with Tessent Visualizer using the keyboard.

Each window can contain two panes (separate sets of tabs). This is called a split view. To create a split, drag one of the tabs to either side. Drag and drop a tab to move it from one side of the split to the other. There are at most two work areas (panes) per window, as shown in [Figure 11-1](#).

Figure 11-1. Tessent Visualizer Split View



You can undock any tab from a window as a separate (floating) window and redock as necessary. Floating windows have the original window's full functionality, with access to features such as global settings and the capability to offer a split view. To undock a tab, double-click the tab or drag and drop it away from the parent window. To redock, drag it back to the parent window (or any other Tessent Visualizer window) and drop it as a tab in that window.

Tessent Visualizer Components and Preferences

Tables and schematics represent different views of pins, instances, nets, and other design objects, as well as actions associated with those design objects. For example, a hierarchical pin shown on a hierarchical schematic, the instance browser, or a search window can also be shown on a flat schematic.

Tables	631
Schematics	639
Tessent Visualizer Preferences	671
Macros	672
Tooltips	673
Gate Report Settings	674
Saving and Restoring the Session State	675
Window Title Prefixes	675

Tables

Tessent Visualizer presents data such as tracing results, pin listings, instances, and nets in tabular form in all windows, including schematics. These tables are configurable, and data can be filtered dynamically.

Table Toolbar Features	632
Columns and Filters Editor	634
Table Filters	635
Data Sorting	637
Row Highlighting	637

Table Toolbar Features

All tables used in Tessent Visualizer have several common elements, including toolbars, dynamic headers, filter fields, and data rows and cells.

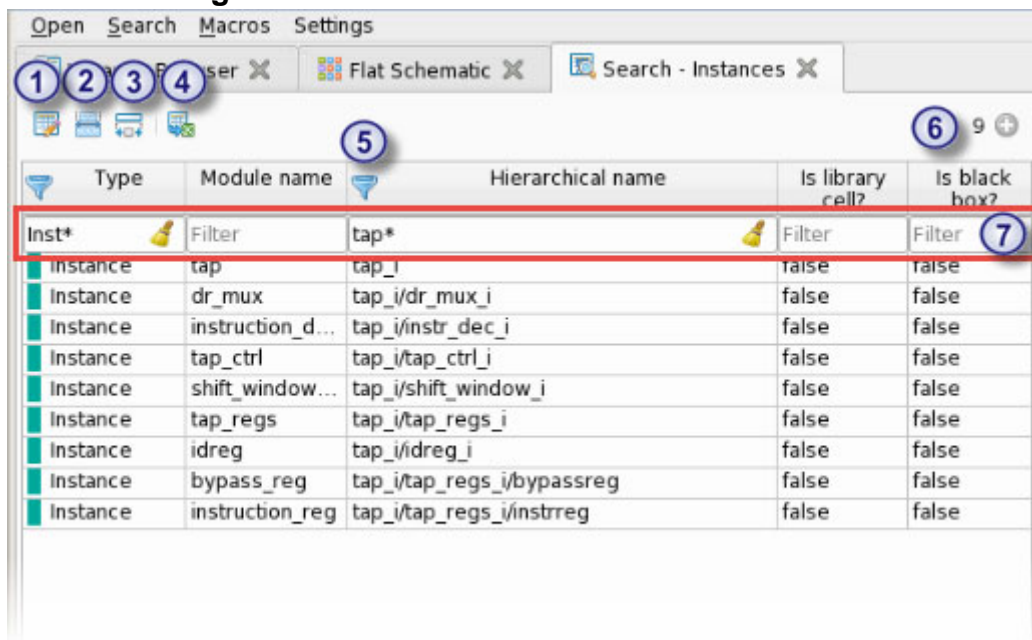
Description

Most report tables in Tessent Visualizer use a spreadsheet-like format with customizable columns, a toolbar, and filter fields for each column. The following are additional features common to tables:

- Customize the data types included in the table with the Columns and Filters Editor.
- Select one or more table rows in the Tessent Visualizer window and perform various compatible actions using the popup menus available by right-clicking.
- Export table data in CSV format with the Export Table icon.
- Add line numbers to tables from the Preferences dialog box.

Figure 11-2 illustrates the locations of these items.

Figure 11-2. Common Table Toolbar Features



Objects

Object	Description
1	Columns and Filters Editor. Controls the form and content of the table.

Object	Description
②	Automatic/Manual Column Width. Toggles between automatic and manual mode for controlling column widths and column header wrap. In automatic mode, column widths stretch to fit the table and column headers wrap if necessary to accommodate narrow data.
③	Resize columns to contents. This fits the column widths to accommodate the size of the displayed data.
④	Export table in CSV format. Alternately, use Ctrl+S.
⑤	Funnel. If present, this indicates that filtering is active. This symbol is not displayed unless a filter is applied.
⑥	Loaded row counter. By default, the table loads up to 250 items. Change the default threshold from the Preferences dialog box. The “+” next to this number indicates that all data for the table has been loaded if it appears in a gray circle. A green circle indicates that additional data exists in Tessent Shell. Load this additional data by scrolling down or clicking the circle.
⑦	Filter fields. These control which data are displayed in the table.

Related Topics

[Columns and Filters Editor](#)

[Tessent Visualizer Preferences](#)

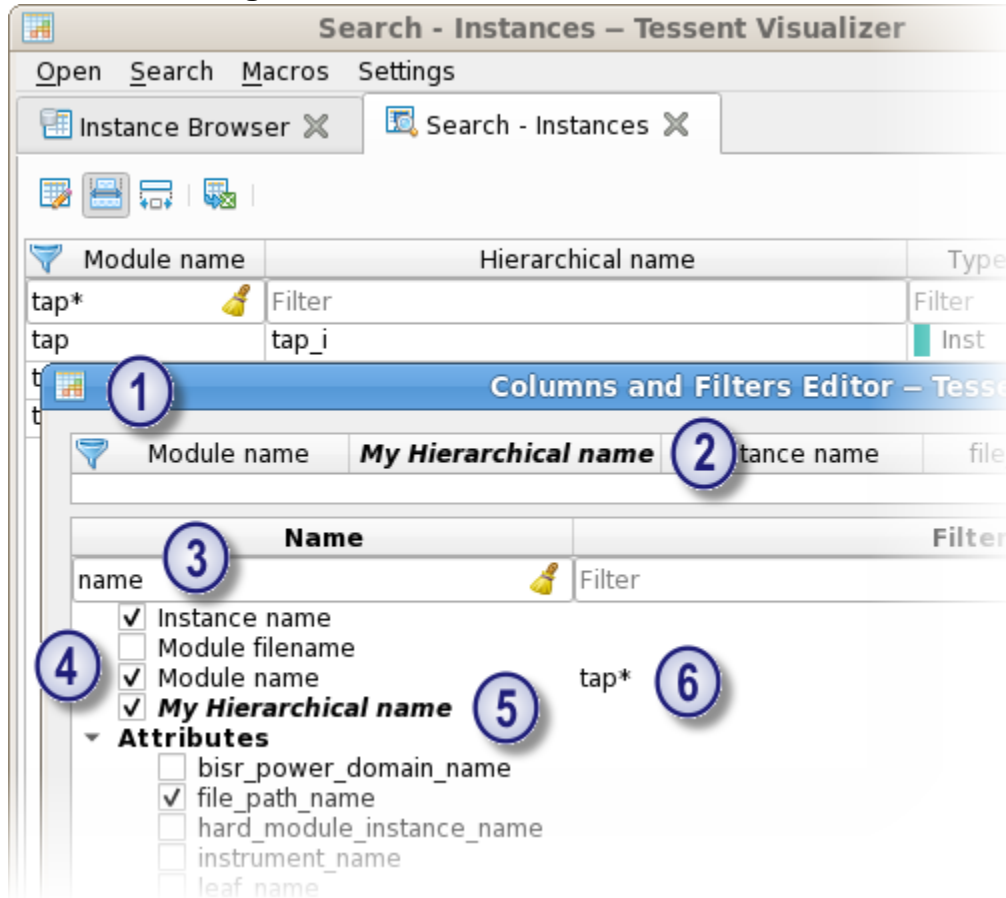
Columns and Filters Editor

To access: click the “Columns and Filters Editor” icon in the Tessent Visualizer toolbar.
 Use the Columns and Filters Editor to control the content and format of tables.

Description

Figure 11-3 shows features of the Columns and Filters Editor:

Figure 11-3. Columns and Filters Editor



Objects

Object	Description
①	Apply filters by typing terms in the filter fields, shown in Figure 11-3. When a column is being filtered, the funnel icon is displayed next to the column name in both the Column and Filters Editor and the tabbed window being modified by the Column and Filters Editor.
②	Reorder columns by dragging and dropping the header items at the top of the Columns and Filters Editor.

Object	Description
③	Search for specific data types by using string matching in the column for the property name. Search results are displayed as you type.
④	Add or remove table columns by checking or unchecking the boxes associated with column names.
⑤	Customize column label names by double-clicking the name representing the column. The name change is indicated by a bold italic font in the column and filters editor as well as the results table.
⑥	Add or edit column filters by double-clicking the filter cell.

Usage Notes

The report table content is automatically refreshed when you accept the modifications by clicking **OK**.

Table Filters


Filters control the data displayed in tables by limiting that display with search criteria you specify.

The filter field is located directly beneath the column headers, as shown in [Figure 11-2](#) on page 632. Tessent Visualizer uses the following filters:

- **Exact match** — A string literal or value, optionally prepended with an equals sign (=). This restricts the display of data to those objects matching the string or value. String literals containing spaces and special characters should be enclosed by single quotation marks; for example, 'x = y'.
- **Exact inverse match** — A string literal or value prepended with the not equals sign (!=). This restricts the display of data to those objects not matching the string or value.
- **Logical operators** — AND, OR, and NOT. For example:
 - != abc AND != xyz restricts the display of data to all objects that are not named abc and are not named xyz.
 - abc OR xyz restricts the display of data to those objects named abc or xyz.
 - NOT is a special logical modifier and requires an explicit REGEXP or GLOB.
- **Wildcard matching** — The keyword GLOB with a single character symbol (?) or an asterisk (*) that represents zero or more characters. For example:
 - GLOB '*abc' restricts the display of data to all objects with names ending in the string abc.

- GLOB '*abc*' restricts the display of data to all objects with names that contain the string abc.
- GLOB 'abc?xyz' restricts the display of data to all objects with seven-character names that begin with the string abc, followed by a single character, and end with the string xyz.
- GLOB '?*' matches any non-empty string.

Note

 In GLOB filters, if there is no empty space as part of the expression, the keyword GLOB and the single quotation marks are optional. For example: *abc*

- **Numerical comparison operators** — Data that includes numerical information can be filtered using standard numerical comparison operators (<, <=, >, >=).
- **Standard regular expressions** — Data can be filtered using regular expressions by using the construct REGEXP 'RE'. For example, REGEXP '.*clk_\d+\$' restricts the display of data to all objects with names ending in the string clk_ followed by one or more decimal digits.

Note


 In this filter, the keyword “REGEXP” and the single quotation marks are required.


Table 11-1. Filtering Summary

Filter Type	Filter Format	Examples
Exact match	<i>value</i> = <i>value</i>	abc =xyz =12
Exact match (inverse)	! <i>value</i>	!=abc
Numerical operators	< <= > >=	>10 >20 AND <= 50
Wildcards	GLOB ' <i>expression</i> '	GLOB 'abc*' GLOB '*xyz*' GLOB 'abc?xyz' GLOB 'abcdef??'
Regular expression	REGEXP ' <i>expression</i> '	REGEXP '^abc.*' REGEXP '.*_\d0\$'

Table 11-1. Filtering Summary (cont.)

Filter Type	Filter Format	Examples
Logical operators	AND OR NOT GLOB ' <i>expression</i> ' NOT REGEXP ' <i>expression</i> '	abc OR xyz !=abc AND != xyz NOT GLOB 'abc*' NOT REGEXP '^abc.*'

Note

 Filters applied to table columns are processed in Tessent Shell on the complete data model, no matter how many resulting rows appear in the Tessent Visualizer tabular reports.


Data Sorting

The default order of rows in tabular data is determined by the underlying data structure in Tessent Shell. In some tables, you can sort this data by clicking the column title cell. The ordering cycles through each of ascending, descending, and the default ordering.

Data sorting is enabled when the number of rows loaded is less than the limit (250 by default, but configurable in the **Preferences** menu). Exceptions: sorting is always enabled in the following cases:

- A table with child instances in the **Instance Browser**.
- A table with modules in the **Cell Library Browser**.

Tip

 For better performance, limit the set of visible data with filtering before using the sorting function.

Related Topics

[Tessent Visualizer Preferences](#)

Row Highlighting

Tables in Tessent Visualizer use color coding to indicate the status of rows.

Figure 11-4. Selection Colors

Type ▲	Instance name	Module name	Displayed
Filter	Filter	Filter	Filter
Inst	tap_i	tap	false
Inst	core_i	test_design_edt...	false
Inst	bsr_i1	bsr_instance_1	true
Inst	pad_i1	pad_instance_1	false

- **Dark blue** — Selected objects
- **Light blue** — Active cell
- **Gray** — Objects currently displayed in the schematic

Most popup menus operate on the selected object (dark blue). Copy and paste actions (Ctrl+C and Ctrl+V) work only on the active cell (light blue). The gray highlighting shows which objects are currently displayed in a schematic.

Use Shift+click to select multiple adjacent objects in a table and Ctrl+click to select multiple nonadjacent objects.

You can also use a Boolean value in the “Displayed” column for easy filtering of objects displayed in a schematic.

Schematics

Tessent Visualizer schematics contain features that enable you to inspect and navigate your design.

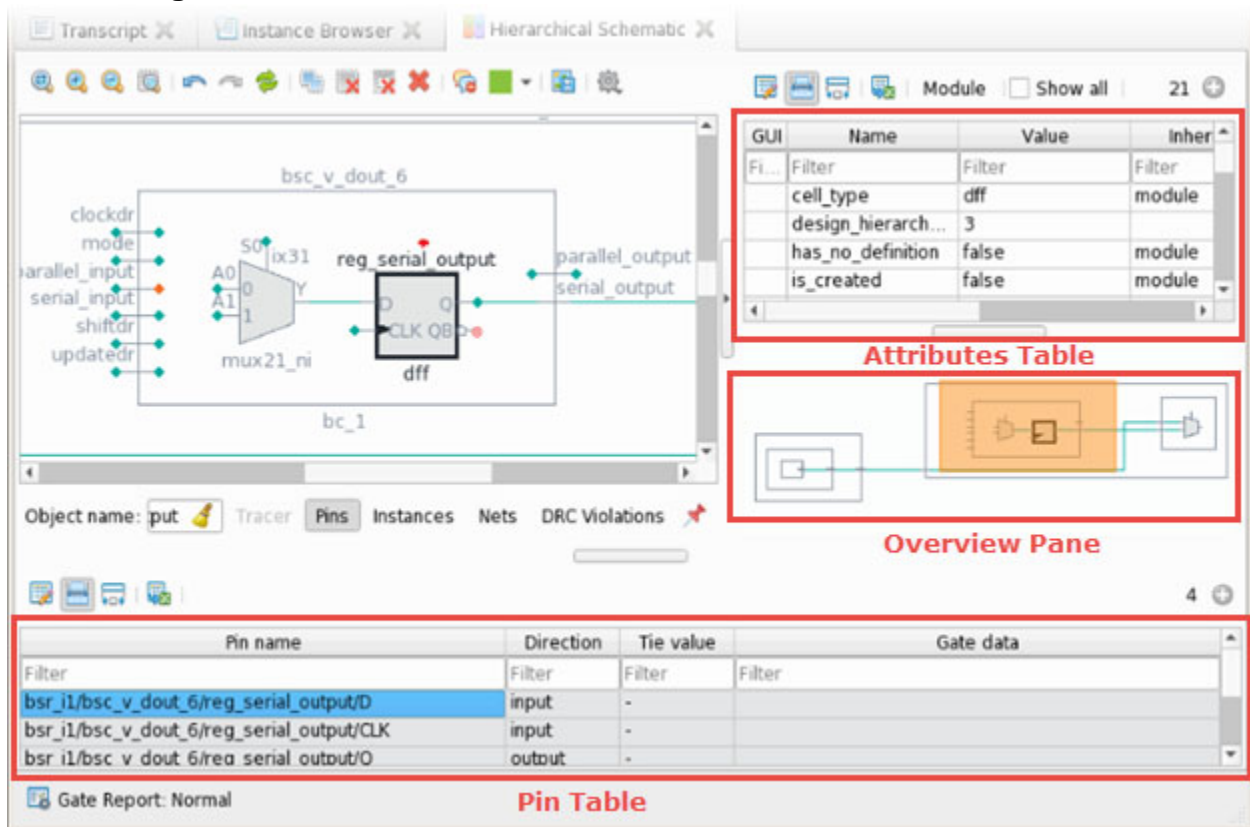
Figure 11-6 in “[Toolbar](#)” on page 640 shows a Flat Schematic with the following GUI objects highlighted:

- **Toolbar** — Controls how objects are displayed in the schematic and export schematic objects for use by other tools.
- **Address Bar** — Displays the name of the currently selected object. You can also use the address bar to add new objects to the schematic by name.
- **Selection Buttons for Context Table** — Select which context table you want to display.

Figure 11-5 illustrates some additional schematic common features:

- **Attributes Table for the Selected Object** — Displays information about the currently selected object. The specific types of data in this table depend on the object selected.
- **Overview Pane** — Shows all objects that have been added to the schematic. The currently zoomed view is highlighted. Drag and drop the highlight rectangle to pan the view.
- **Pin Table** — Displays information about the pins on the currently selected object.

Figure 11-5. Schematic With Overview Pane and Attributes Table



Toolbar	640
Schematic Symbols	643
Context Tables	655
Signal Net Tracing Strategies	665
Displayed Property	668
Tessent Shell Attributes	668
Mouse Gestures	670

Toolbar

There is a toolbar at the top of each Tessent Visualizer schematic.

Figure 11-6 shows the location of the schematic toolbar.

Figure 11-6. Schematic Elements: Toolbar, Address Bar, and Selection Buttons

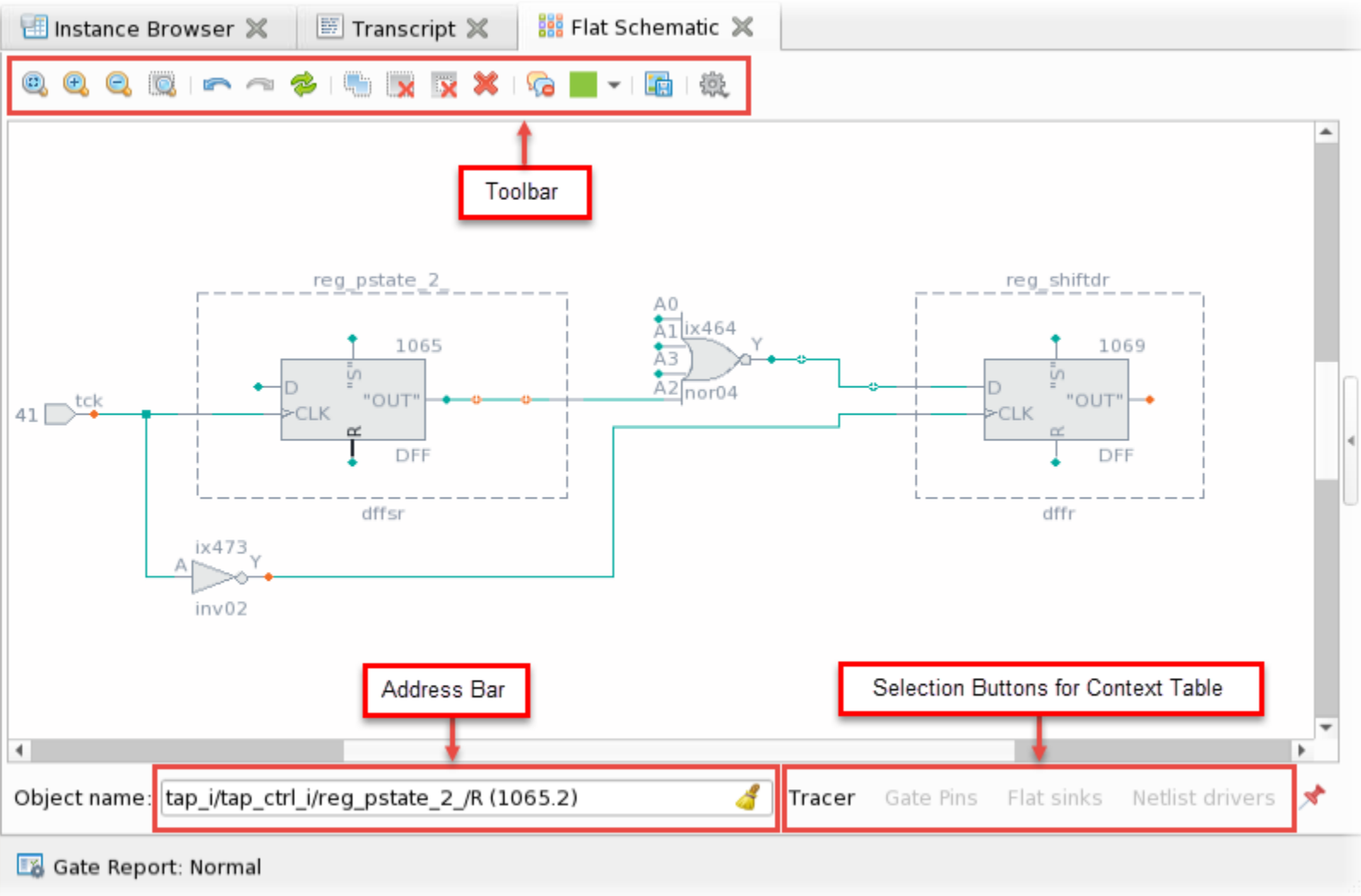


Table 11-2 summarizes the contents of the toolbar.

Table 11-2. Schematic Toolbar Actions

















Tool	Description
	Zoom all: fit the current view of the schematic in the window.
	Zoom in.
	Zoom out.
	Zoom selected: zoom in on and center the selected object or objects.
	Undo: undo the previous action.

Table 11-2. Schematic Toolbar Actions (cont.)

Tool	Description
	Redo: redo the previous action.
	Redraw: refresh the schematic view.
	Select all: select all objects in the schematic view.
	Delete selected: delete all selected objects from the schematic view.
	<p>Delete unselected: delete all objects from the schematic view except those currently selected. There are two exceptions:</p> <ul style="list-style-type: none"> • If a selected object is an instance that has some pins displayed, those pins are not deleted. • If the selected objects are pins that are connected with a net, that net connection is not deleted.
	Delete all: delete all objects from the schematic view.
	Collapse callouts: collapse all callout messages to their icon representations, or expand if currently collapsed.
	Mark/Unmark: highlight the currently selected object or objects with the selected color, or clear the highlighting on the selected object. If the color you want to use is already active, you can use Ctrl+M to highlight the selected object(s).
	Export schematic: save the current view of the schematic in PDF or PostScript format, or as a schematic dofile. You can run the generated schematic dofile (or any other dofile) using the Execute dofile option under the Open menu, or by using the dofile command in the Transcript or shell window.
	Options: set schematic-specific options (different schematic types have different options in this submenu).

Note

 When a dofile generated with the **Export schematic** action is executed in a future session, the displayed schematic is re-created as closely as possible. In rare circumstances, some objects may not be displayed or connected exactly as in the original schematic.

Schematic Symbols

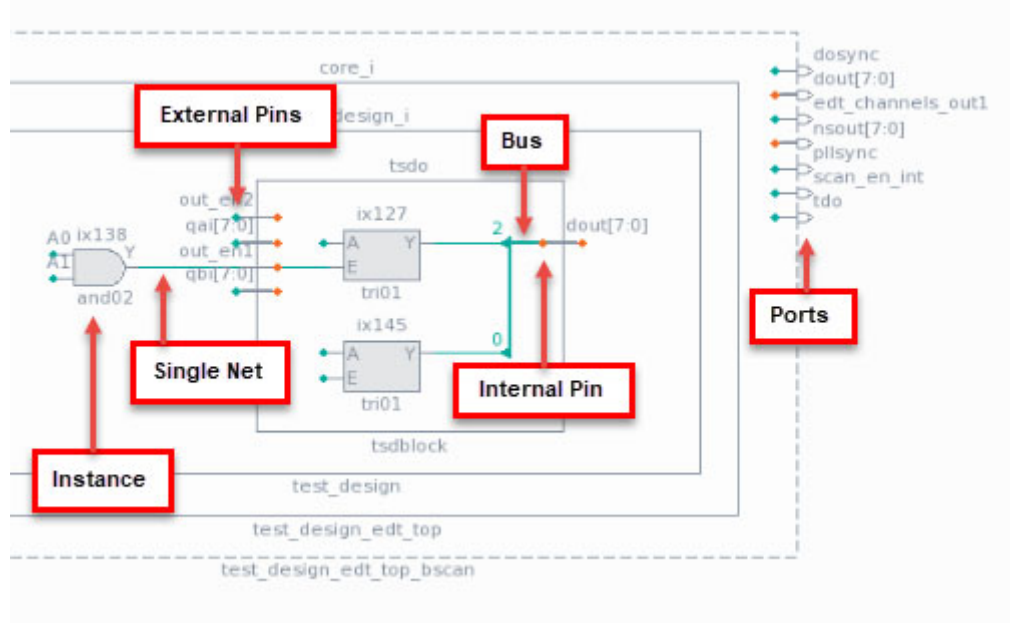
Tessent Visualizer schematics use a variety of conventional symbols to present the structure, objects, and connections of a design.

Object Types in Schematics	643
Markers	649
Buffer and Inverter Collapsing	651
Folding and Unfolding Instances	654

Object Types in Schematics

The object types displayed in schematics are instances, pins, ports, and nets. You interact with these objects to explore your design, obtain information about the objects' properties, trace nets forward and backward, and so on.

Figure 11-7. Hierarchical Schematic Showing Pins, Ports, Instances, and Nets



Instances

Instances are individual copies of library cells, primitives, or groups of cells connected by nets to form hierarchical instances. Both the **Hierarchical Schematic** and **Flat Schematic** tabs can contain instances of library cells and primitives, which display as conventional logic symbols (such as NAND, NOR, and MUX). Library cells display as rectangles in the **Hierarchical Schematic** tab, but you can see their functional representation in the **Flat Schematic** tab.

Note

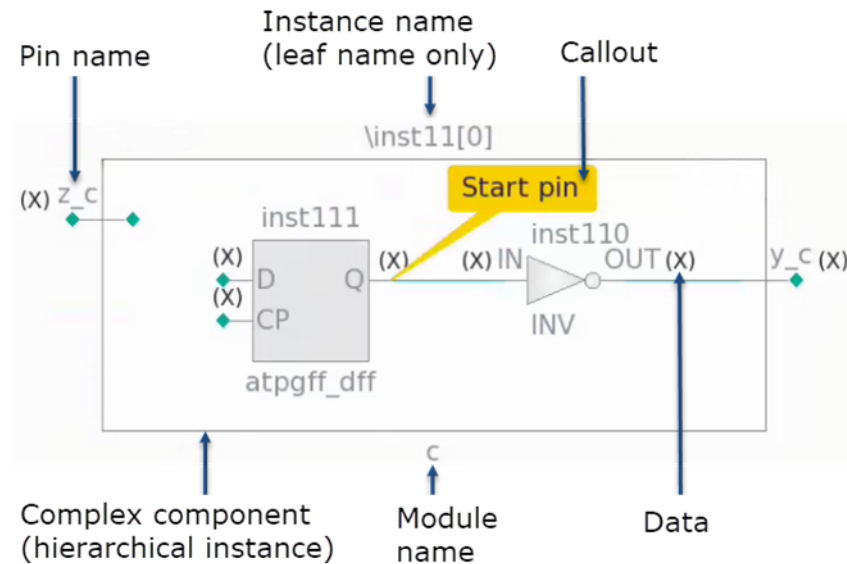
The ordering of pins on displayed instances matches the ordering as provided by report_gates.

Hierarchical Instances

Tessent Visualizer schematics show groupings of symbols by enclosing them in rectangles, objects called hierarchical instances. Hierarchical instances include their pins. All instances except those at the leaf level display with both the internal and external connections to those pins.

Figure 11-8 depicts an instance in the **Hierarchical Schematic** tab. Callouts convey information about specific design objects in certain contexts, and the schematic shows simulation data on design pins where appropriate.

Figure 11-8. Hierarchical Instance With Callout and Pin Data



Use the right mouse button as shown in Figure 11-9 to show the internal connectivity of a selected hierarchical instance. If the number of objects at the selected instance's interface is large, this option is not available. In this case, use context tables to selectively add instances and connections to the schematic.

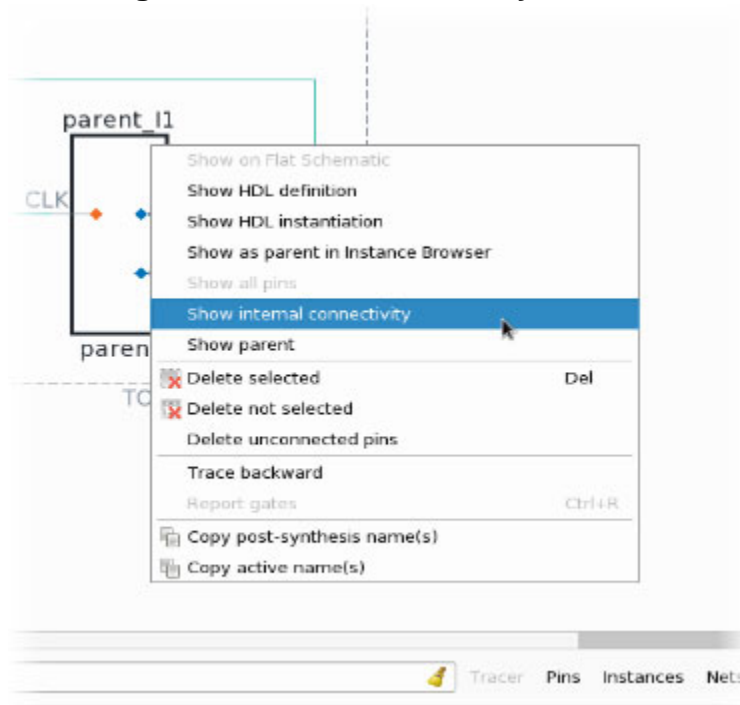
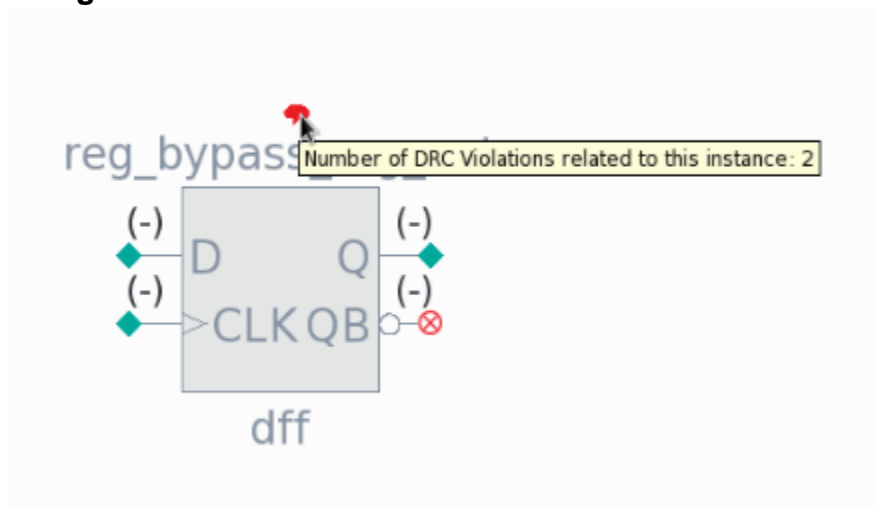
Figure 11-9. Showing the Internal Connectivity of a Hierarchical Instance

Figure 11-10 shows an instance in the **Hierarchical Schematic** tab with a red callout symbol. Hover the mouse pointer over the symbol to show the number of DRC violations, and click the symbol to display a table of those violations.

Figure 11-10. Hierarchical Instance With DRC Callout

Hierarchical cells (HCells, defined by ``celldefine` and ``endcelldefine` in Verilog) are depicted with a slightly lighter gray color than the library cells. You can view objects inside hierarchical cells, but these are not annotated with gate data from the flat model.

Figure 11-11. Hierarchical Cell

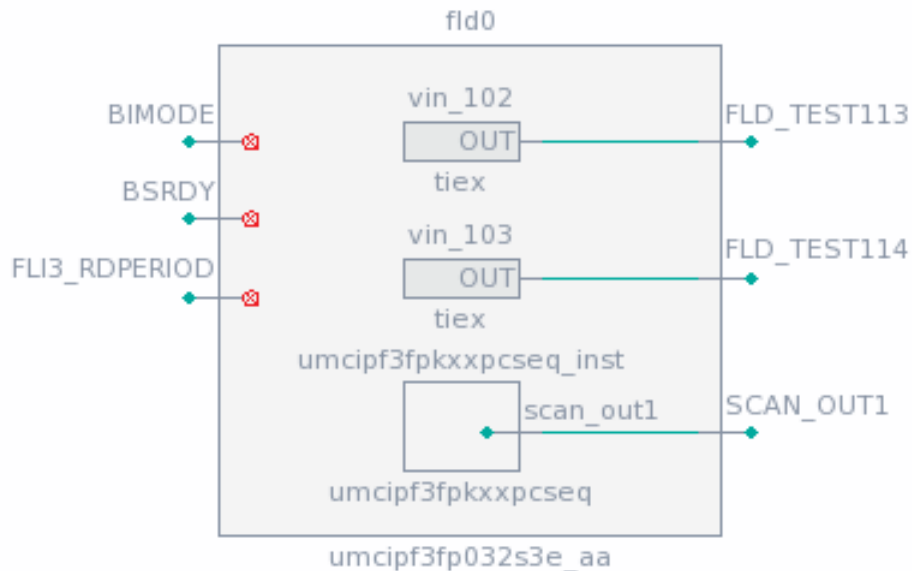


Figure 11-12. Hierarchical Instance Representing an Assign Statement



The code for the assign statement symbol is similar to this:

```
module RDS_process_rtl_tessent_buf_49(a, y);
  input a;
  output y;

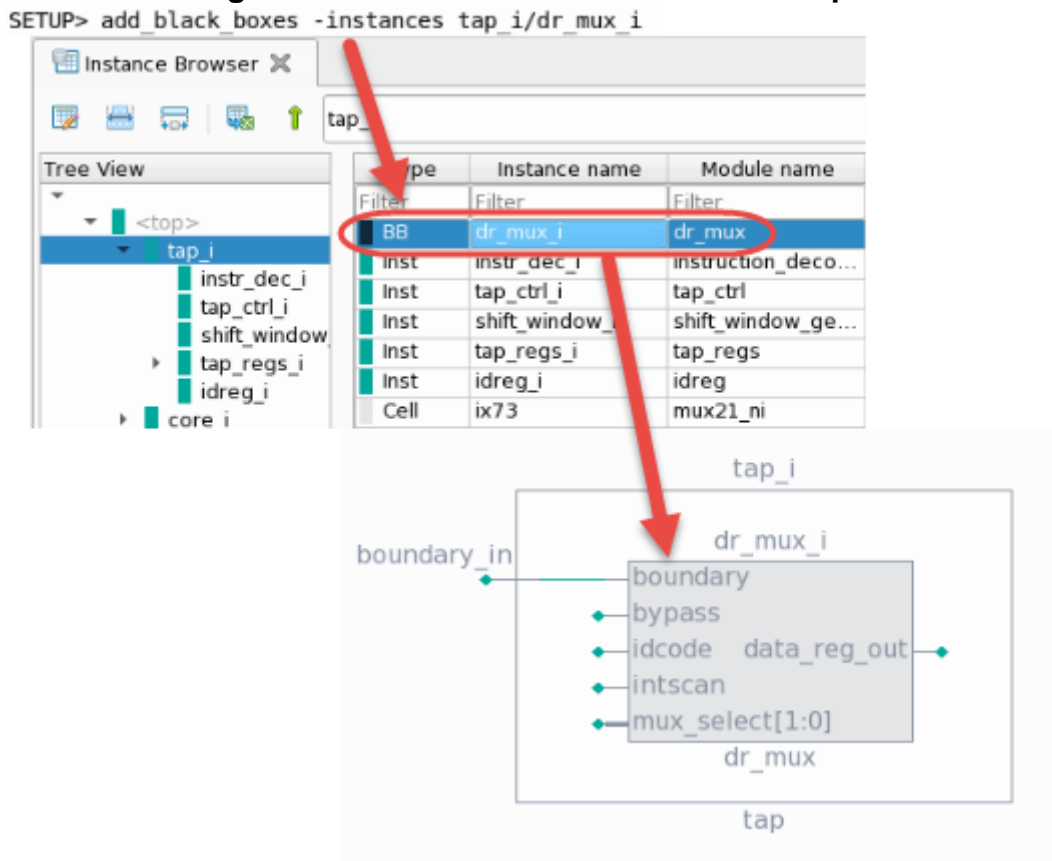
  assign y = a;
endmodule
```

Black-Boxed Instances

A black-boxed instance is an instance with no defined model that you create with the [add_black_boxes](#) command.

[Figure 11-13](#) shows how the `add_black_boxes` command creates a blackbox from the `dr_mux_i` instance within the `tap_i` module. The instance then becomes a BB type and displays as a solid gray rectangle in the **Hierarchical Schematic** tab.

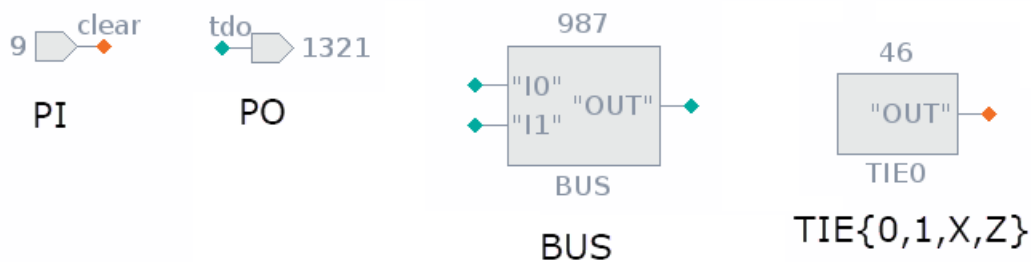
Figure 11-13. Black-Boxed Instance Example



Instances in the Flat Schematic Tab

Flattening the design creates certain artificial gates, as shown in [Figure 11-14](#).

Figure 11-14. Objects in the Flat Schematic Tab



The flattening process creates primary inputs and outputs, as well as bus instances necessary for modeling bidirectional pins and ports. In addition, there are instances that tie a net to a fixed value.

User-defined cells display as rectangles with a solid fill. User-added ports display with an orange fill.

Figure 11-15 shows a persistent buffer symbol in the **Flat Schematic** tab. The same buffer would look like Figure 11-12 in the **Hierarchical Schematic** tab because you typically implement persistent buffers using an “assign” statement.

Figure 11-15. Persistent Buffer Symbol



Pins and Ports

Pins and ports are electrical connections between an instance and a net. Both of these, along with buses (a collection of related pins) indicate signal sources or destinations in the schematic, and are characterized by their size and direction (input, output, or bidirectional). For bus naming conventions, see “[Buses](#)” on page 679.

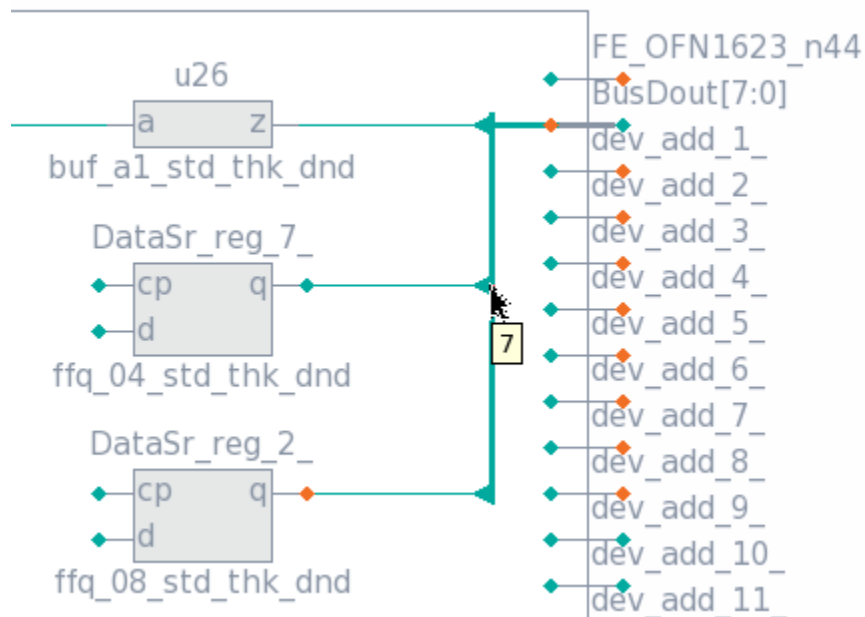
Callout Messages

Callouts are informational messages that display on a schematic, as shown in [Figure 11-8](#) on page 644. You can add callouts to any instance, pin, or net object that exists on the schematic using the `add_schematic_callout` command.

Callouts can also display in a collapsed format, as shown as “collapsed callouts” in [Table 11-3](#) on page 649. The table describes how to view the text of a collapsed callout. Use the “Collapse callouts” button as described in [Table 11-2](#) on page 641 to collapse all callouts. Use Ctrl+left mouse button to drag and drop individual callouts.

Nets

Single nets display as thin green lines, and bus nets display as thicker green lines. When a bus net branches to a single net, the bus index displays when the mouse pointer is hovered over the triangular rip point symbol, as shown in [Figure 11-16](#).

Figure 11-16. Bus Index Displayed at Rip Point

Markers

Tessent Visualizer uses marker symbols in schematics to indicate additional properties of the elements shown. Some, such as trace markers and buffer-collapsing markers, have actions associated with them.

Table 11-3 lists a summary of marker symbols used in Tessent Visualizer schematics, and whether they are displayed in flat or hierarchical schematics (or both).

Table 11-3. Marker Symbols

Symbol	Hier	Flat	Description
	✓	✓	Green diamond: pin with single connection available to display. Click to display unambiguous connection from this pin.
	✓	✓	Orange diamond: pin with multiple connections available to display. Click to open the Tracer table and select the tracing destination.
	✓		Blue diamond: ambiguous bidirectional pin. When all drivers and drains of the port are visible, the color changes to green or orange, as appropriate. Click to open the Tracer table and select the tracing direction and destination.
	✓		Half-filled diamond (green): single connection left on bus. Remainder of displayed pins tied or unconnected. Click to open the Pins context table to show the tie values on all bus pins.

Table 11-3. Marker Symbols (cont.)


















Symbol	Hier	Flat	Description
	✓		Half-filled diamond (blue): one or more ambiguous bidirectional pins on bus. Remainder of displayed pins tied or unconnected. When all drivers and drains of the port are visible, the color changes to green or orange, as appropriate. Click to open the Pins context table to show the tie values on all bus pins.
	✓		Half-filled diamond (orange): multiple connections on bus, some pins tied or unconnected. Click to open the Pins context table to show the tie values on all bus pins.
	✓		Red triangle: pin with coercion (connection against a declared direction). Click to open the Tracer context table and select the tracing destination.
	✓		Blue square: RTL connection (no connection in hierarchical model). Click to open the Text/HDL Viewer with a textual representation of this connection.
	✓	✓	Crossed circle (red): no connection.
	✓		Circle with 0/1/X/Z (green): tied to 0/1/X/Z. When shown on a bus port, click to open the Pins context table to show the tie values on all bus pins.
	✓		Circle with “a”: ambiguous tie values (mix of some or all of 0/1/X/Z) on bus. Click to open the Pins context table to show the tie values on all bus pins.
	✓	✓	Diamond with “+”: collapsed buffers and inverters. Click this symbol to expand. <ul style="list-style-type: none"> • Green: no hidden fanout. • Orange: fanout within collapsed region.
	✓	✓	Diamond with “-”: expanded buffers and inverters. Click this symbol to collapse. <ul style="list-style-type: none"> • Green: no fanout in collapsible region. • Orange: collapsing hides potential fanout.
	✓	✓	Collapsed callout. The green symbol indicates that the callout was added with the “add_schematic_callout -collapsed” command. View the callout text as a tooltip by mouse hover. The yellow symbol is a native callout, which you can expand with the “Collapse/Uncollapse callout” toolbar button.

Table 11-3. Marker Symbols (cont.)

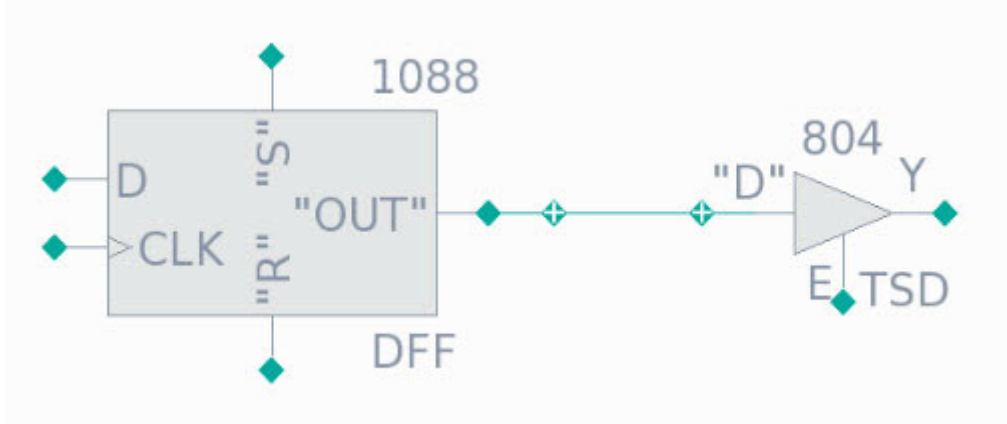
Symbol	Hier	Flat	Description
	✓		DRC violation callout. This symbol indicates that there are DRC violations related to the marked object. View the number of DRC violations by mouse hover. Click the symbol to open the list of violations in the DRC Violations table.
	✓		Red scissors: cut point (input connections cut or disconnected by the user or the tool).
		✓	Green scissors: cut point driver (single sink). Click the green scissors symbol to show the driver or sink port.
		✓	Orange scissors: cut point driver (multiple sinks). Click the orange scissors symbol to display a context table showing the drivers or sinks.
	✓		Orange polygon: pseudo-port (primary output added by the user or tool).  Note: Orange fill is also used to indicate primary input or output pseudo-ports in the flat schematic.
	✓	✓	Tessent Shell attributes markers with <code>gui_marking_index</code> set and “ <code>set_attribute_option -display_in_gui</code> ” enabled. <ul style="list-style-type: none"> • Filled circle: all displayed objects with the marker have a value set for the associated attribute. • Half circle: not all displayed objects with the marker have a value set for the associated attribute. • Empty circle: displayed in the attributes table; indicates that the attribute is set to the default value and hence is not marked on the schematic.

Buffer and Inverter Collapsing

Buffers and inverters can be collapsed and expanded in schematics to improve readability. Set the preference for this feature in the **Options** (gear) menu of the schematic toolbar.

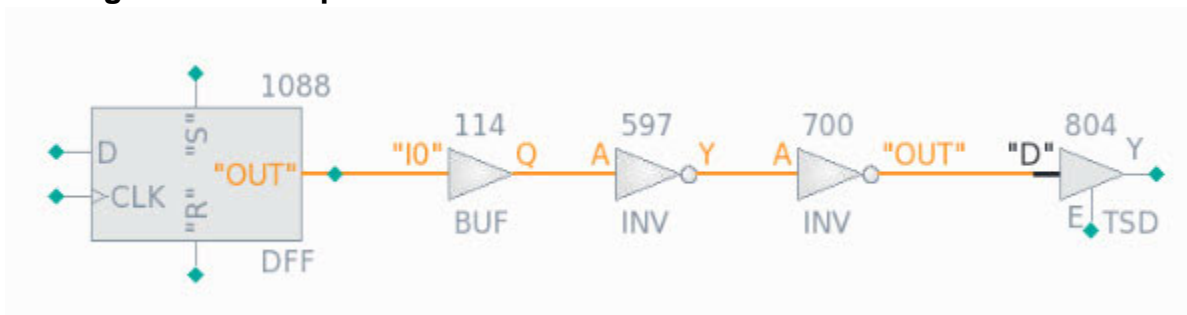
The collapsed and expanded symbols are shown in either green or orange. Green indicates that there are no hidden fanouts associated with the collapsed objects, as shown in [Figure 11-17](#):

Figure 11-17. Collapsed Buffers and Inverters With No Hidden Fanout



After expanding, there is no additional fanout, as shown in [Figure 11-18](#):

Figure 11-18. Expanded Buffers and Inverters With No Hidden Fanout



Orange indicates that there is additional fanout hidden by the buffer/inverter collapsing, as shown in [Figure 11-19](#) and [Figure 11-20](#):

Figure 11-19. Collapsed Buffers/Inverters With Hidden Fanout

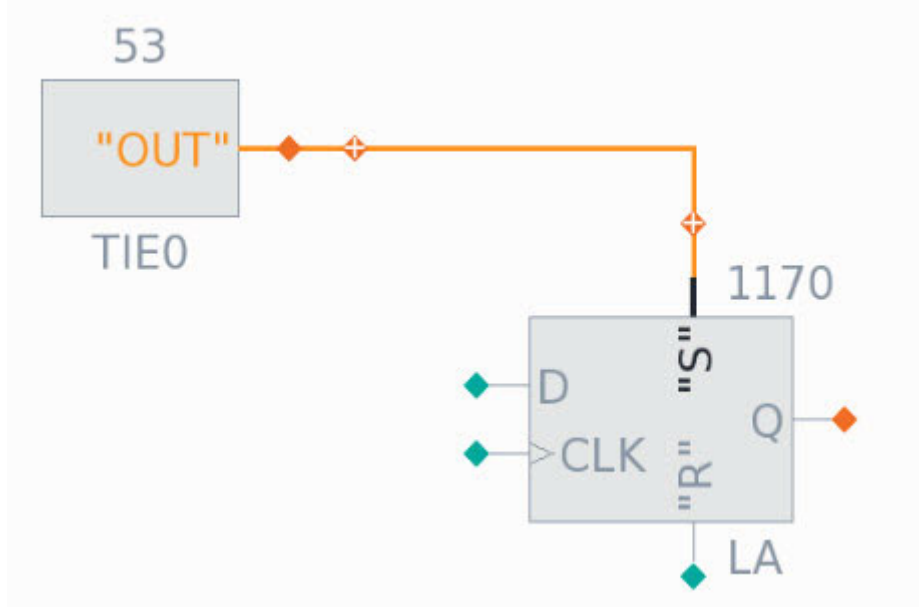
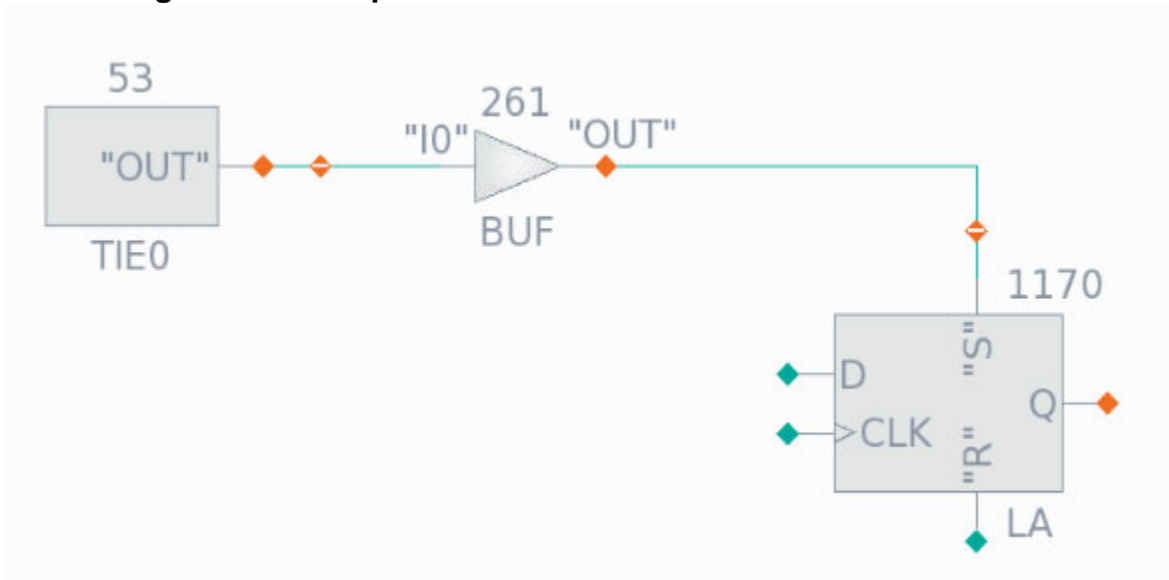
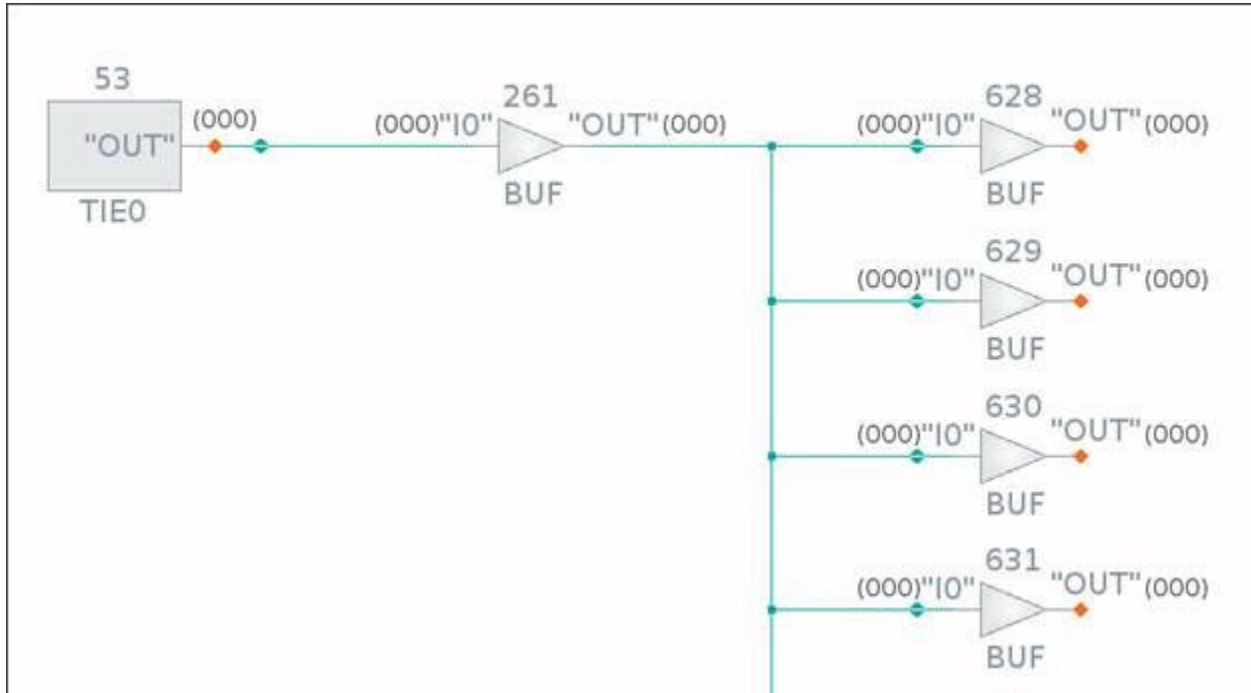


Figure 11-20. Expanded Buffers/Inverters With Hidden Fanout



Additional objects appear after adding the fanouts to the schematic, and the color of the markers changes to green as shown in [Figure 11-21](#):

Figure 11-21. Expanded Buffers/Inverters With Fanout Revealed



Note

Only buffers and inverters added implicitly to the schematic as a result of tracing are collapsible. These are identified with a gradient fill. Buffers and inverters added explicitly by name or gate ID are not collapsible and are identified with a solid fill.

Folding and Unfolding Instances

Cell grouping boxes in the Flat Schematic and instances in the Hierarchical Schematic can be folded and unfolded.

Folding and unfolding grouping boxes or instances is analogous to collapsing and expanding buffers and inverters, as described in “[Buffer and Inverter Collapsing](#)” on page 651. Double-click the grouping box or instance to do this. [Figure 11-22](#) shows how you can also unfold hierarchical instances by clicking the plus sign (+) symbol at the top-left corner.

Figure 11-22. Unfolding an Instance

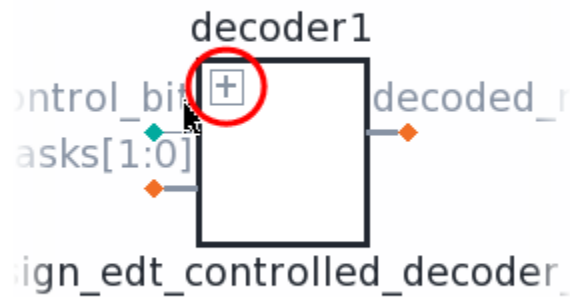


Figure 11-23. Folded Cell Group

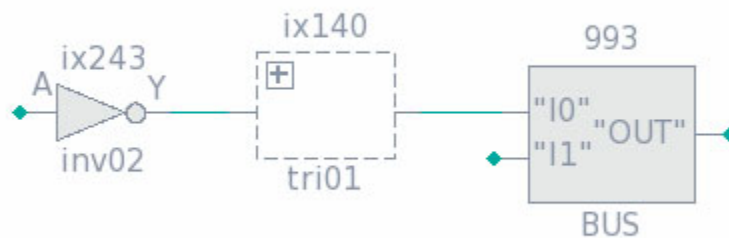
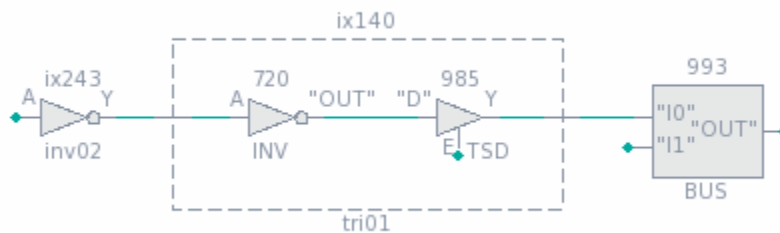


Figure 11-24. Unfolded Cell Group

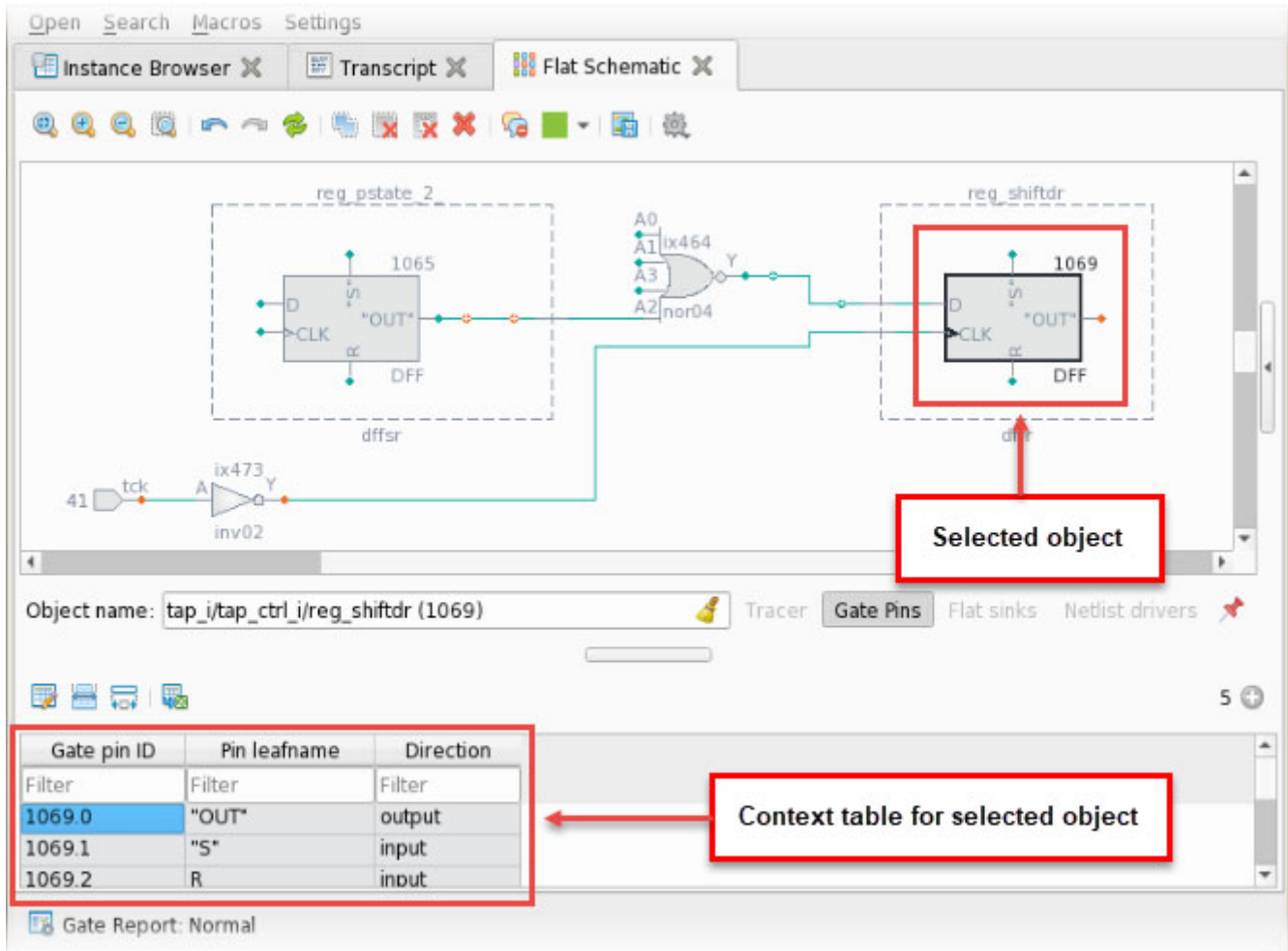


Context Tables

Schematics include tables to display information about the objects in the schematic. The tables take different forms depending on the current selection in the schematic.

Context tables are interactive. They display information, but you also use them to select and manipulate objects in the schematic. The following example shows a context table listing the data for a selected instance.

Figure 11-25. Schematic Elements: Context Table



Tracer

The context table for the tracing function provides information about the objects available to the tracing process.

When you trace from an ambiguous point (orange trace marker) or click the **Tracer** button next to the address bar when a pin is selected, the tracer context table opens. Use this table to add the next object in the tracing path to the schematic by double-clicking the appropriate row in the table. You can select the tracing direction and strategy, as described in [“Signal Net Tracing Strategies”](#) on page 665.

Figure 11-26. Tracer Context Table

Object name: Tracer Pins Instances Nets

Direction: Find drivers Strategy: Decision point

Pin name (start)	Pin name (end)	Distance
Filter	Filter	Filter
core_i/test_design_i/data1/q[5]	core_i/test_design_i/data1/reg_q_5_/Q	1
core_i/test_design_i/data1/q[4]	core_i/test_design_i/data1/reg_q_4_/Q	1
core_i/test_design_i/data1/q[3]	core_i/test_design_i/data1/reg_q_3_/Q	1

Gate Report: Normal

Pins and Gate Pins

Context tables are available for instance pins in the Hierarchical Schematic and gate pins in the Flat Schematic.

You can select an instance in the **Hierarchical Schematic** tab or **Flat Schematic** tab and click the **Pins** button or the **Gate Pins** button near the address bar, respectively, to view information about the pins belonging to that instance.

Figure 11-27. Context Table for Instance Pins (Hierarchical Schematic)

Object name: Tracer Pins Instances Nets

Direction	Pin name	Is part of bus?	Is pseudo-port?
Filter	Filter	Filter	Filter
input	core_i/test_design_i/data1/sen	false	false
output	core_i/test_design_i/data1/scan_out1	false	false
input	core_i/test_design_i/data1/scan_in1	false	false

Gate Report: Normal

Instances (Hierarchical Schematic)

A context table is available for instances in the **Hierarchical Schematic** tab.

Select an instance in the **Hierarchical Schematic** tab and click the **Instances** button near the address bar to view information about the child instances.

Figure 11-28. Context Table for Instances (Hierarchical Schematic)

The screenshot shows the Tessent Visualizer interface in the Hierarchical Schematic tab. The main window displays a schematic diagram of a component named 'data1' (a register8_scan1). The component has several inputs and outputs: 'clear', 'clock', 'd[7:0]', 'scan_in1', 'sen', 'q[7:0]', and 'scan_out1'. A black box highlights the component symbol. Below the schematic, the 'Object name' field contains 'core_i/test_design_i/data1'. The 'Instances' tab is selected in the bottom toolbar. A context table is displayed below the toolbar, showing the hierarchy of instances and their fault counts.

Type	Hierarchical name	Module name	Total Faults
Filter	Filter	Filter	Filter
Cell	core_i/test_design_i/data1/uu1	buf02	0
Cell	core_i/test_design_i/data1/reg_q_0_	sffr_ni	0
Cell	core_i/test_design_i/data1/reg_q_1	sffr_ni	0

Gate Report: Normal

Nets (Hierarchical Schematic)

A context table is available for nets in the **Hierarchical Schematic** tab.

Select an instance in the **Hierarchical Schematic** tab and click the **Nets** button near the address bar to list the nets one level below that instance.

Figure 11-29. Context Table for Nets (Hierarchical Schematic)

Object name: Tracer Pins Instances **Nets**

Net name	is_bus	is_array	index	leaf_name
Filter	Filter	Filter	Filter	Filter
core_i/test_design_i/data1/clear	false	false	0	clear
core_i/test_design_i/data1/clock	false	false	0	clock
core_i/test_design_i/data1/scan_in1	false	false	0	scan_in1

Gate Report: Normal

Flat Sinks (Flat Schematic)

Flat sinks are the set of primary inputs that you add to control internal pins. For example, you can add a primary input at the output of a PLL to model the clock, while keeping the PLL black-boxed.

The green dashed line in the following figure representing the original fanout net shows that this replacement is complete, and that there is only one user-added primary input associated with this fanout. The red scissors symbol indicates that a pseudo-port now drives that fanout.

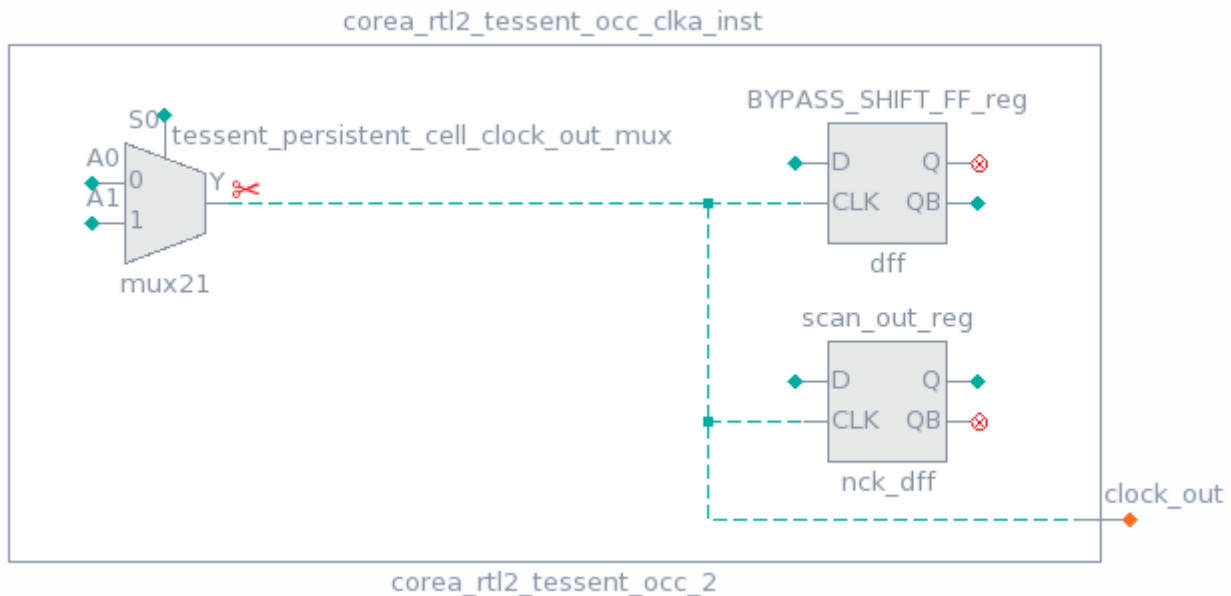
Figure 11-30. Fanout Replaced by User-Added Primary Input

Figure 11-31 shows the same MUX in the Flat Schematic, as well as the user-added primary input. The primary input is added to the schematic when you click the green scissors symbol on the MUX output.

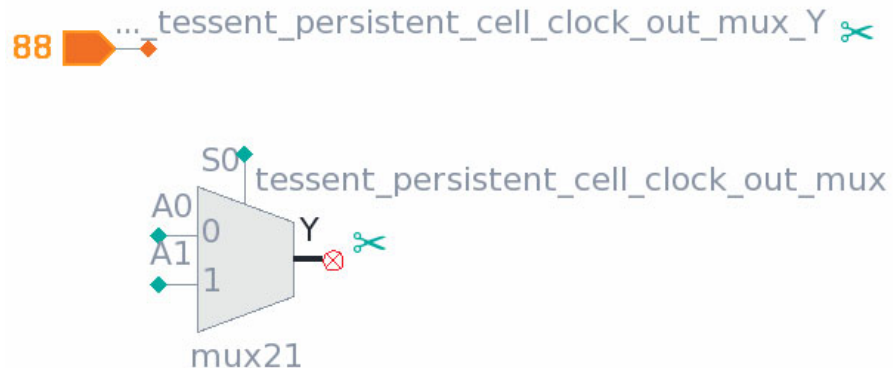
Figure 11-31. Flat Sink

Figure 11-32 shows a Hierarchical Schematic view of the same clock MUX, but in this case it has had all of its fanout replaced by multiple user-added primary inputs. The orange dashed line representing the original fanout net shows that this replacement is complete, and that there is more than one user-added primary input associated with this fanout. The red scissors symbols indicate that pseudo-ports now drive that fanout.

Figure 11-32. Fanout Replaced by Multiple User-Added Primary Inputs

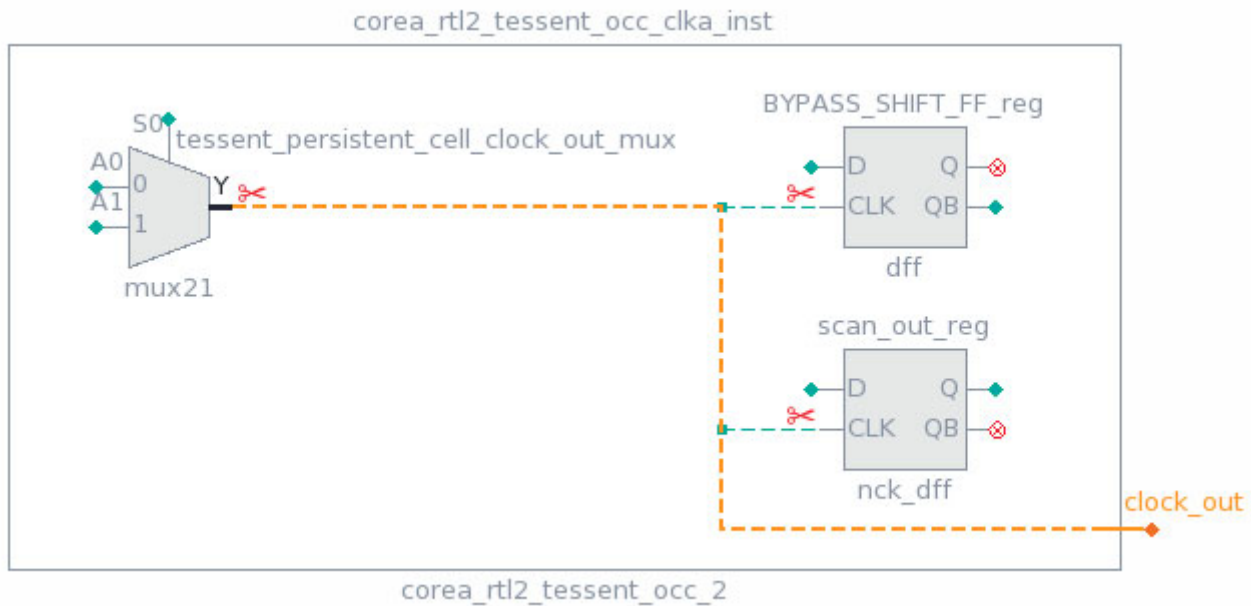
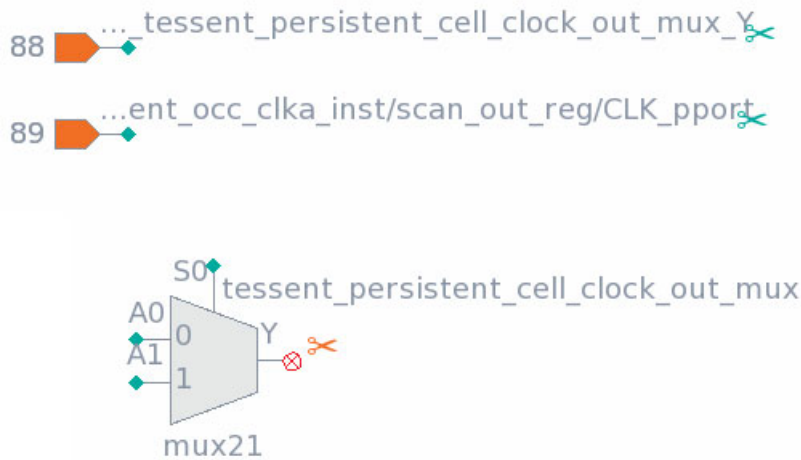


Figure 11-33 shows the same MUX in the Flat Schematic, as well as the user-added primary inputs. The primary inputs are added to the schematic when you click the orange scissors symbol.

Figure 11-33. Multiple Flat Sinks



Clicking the orange scissors symbol displays the context table for the user-added primary inputs, and double-clicking a row displays the flat sink.

Figure 11-34. Context Table for Multiple Flat Sinks

Object name: corea_rt12_tessent_occ_clka_inst/tessent_persistent_cell_cl

Gate pin ID	Pin leafname	Direction
Filter	Filter	Filter
88.0	corea_rt12_tessent_occ_clka...	output
89.0	corea_rt12_tessent_occ_clka... scan_out_reg/CLK_pport	output
90.0	corea_rt12_tessent_occ_clka... BYPASS_SHIFT_FF_reg/ CLK_pport	output

Gate Report: Normal

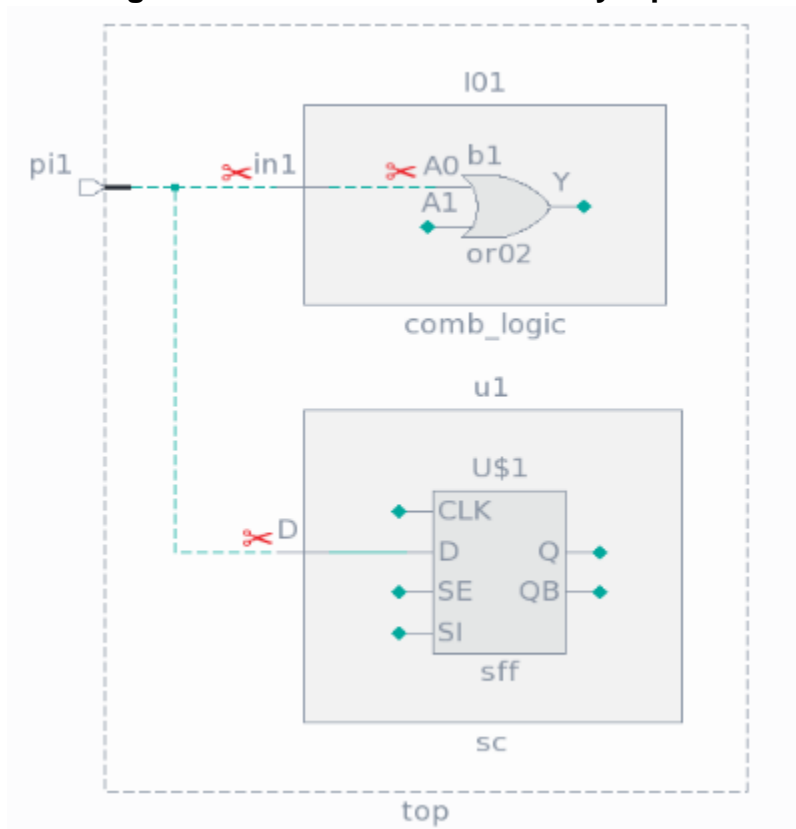
Netlist Drivers (Flat Schematic)

Netlist drivers are the original drivers of the fanout of user-added primary inputs.

Use the following commands to add primary inputs. These are indicated by the red scissors symbol in the **Hierarchical Schematic** tab.

```
add_primary_inputs I01/in1 -internal -pin INTERNAL_1
add_primary_inputs I01/b1/A0 -internal -pin INTERNAL_2
add_primary_inputs u1/D -internal -pin INTERNAL_3
```

Figure 11-35. User-Added Primary Inputs



As a result, the original fanout of the pi1 pin was disconnected, as indicated by the dashed line.

Figure 11-36 shows the pi1 pin in the Flat Schematic. The circle with the red “X” shows that it is disconnected, and the orange scissors symbol indicates that its original fanout is now driven by multiple user-added primary inputs. Click the orange scissors symbol to display the context table listing the flat sinks, as shown in the following figure.

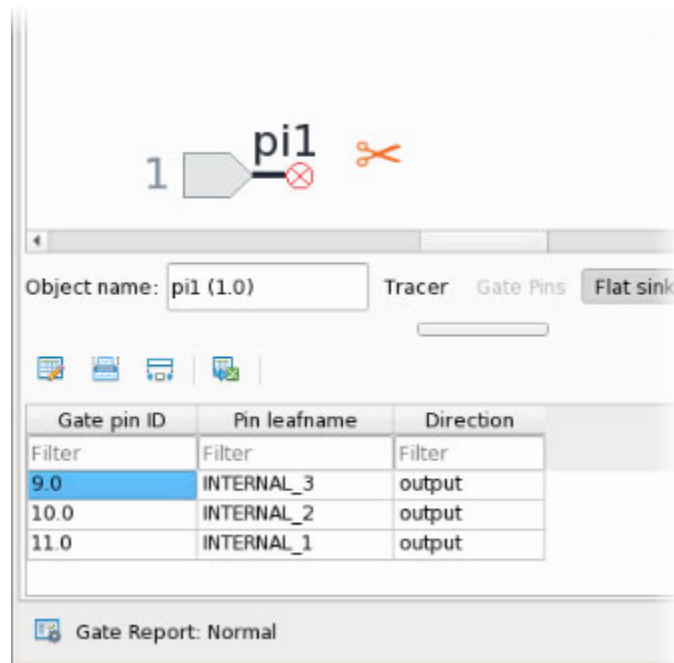
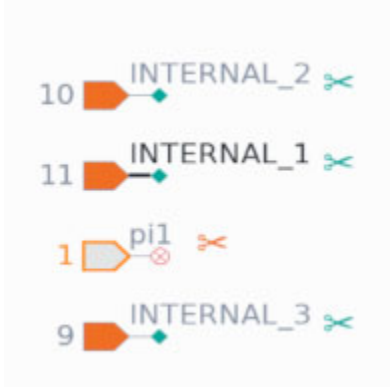
Figure 11-36. Netlist Driver and Context Table for User-Added Primary Inputs

Figure 11-37 shows the user-added primary inputs in the Flat Schematic. The orange fill indicates that these are user-added, and the green scissors symbol indicates that they have a single original driver. Clicking the scissors symbol shows the driver.

Figure 11-37. User-Added Primary Inputs (Flat Schematic)

Related Topics

[Markers](#)

Signal Net Tracing Strategies

The Flat and Hierarchical schematics feature multiple tracing strategies. They are available when the Tracer context table is activated for a pin, either by clicking an orange trace marker in

the schematic or the **Tracer** button after a pin is selected. The tracing strategies are common for both schematics.

For the green trace markers, the following tracing strategies are available in the schematics:

- Trace to decision point (default)
- Trace by one

Use the **Options** button (⚙️) in the schematic toolbar to select an option. A decision point is any point where a logic decision is made, such as a gate, state element, or a hierarchical pin with multiple fanouts, as shown in [Figure 11-38](#).

Figure 11-38. Decision Point Strategy on Hierarchy Boundary at q[6] Port

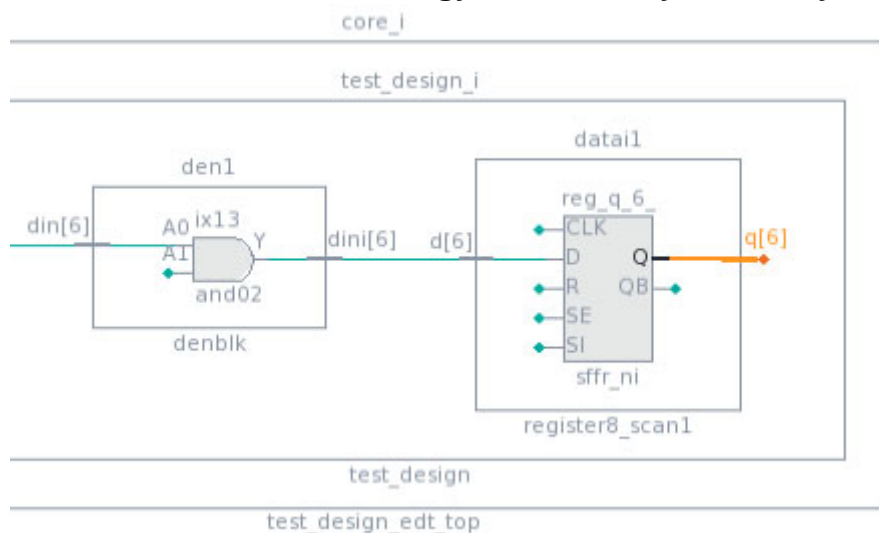


Figure 11-39. Choosing a Path From the Tracing Table

The screenshot shows the Tessent Visualizer interface. The 'Object name' field contains 'core_i/test_design_i/data1/q'. The 'Tracer' button is active. Below the object name, there are icons for various actions. The 'Direction' dropdown is set to 'Find drains' and the 'Strategy' dropdown is set to 'Decision point'. A table displays the tracing results:

Pin name (start)	Pin name (end)	Distance
Filter	Filter	Filter
core_i/test_design_i/data1/q[3]	core_i/test_design_i/data1/reg_q_3_/D	2
core_i/test_design_i/data1/q[2]	core_i/test_design_i/nonscanblock1/req dout 2 /D	2

At the bottom, there is a 'Gate Report: Normal' indicator.

You can continue tracing from the decision point by double-clicking one of the paths in the table.

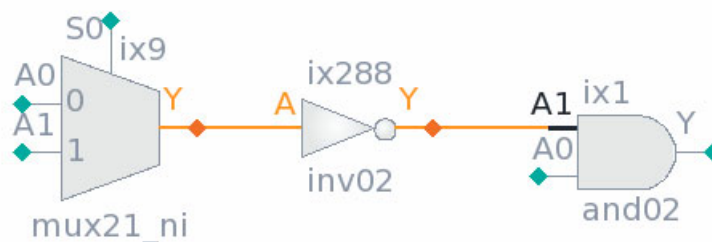
For additional information about tracing buses, see “[Buses](#)” on page 679.

Decision Point Tracing Strategy

Decision point tracing is the default strategy. In the decision point tracing strategy, a path is traced to the point where a logic decision is to be made. Tracing continues through buffers, inverters, wire primitives, and hierarchical boundaries as long as there is only one fanout (forward tracing) or fanin (backward tracing), and stops on other gates, cells, and logic primitives.

[Figure 11-40](#) shows the result of a trace using the Decision Point strategy. Pin A1 on the AND gate has been traced back through the inverter to the MUX, which is the first logic decision point.

Figure 11-40. Decision Point Strategy



Endpoint Tracing Strategy

When you use the endpoint tracing strategy, the tracing stops on any endpoint element. An endpoint is defined as any primary input or output, or any sequential element, memory, or black box. Tracing under the endpoint strategy traverses all buffers, inverters, logic gates, and hierarchical boundaries between the tracing start point (the pin selected on the schematic) and the endpoint, and the resulting paths includes those elements.

Nearest State Element Tracing Strategy

When you use the nearest state element tracing strategy, tracing stops on the first state element encountered. State elements are sequential elements or memories. Tracing under the nearest state element strategy includes all buffers, inverters, logic gates, and hierarchical boundaries between the tracing start point and the state-element endpoint, and the resulting paths include those elements.

Complete Fanin and Fanout Tracing Strategy

Complete fanin or fanout tracing performs full structural tracing. When tracing with the complete fanin and fanout strategy, the entire input or output logic cone is included in the results.

Stop on First Match Tracing Strategy

For the first match tracing strategy, you provide additional data in the form of a filter. The complete fanin / fanout strategy is used, but tracing stops when the first element matching the filter is found on the path. A single result is reported.

Trace by One Tracing Strategy

When you use the trace by one strategy, tracing stops at the next hierarchical level found in the Hierarchical Schematic, or the next logic element found in the Flat Schematic.

Displayed Property

All objects in Tessent Visualizer have a property called “displayed.”

The “displayed” property of an object refers to the corresponding schematic for the object, and indicates whether that object is added to the schematic. The context of being displayed is important for pin buses because all tracer actions are run for parts of the bus being added to the schematic. By default, the “displayed” property is indicated in tables by a distinct background color for a row, and can also be added as a column in those tables.

Tessent Shell Attributes

Tessent Visualizer includes functionality to view Tessent Shell attributes and annotate the objects displayed in schematics with them.

The Tessent Shell attributes for the currently selected object in the schematic are shown in a table.

Figure 11-41. Tessent Shell Attributes Table

The screenshot displays the Tessent Shell interface. On the left, a schematic window shows a component named 'bypass_logic_i' with a pin 'edt_scan_in[3:0]'. A pink circular marker is placed on this pin, and a tooltip indicates 'direction = output'. The 'Attributes Table' on the right lists the following attributes:

GUI	Name	Value
	Filter	Filter
	design_hierarch...	3
●	direction	output
	has_functional_s...	true
	index	<multiple values>
◐	is_MSB	<multiple values>
	is_bus	true
	is_created	false
	is_hard_module	false
○	is_non_editable	false
	is_non_editable_...	

Below the attributes table is a 'Pin Attributes Table' with the following data:

Pin leafname	Direction	Gate data
Filter	Filter	Filter
edt_scan_in[3]	output	
edt_scan_in[2]	output	
edt_scan_in[1]	output	

By default, only those attributes with non-default values, or that were registered with the “-show_default” option, are shown. You can use the **Show all** checkbox to show all attributes, including those with default values.

You can double-click a cell in the GUI column to select a color in the GUI marking index and show a corresponding marker near schematic objects that have that attribute set to a non-default value. Hover the mouse pointer over that marker in the schematic as shown in [Figure 11-41](#) to show a tooltip with the attribute name and value.

Note

For Boolean attributes, the marker is only shown if the value is set to “true.”

An empty circle in the GUI column indicates that the attribute is set to the default value. A corresponding marker is not displayed in the schematic window in this case. A half-filled circle

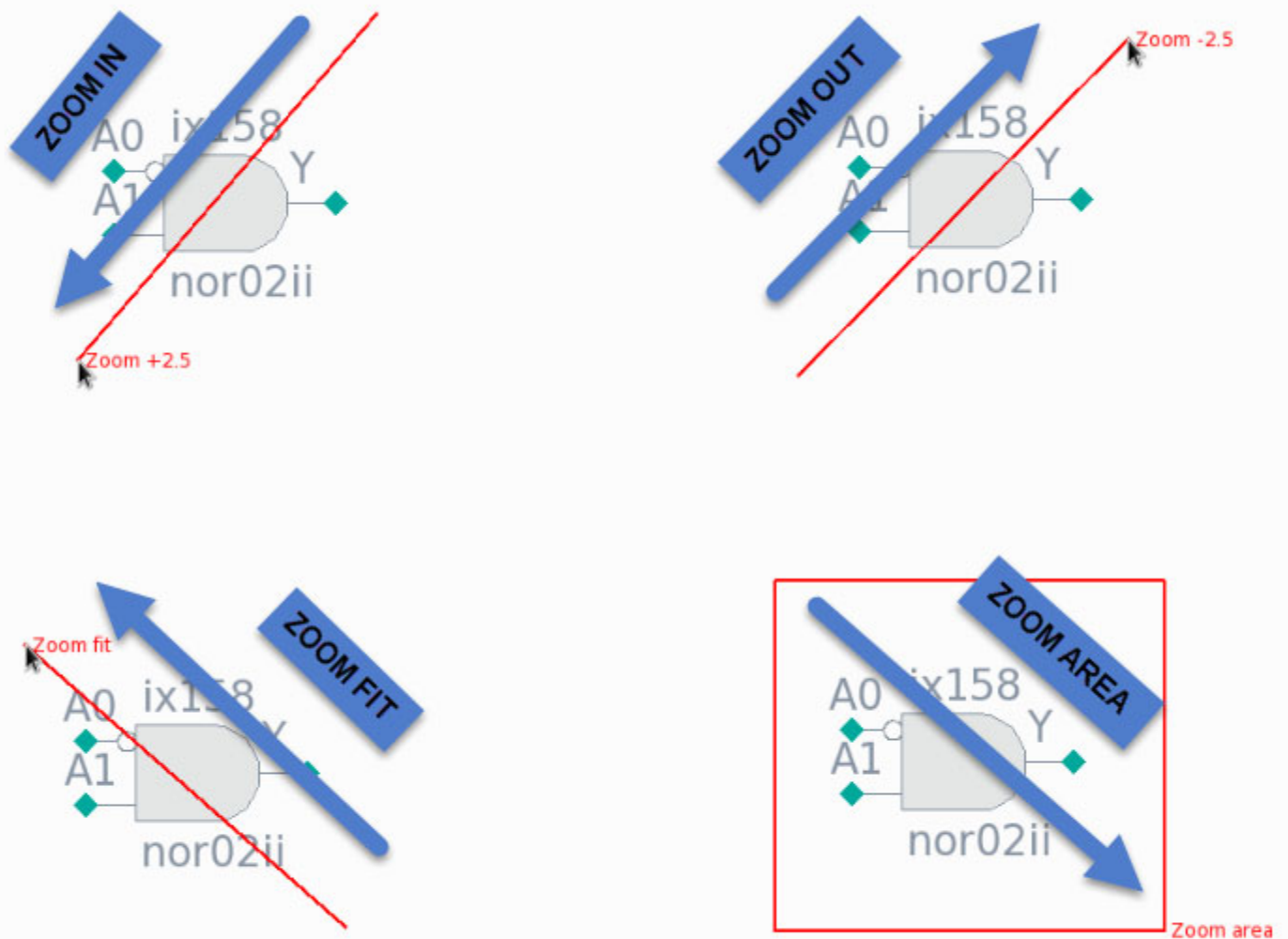
applies to buses only and indicates that not all of the pins composing the bus have the relevant attribute set to nondefault values. Additionally, the “Value” column indicates “<multiple values>” for buses where the values for this attribute of separate pins are not consistent.

Mouse Gestures

Zoom in or out within schematics using the left mouse button.

Figure 11-42 summarizes the four mouse gestures (stroke commands) available to perform a zoom.

Figure 11-42. Mouse Gestures in Tessent Visualizer Schematics



You can also zoom in and out using the scroll wheel of your mouse.

Additional mouse gestures are available:

- Shift-click and drag with the left mouse button: area select.
- Click and drag with the scroll wheel: pan the view.

Customize the mapping of mouse gestures to specific mouse buttons and keyboard modifiers in the Preferences dialog box. See “[Tessent Visualizer Preferences](#)” on page 671.

Tessent Visualizer Preferences

Customize Tessent Visualizer with the Preferences dialog box, which is available from the **Settings** pulldown menu.

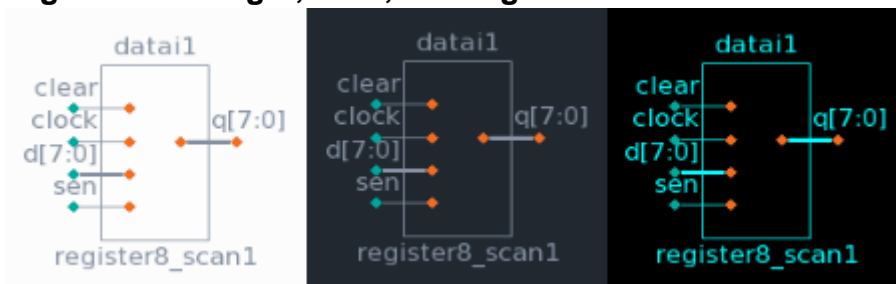
The Preferences dialog box contains several tabs:

- **Schematics**
- **Tables**
- **Text viewers**
- **Other**

Use the **Schematics** tab to define how schematics look and operate. Set maximum lengths for names, set the width of net representations, redefine how the mouse works when interacting with schematics, and choose the color theme.

There are three predefined color themes available in Tessent Visualizer schematics: light, dark, and high-contrast. Set the color scheme with the Preferences dialog box.

Figure 11-43. Light, Dark, and High-Contrast Color Themes



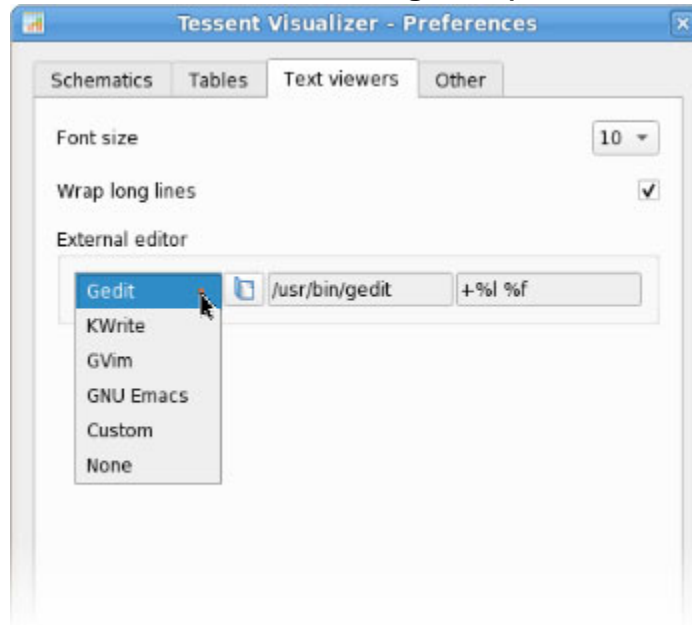
Use the **Tables** tab to set the maximum number of rows loaded in tables and to show line numbers in tables.

Use the **Text viewers** tab to set the font size and line-wrapping policy in the Text Viewer, and to define an external text editor that can be launched from the Text/HDL Viewer tab. Choose from several common text editors, or define your own preferred one. Use the variables “%1”


(line number) and “%f” (filename) as arguments to the text editor executable. For example, this text editor definition opens the file currently in the Text/HDL Viewer tab in the Gedit editor with the cursor positioned at the current line:

```
/usr/bin/gedit +%l %f
```

Figure 11-44. Preferences Dialog Box (Text Viewers Tab)



Note

 Tessent Visualizer invokes whichever executable you specify as the external text editor. Ensure that valid and working options are set here.

Use the **Other** tab to set the global font size.

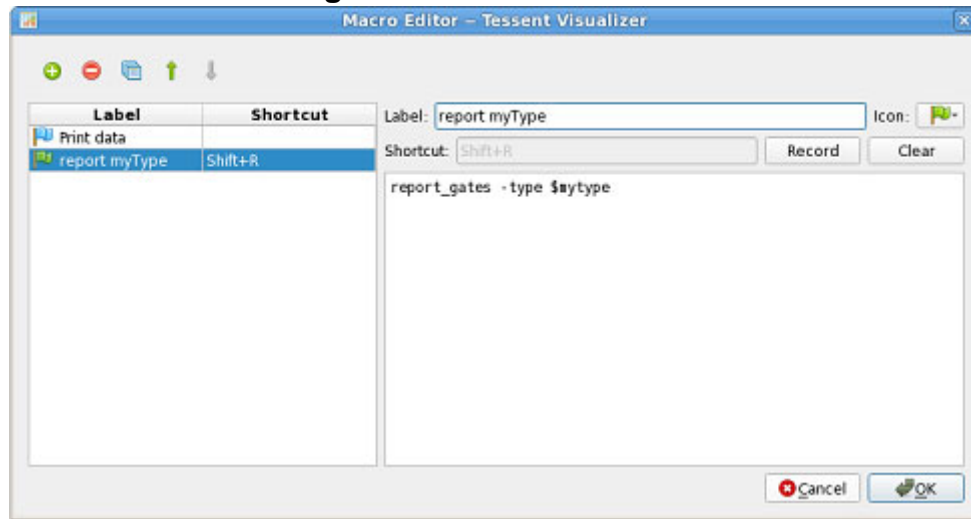
Macros

You can define a macro for any script or command that you can run in the Tessent Shell environment.

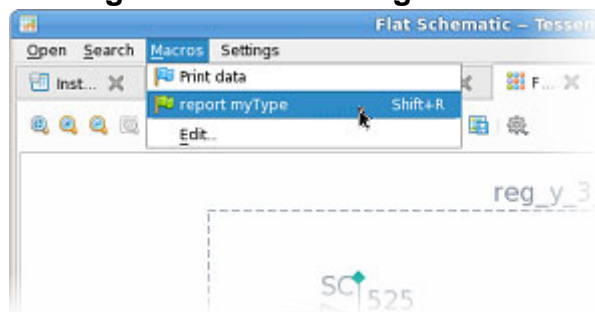
Open the **Macros** menu to define or run custom macros. You can use the Macro Editor dialog box to create new macros or modify existing macros. [Figure 11-45](#) shows the Macro Editor with two simple macros defined. With the Macro Editor, you can do the following:

- Add, remove, or duplicate a macro.
- Reorder the macros in the **Macros** menu.
- Record a keyboard shortcut for a macro.
- Undefine a macro keyboard shortcut.

- Select an icon for a macro.

Figure 11-45. Macro Editor

The following example shows the macros menu after creating the “report myType” macro in the editor. In this example, the macro runs the `report_gates` command with the `-type` option and the predefined variable shown in [Figure 11-45](#) above.

Figure 11-46. Running a Macro

You can define a keyboard shortcut, for example, Shift+R, for a new macro. You can run the macro directly from the macros menu, or from the keyboard using that shortcut. If you attempt to record a keyboard shortcut for a macro using an existing shortcut, an error message is reported.

Tooltips

Many Tessent Visualizer features have associated tooltips, which are informational text popups that appear when you hover the mouse pointer over a graphical element such as an object, filter box, or button.

[Figure 11-47](#), [Figure 11-48](#), and [Figure 11-49](#) show examples of tooltips for several Tessent Visualizer features.

Figure 11-47. Tooltip for a Collapsed Buffer Indicator

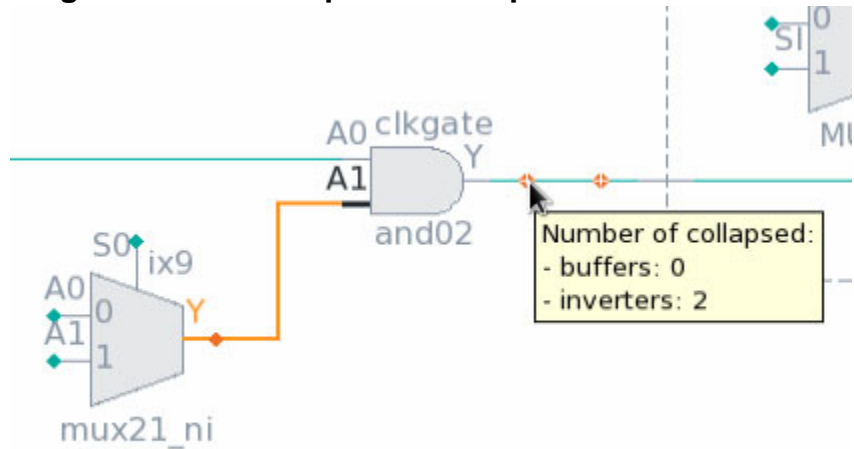
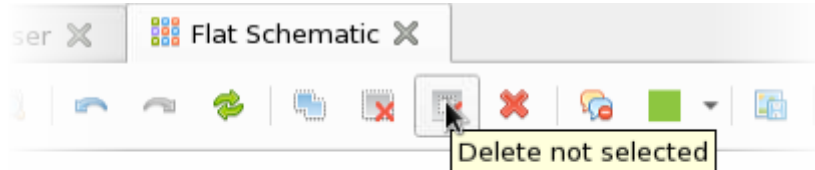


Figure 11-48. Tooltip for a Filter Box

Type	Instance name	Module name	AU	Total Faults
Filter	Filter	Filter	Filter	Filter
Inst	se	Exact match: = 'value'		0
Inst	cl	Match inverse: != 'value'		0
Inst	pl	Operators: <, <=, >, >=		0
Inst	ad	Wildcards: GLOB 'expr'		0
Inst	ad	Regular expression: REGEXP 'expr'		0
Inst	ad	Logical operators: AND, OR, NOT GLOB, NOT REGEXP		0
Inst	den1	denblk	0	0
Inst	data1	register8_scan1	0	0
Inst	data1	register8_scan1	0	0


Figure 11-49. Tooltip for an Action Button



Gate Report Settings

The Gate Report Settings dialog box specifies the information displayed in Tessent Visualizer schematics.

The Gate Report Settings dialog box is a GUI front end for the `set_gate_report` command. The dialog box provides access to a subset of the `set_gate_report` options. These options are active depending on the data models available in Tessent Shell.

Access the dialog box by clicking the **Open Gate Report Settings dialog** button  at the bottom of the main Tessent Visualizer window or from the **Settings > Gate Report** menu.

See the `set_gate_report` command for a description of the simulation values shown in the schematics and tables that correspond to the values set in the dialog box.

Related Topics

[set_gate_report](#)

[report_gates](#)

Saving and Restoring the Session State

The session state, such as the tabs appearing in a window and specific user preferences, is stored in a hidden file in your home directory. Some other specific settings such as filtering and instance highlighting persist in memory until the Tessent Visualizer server is closed. For example, if you close a Tessent Visualizer client and later start a new client connected to the same server, the session appears as you left it.

You can save and restore specific configurations with **Settings > Export configuration** and **Settings > Import configuration**. You can also start Tessent Visualizer with a specific configuration using `-import_configuration` option of the `open_visualizer` with either the `open_visualizer` command or **tessent -visualizer** invocation.

Window Title Prefixes

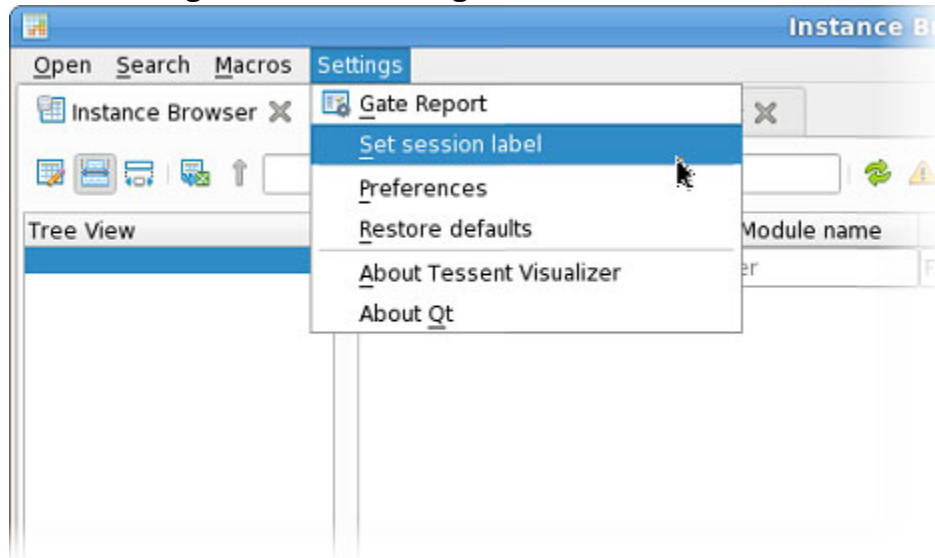
You can define a prefix to be added to the title bars of the Visualizer windows, to distinguish between multiple sessions running on the same machine.

To define a window title prefix, invoke Tessent or open the Visualizer session with the `-label` option:

```
% $TESSENT_HOME/bin/tessent -visualizer -label name  
SETUP> open_visualizer -label name
```

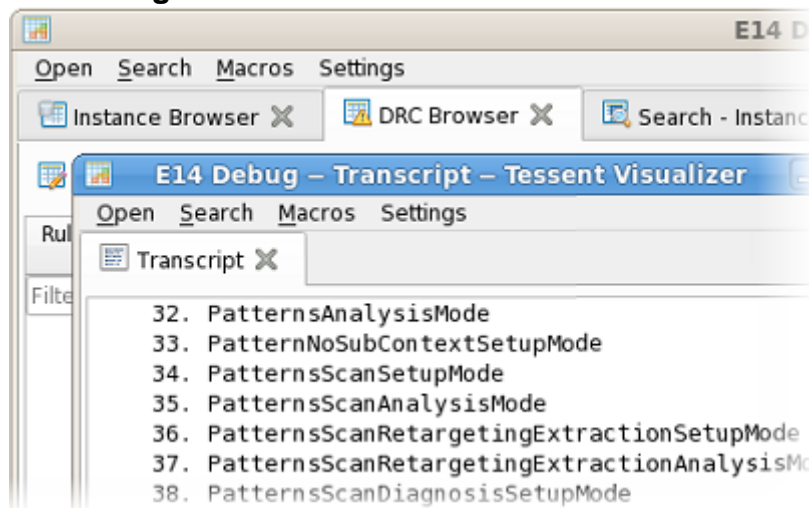
Alternately, set the prefix directly in Visualizer as shown in [Figure 11-50](#):

Figure 11-50. Setting a Window Title Prefix



The standard window table is prepended with the string you set, as shown in [Figure 11-51](#). In addition, “Tessent Visualizer” is added to the window title as a suffix:

Figure 11-51. Visualizer Window Label



Tessent Visualizer GUI Reference

The Tessent Visualizer GUI consists of several tabbed windows that function as different views on design data. You can look at objects as part of schematics, browse tables, view a report, or look at different types of syntactically-highlighted text.

Hierarchical Schematic	677
Flat Schematic	682
Instance Browser	684
Wave Generator	686
Cell Library Browser	688
DRC Browser	689
Pin Data	691
Transcript	692
Text/HDL Viewer	693
Diagnosis Report Viewer	695

Hierarchical Schematic

This section describes features available only in the **Hierarchical Schematic** tab. The Hierarchical Schematic shows a representation of the design before design flattening. Use this feature to explore the structure of your design and the relationships between the elements making up the design.

Note



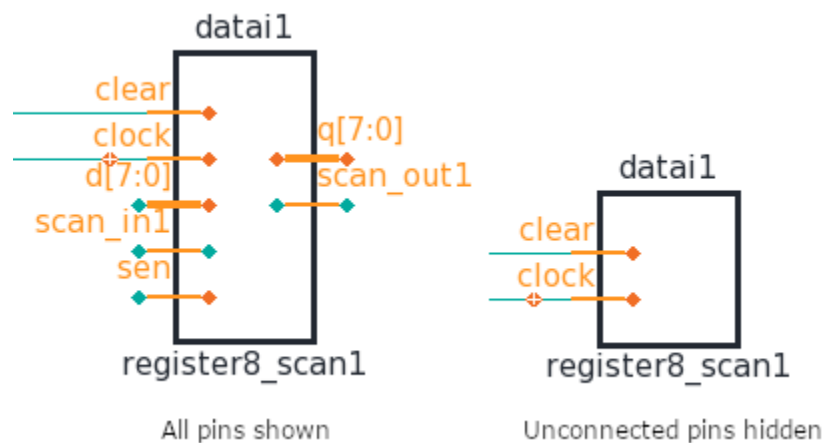
“[Schematics](#)” on page 639 describes schematic features common to both the Hierarchical Schematic and the Flat Schematic.

You can use the tools available from the [Toolbar](#), or you can use the context menu to do the following:


- Show the selected object(s) in the Flat Schematic.
- Show the HDL definition for the selected object in the [Text/HDL Viewer](#).
- Show the HDL instantiation for the selected object in the Text/HDL Viewer.
- Show all pins, or hide unconnected pins, on the selected instance (as shown in [Figure 11-52](#)).
- Show the internal connectivity of a hierarchical instance.
- Show an instance as parent in the Instance Browser.

- Delete all selected objects from the schematic.
- Delete all unselected object(s) from the schematic.
- Trace backward.
- Report gates on selected pins.
- Copy the post-synthesis names of selected objects to the system clipboard.
- Copy the active names of selected objects to the system clipboard. Active names are compatible with Tessent introspection commands.


Figure 11-52. Hierarchical Instances With Pins Shown and Hidden



Note

 The menu items **Show HDL definition** and **Show HDL instantiation** are available only when the Tessent Shell context is set to “dft -rtl.” The **Show HDL definition** menu item is available for library cells in all contexts when the library cell file is available.

Note

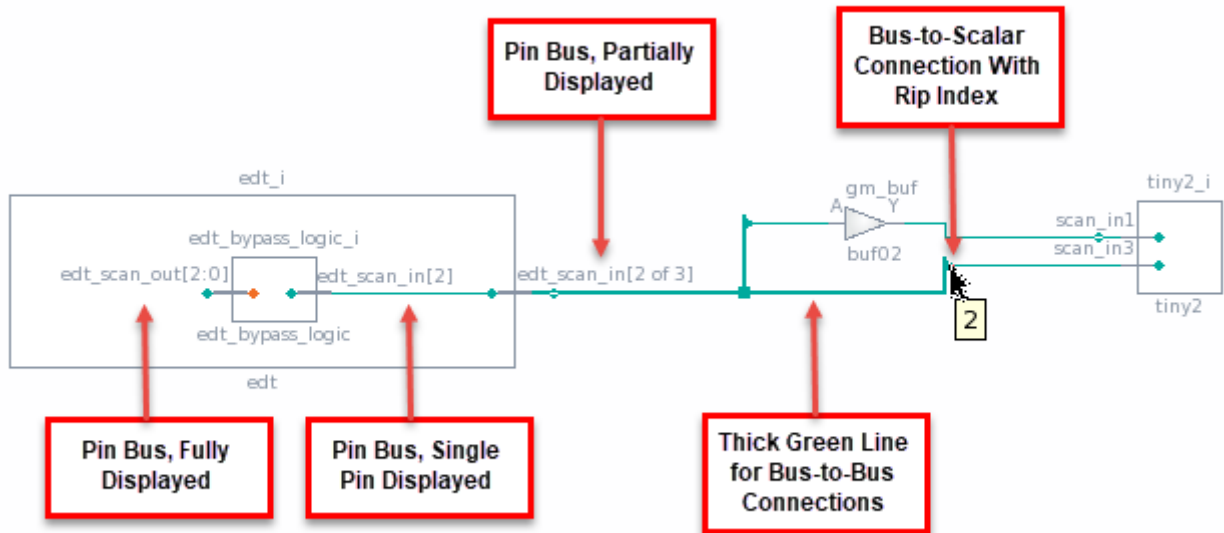
 By default, all pins are initially shown on an instance when it is added to the Hierarchical Schematic if the total number of single pins and bus ports is 16 or fewer.

The following tables relate specifically to the Hierarchical Schematic:

- **Instances Table** — Displays information about the instances contained within a selected instance. For more information, see “[Instances \(Hierarchical Schematic\)](#)” on page 658.
- **Nets Table** — Displays information about the nets of a selected instance. For more information, see “[Nets \(Hierarchical Schematic\)](#)” on page 659.

Figure 11-53 shows several symbols specific to the Hierarchical Schematic:

Figure 11-53. Hierarchical Schematic Symbols



Design hierarchy is shown with rectangles surrounding the objects in a design instance, and library cells are shown without rectangles and filled with gray. The leaf-level instance names are displayed above each instance, and the module or cell names are displayed below them.

Buses

The Hierarchical Schematic can display either complete or partial buses, using the following naming conventions:

- **Fully displayed** — Shows the full range. For example, “scan[7:0]”.
- **Partially displayed** — Indicates the count of displayed pins versus the total pin count. For example, “scan[2 of 8]”.
- **Single pin of a bus** — The index of the displayed pin is shown. For example, “scan[3]”.

You can add or remove bus pins in the display from any table that lists hierarchical pins, such as the Pins table in the Instance Browser or the Pins context table at the bottom of the Hierarchical Schematic.

You can also add bus pins with the following procedure:

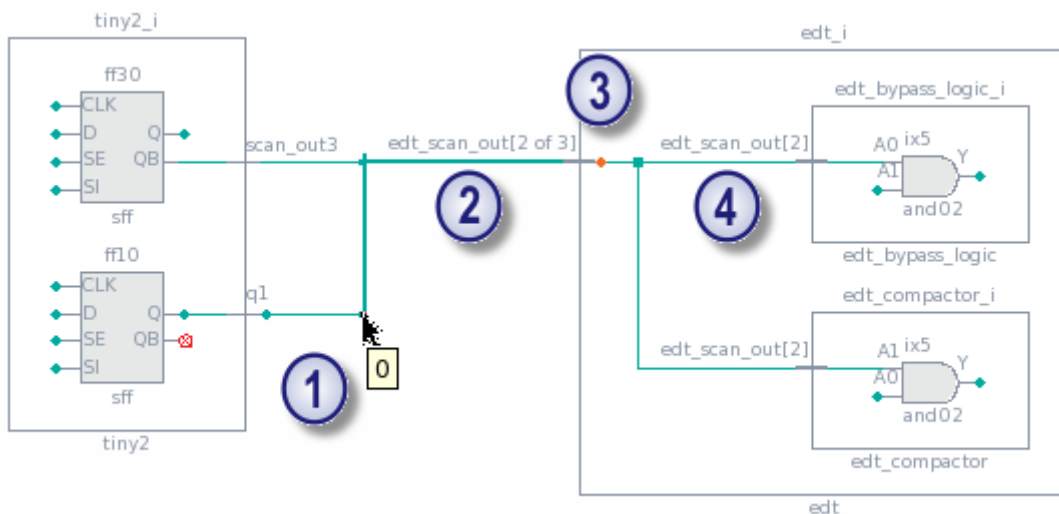
1. Select the instance in the Hierarchical Schematic. The instance name is displayed in the address bar (see [Figure 11-6](#) on page 641).

2. Append text to the instance name to add the bus pins. For a single pin, include the pin index (for example, "data[3]" in "mytop/parent_module/this_instance/data[3]"). To designate the complete bus, use the "*" wildcard character (for example, "data*").
3. Press Enter. The address bar shows both the pins and the nets. Click the pins line. The pins are added to the instance in the schematic.

The display of bus pins affects the operation of the tracing function, because the tracer starts from displayed bus pins. Add individual pins for tracing to the Hierarchical Schematic, or apply filtering in the tracer table for the pins of interest.

Tessent Visualizer uses several naming and symbol conventions for buses in schematic windows. See [Figure 11-54](#) for examples.

Figure 11-54. Bus Tracing in the Hierarchical Schematic

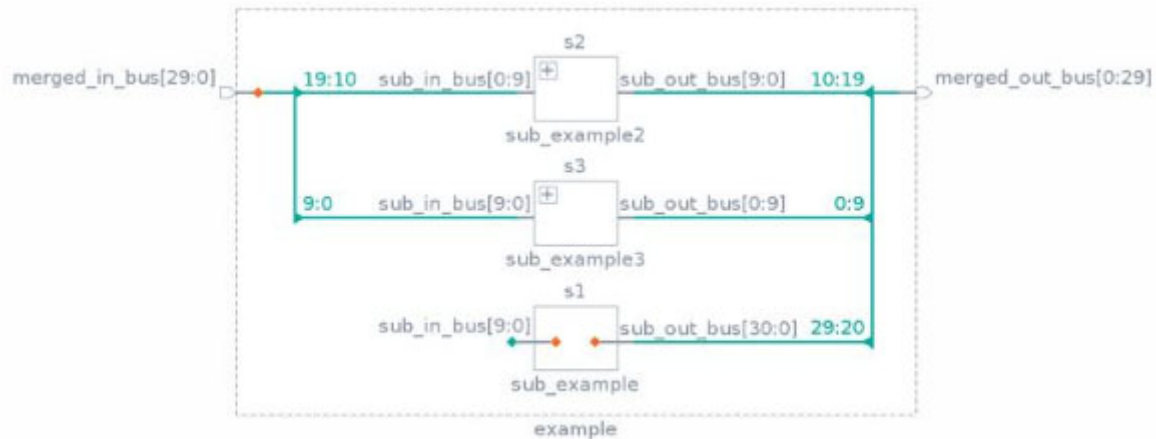


Item	Description
①	Bus-to-scalar connection with rip index ("0" in this case) shown with mouse pointer hover over rip symbol.
②	Generated name for a partial bus (two bits of a three-bit bus in this case).
③	Orange diamond representing multiple paths not shown from this bus pin.
④	Generated name of a partial bus with a single bit displayed (bit index 2, in this case).


Buses can be split into sub-ranges as shown in [Figure 11-55](#). In this case, ten bits of the 31-bit output bus from instance s1 are merged with all ten bits of the output buses from instances s2 and s3 to form the 30-bit merged_out_bus.

Bus rip indices can also be shown directly on the hierarchical schematic by choosing this feature from the options menu.

Figure 11-55. Bus Sub-Ranges



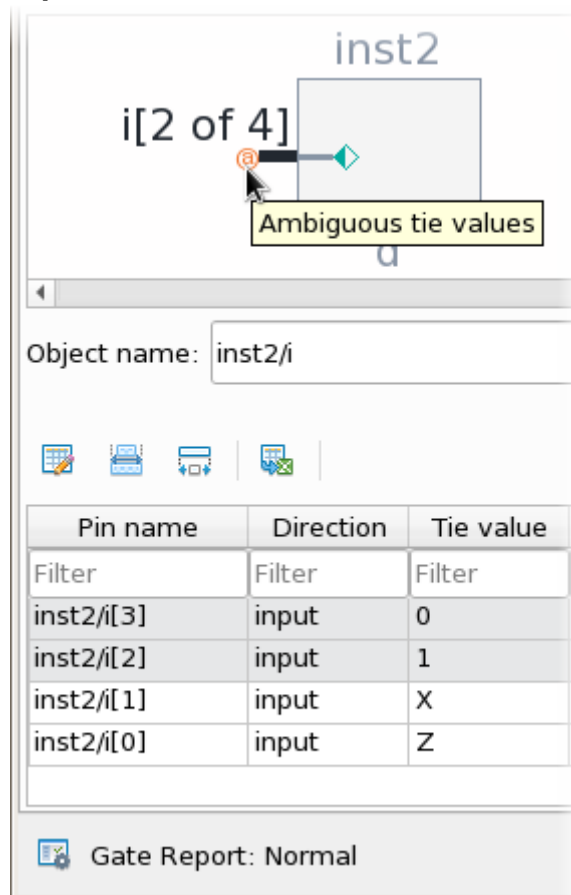
Note

 This schematic does not indicate which ten bits of the output bus from s1 are connected. Use the Tracer table to determine connectivity.

Other Symbols

Additional marker symbols indicate other hierarchical schematic features, such as tied-value indicators. A tied-value indicator is displayed when a net is tied to a logic value or an unknown. See [Figure 11-56](#) for an example, and [Table 11-3](#) on page 649.

Figure 11-56. Example of a Tied-Value Marker With Associated Pin Table



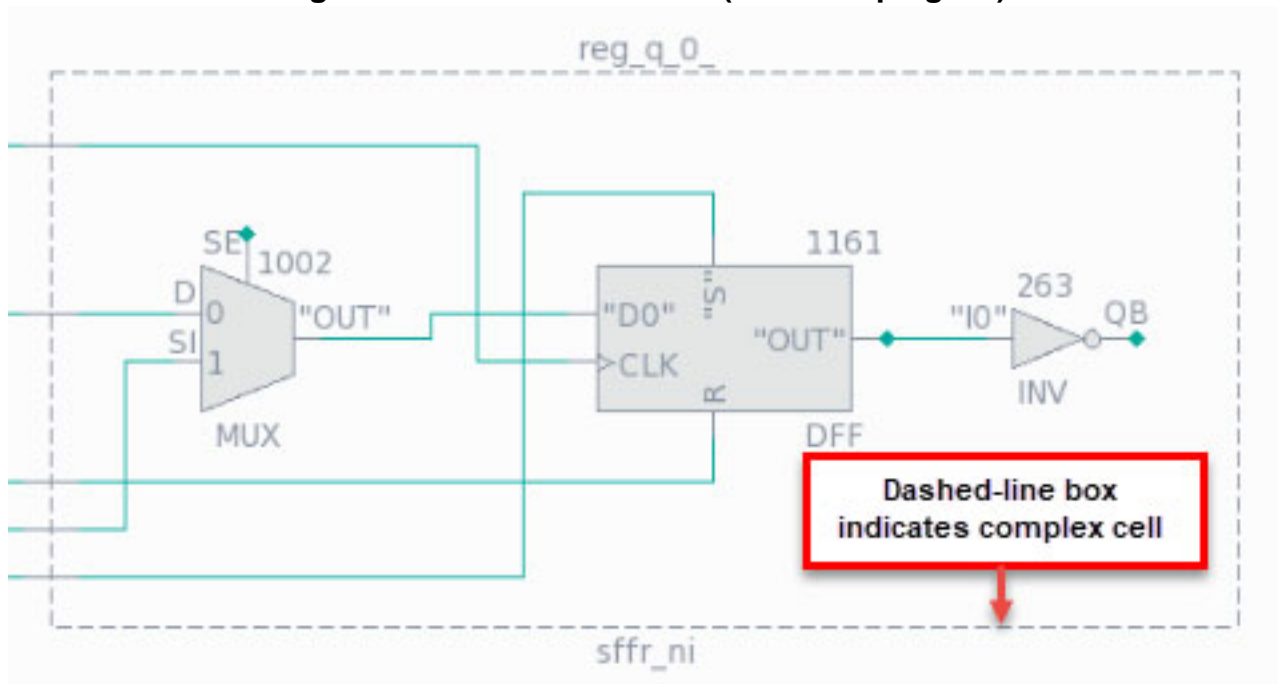
Flat Schematic

The Flat Schematic shows a representation of the design after design flattening. Use the Flat Schematic to analyze and debug issues when the details required for analysis are not available in the Hierarchical Schematic. For example, simulation data and functional representations of hierarchical and library cells are available in the Flat Schematic.

Library and hierarchical cells in the design appear in the Flat Schematic as gates. Cells that consist of a single gate are displayed with the conventional symbols for those gates.


Figure 11-57 shows cells that consist of multiple gates, displayed either with or without a bounding box showing the grouping. You can control this display option using the “Cell grouping” checkbox in the **Options** menu available from the toolbar.

Figure 11-57. Flat Schematic (Cell Grouping On)




You can right-click the dashed-line bounding box and use the context menu to delete the instance from the Flat Schematic, show it in the Hierarchical Schematic, or show all gates within the grouping.

Note

 In a library cell bounding box, a NAND function that has at least one (but not all) inputs inverted is displayed as an OR symbol. A NOR function that has at least one (but not all) inputs inverted is displayed as an AND symbol.

Note

 When cell grouping is turned on, instances shown on the schematic are identified by their module or primitive names (below the symbol) and leaf-level cell library names (above the symbol). When it is turned off, they are identified by their module/primitive names and their gate IDs from the flat model.

Many features of the Flat Schematic are similar to those in the [Hierarchical Schematic](#). However, one difference is that you cannot control the display of pins on most instances, with the exception of RAM and ROM instances. You can add or remove pins for these instances by using the popup menu or by dragging and dropping from the pins table.

Pin names on instances appear in quotation marks when the pin is not part of the design hierarchy.


Instance Browser

Use the Instance Browser to navigate your design and obtain reports about its elements. It is analogous to a file browser.

The Instance Browser consists of four major elements:

- **Tree View** — A pane that shows the instances in your design hierarchy directly.
- **Child Instances table** — A pane that displays information about the children of the instance currently selected in the tree view or the Address Bar.
- **Address Bar** — A text field that shows the instance currently selected in the tree view and the parent instance for instances displayed in the Child Instances table. Type an instance name in this field to select an instance without using the Tree View or Child Instances table.
- **Pins and DRC Violations tables** — A table that lists the pins or DRC violations for the instance(s) currently selected in the Child Instances table.

Note

 For huge designs, it may be more efficient to select an instance and then use the filtering functions in the Child Instances table to find objects and explore the hierarchy from the search results.

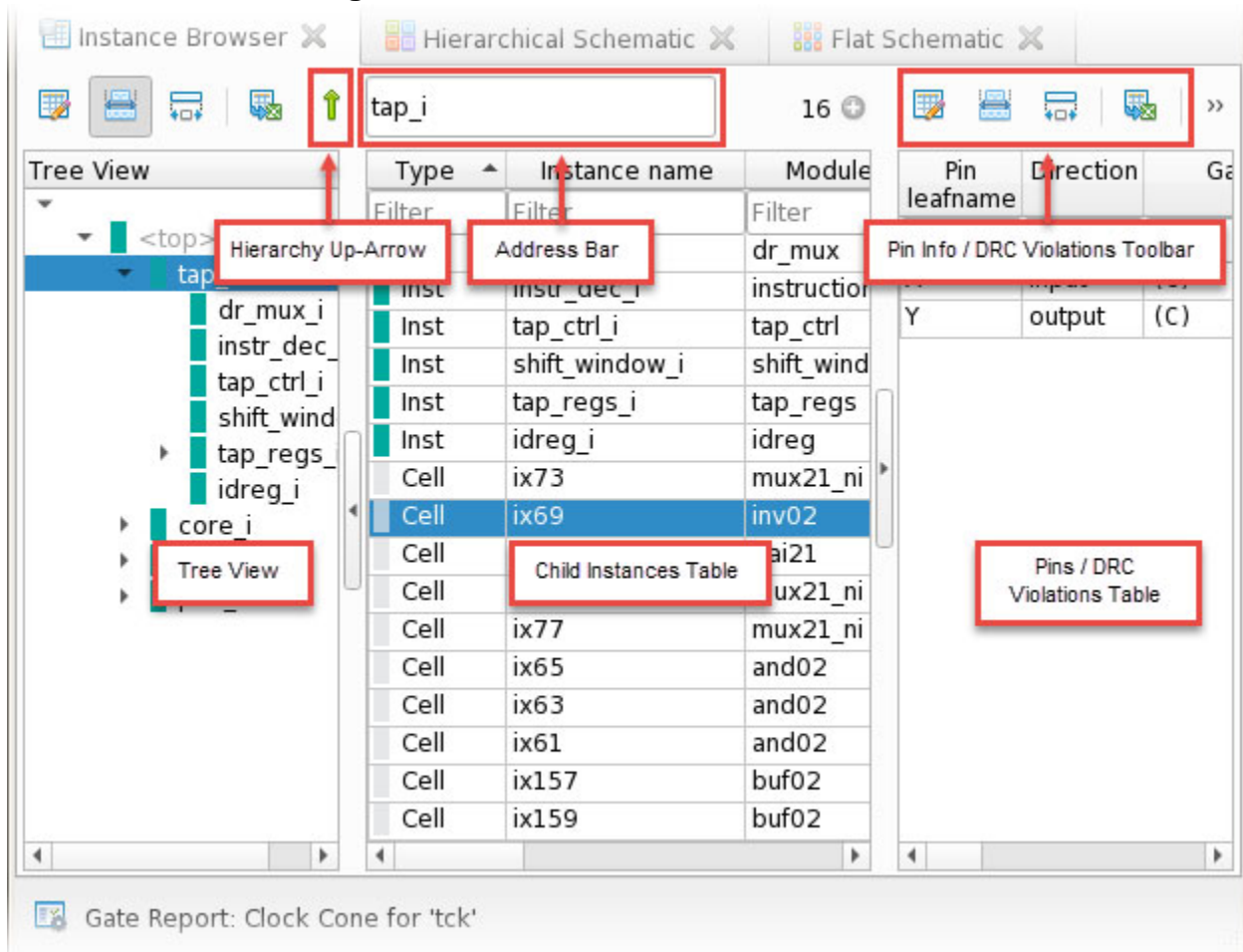
Click an instance in the tree view to display that instance's contents in the Child Instances table. Double-clicking an instance in the tree view expands the node in the tree. First, the node is selected, and it becomes the parent instance. If the double-clicked node is different than the one previously selected, the Child Instances table is reloaded with the children of the new parent.

Select an instance of any type in the Child Instances table to display its pins or DRC violations in the appropriate context table. Double-click, or use the Enter key, to open the next level of hierarchy (if any) in the Child Instances table. Press the Backspace key to navigate up one level of hierarchy.

Show instances of cells or modules in the Hierarchical Schematic, the Flat Schematic, or the Text/HDL Viewer by right-clicking the cell or module in the tree view or Child Instances table and choosing the appropriate option from the popup menu. Add a pin to the pin data table by right-clicking the pin in the Pin Information table and choosing the **Add to Pin Data** menu item.

Copy the active or post-synthesis name of an instance to the system clipboard by right-clicking the name in the tree view and choosing the appropriate option from the popup menu. Active names are compatible with Tessent introspection commands; post-synthesis names are not.

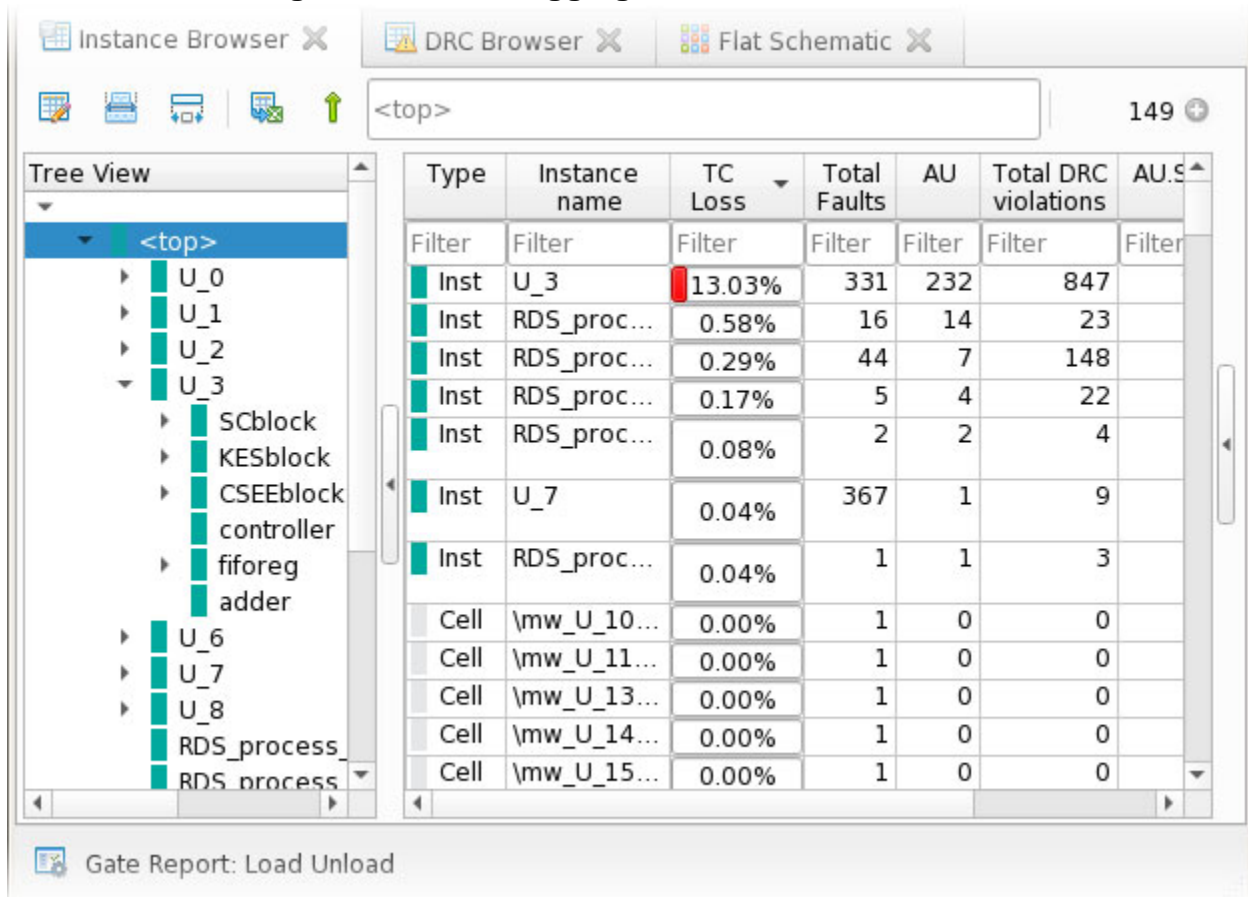
Figure 11-58. Instance Browser Elements



The Child Instances table and Pin/DRC Violations table include toolbars that provide the common controls described in “Tables” on page 631, as well as buttons you can use to switch between pin information and DRC violations in the table. In addition, the Child Instances table has a **Hierarchy-Up** button to navigate to the currently selected instance’s parent.

To debug directly in the instance browser, use the sorting and filtering functions as shown in Figure 11-59 to investigate faults and design rule violations, and to discover instances with problematic coverage statistics. In this example, the “test coverage loss” and “total DRC violations” columns have been added and sorted, and a significant problem can be seen in the U_3 instance.

Figure 11-59. Debugging in the Instance Browser



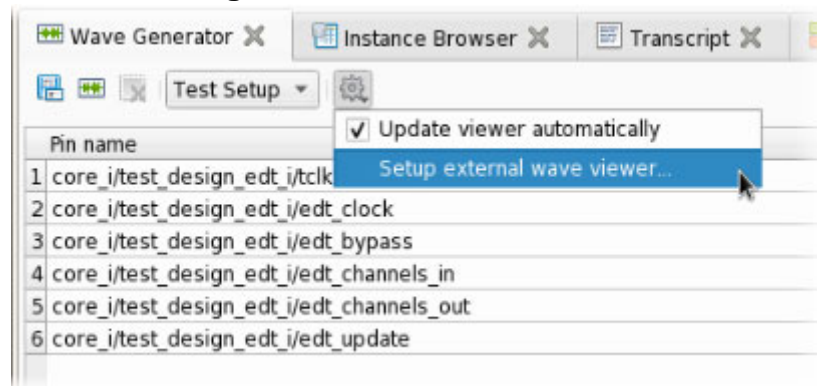
Wave Generator

Use the Wave Generator to export data for selected pins to value change dump (VCD) files or to view their waveforms.

Open the Wave Generator by right-clicking pins on a hierarchical or flat schematic and choosing **Add to Wave Generator**. You can also add pins to the Wave Generator from search tabs, the Instance Browser, or from context tables showing pins or gate pins.

You can also open the Wave Generator by using the [add_wave_generator_pins](#) command or directly from the main window dropdown menu.

Figure 11-60. Wave Generator



Choose the Test Setup or Test End procedure, and export the waveform data to a VCD file or a dofile from the Wave Generator toolbar. If the Test Setup or Test End procedure is not loaded into Tessent Shell, an error message is displayed.

Run an exported dofile with the “dofile” or “source” command to restore the currently displayed pins to the Wave Generator.

Use the Options (gear) menu of the Wave Generator toolbar to set up an external waveform viewer (for example, GTKWave or VisWave).

To control the ordering of the waveforms in the waveform viewer from the Waveform Generator, check the **Update viewer automatically** box. When you drag-and-drop the pin names in the Waveform Generator, the ordering in the waveform viewer changes accordingly. If you delete a pin from the Wave Generator, that pin is also deleted from the waveform viewer.

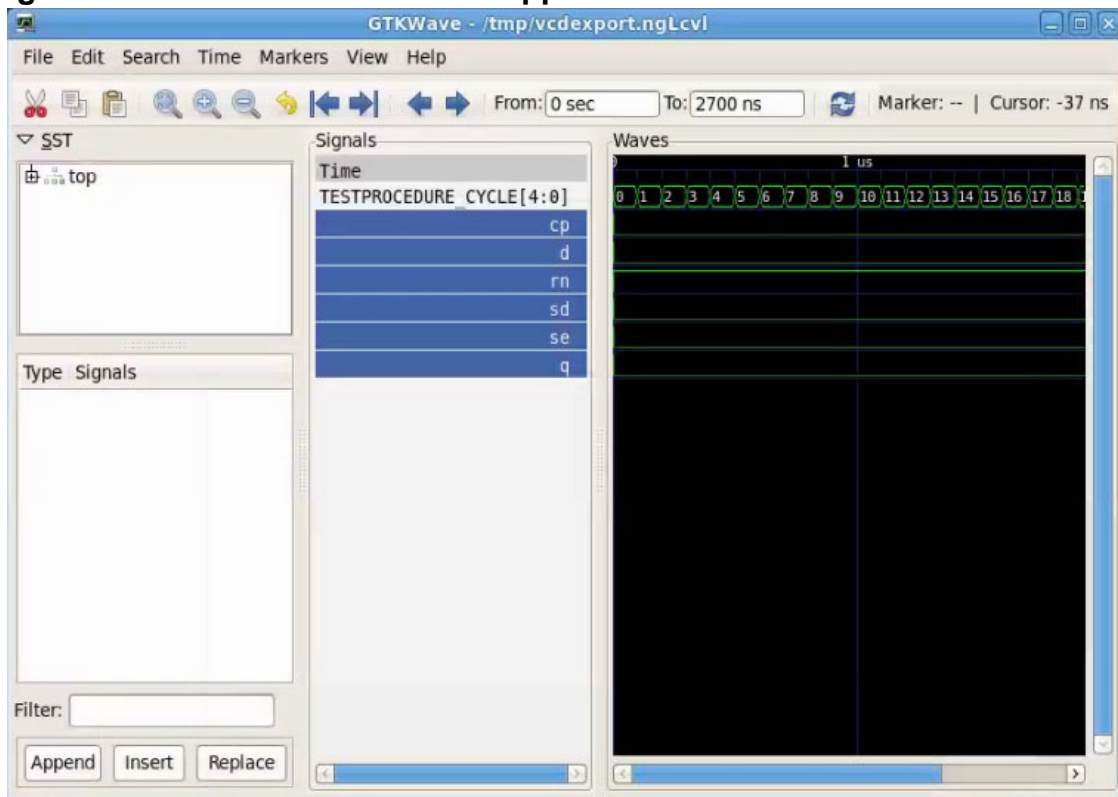
You can right-click a selected set of pin names in the Wave Generator table and choose **Copy names(s)** from the context menu. This action creates a Tcl list in the system clipboard that can be pasted elsewhere, such as into a script or to the Tessent Shell command line.

```
// Copy name(s) example:  
[list {edt_i/edt_scan_in[0]} {edt_i/edt_scan_out[0]} edt_i/edt_clock]
```


You can also use the context menu to:

- Show selected pins in the Hierarchical or Flat Schematic
- Show the HDL definitions or instantiations of selected pins
- Delete selected pins

Figure 11-61. A GTKWave Viewer Application Invoked From Wave Generator



Note

 GTKWave is an open-source third-party application. You should install the GTKWave binary package from a native Red Hat or SUSE Linux distribution repository. You must configure custom GTKWave installations (those built from sources) with the Tcl interpreter enabled to communicate with Tessent Visualizer. You can verify this requirement by issuing the `gtkwave -W` command.

You can also delete pins from the Wave Generator with the `delete_wave_generator_pins` command.

Related Topics

- [add_wave_generator_pins](#)
- [delete_wave_generator_pins](#)

Cell Library Browser

Use the Cell Library Browser to debug test coverage and fault coverage loss from the perspective of library cells.

The Cell Library Browser consists of two panes:

- **Left Pane** — Lists the available cell libraries. This pane can include data such as the module name, statistics including the number of instances in the design, and fault coverage.
- **Right Pane** — Shows the instantiations of those library cells in the design. You can add columns to display data such as the attributes of those instantiations, the hierarchical name, the parent module, and fault information.

Figure 11-62. Cell Library Browser

The screenshot shows the Cell Library Browser interface with two panes. The left pane displays a table of module names and their test coverage statistics. The right pane displays a table of hierarchical names and their AU values.

Module name	Avg Test Coverage	Max Test Coverage
Filter	Filter	Filter
dff	<no data>	<no data>
mux21_ni	45.00%	62.50%
inv02	32.26%	100.00%
sffr_ni	50.00%	50.00%
and02	8.33%	83.33%
xnor2	<no data>	<no data>
tri01	3.92%	66.67%
or02	<no data>	<no data>
buf02	100.00%	100.00%
dffsr	<no data>	<no data>
nor02_2x	<no data>	<no data>
dffr	0.00%	0.00%
nor04	<no data>	<no data>
nand02_2x	0.00%	0.00%
ao21	0.00%	0.00%

Hierarchical name	AU	1
Filter	Filter	F
tap_i/tap_ctrl_i/ reg_pstate_0_	0	
tap_i/tap_ctrl_i/ reg_pstate_2_	0	
tap_i/tap_ctrl_i/ reg_pstate_1_	0	
tap_i/tap_ctrl_i/ reg_pstate_3_	0	
tap_i/tap_ctrl_i/ reg_pstate_3_r...	0	
tap_i/tap_ctrl_i/ reg_pstate_2_r...	0	
tap_i/tap_ctrl_i/ reg_pstate_1_r...	0	
tap_i/tap_ctrl_i/ reg_pstate_0_r...	0	
tap_i/tap_ctrl_i/ reg_pstate_0_r...	0	
tap_i/tap_ctrl_i/ reg_pstate_0_r...	0	
tap_i/tap_ctrl_i/ reg_pstate_1_r	0	

Gate Report: Clock Cone for 'tck'

Click in a row in the left pane to show the instantiation information for that cell type in the right pane. Right-click an instance name in the table in the right pane to get access to all actions available for hierarchical instances.

DRC Browser

The DRC Browser is a tabular reporting tool for exploring the rule violations in your design. Customize the browser to group and filter DRC violations by various criteria.

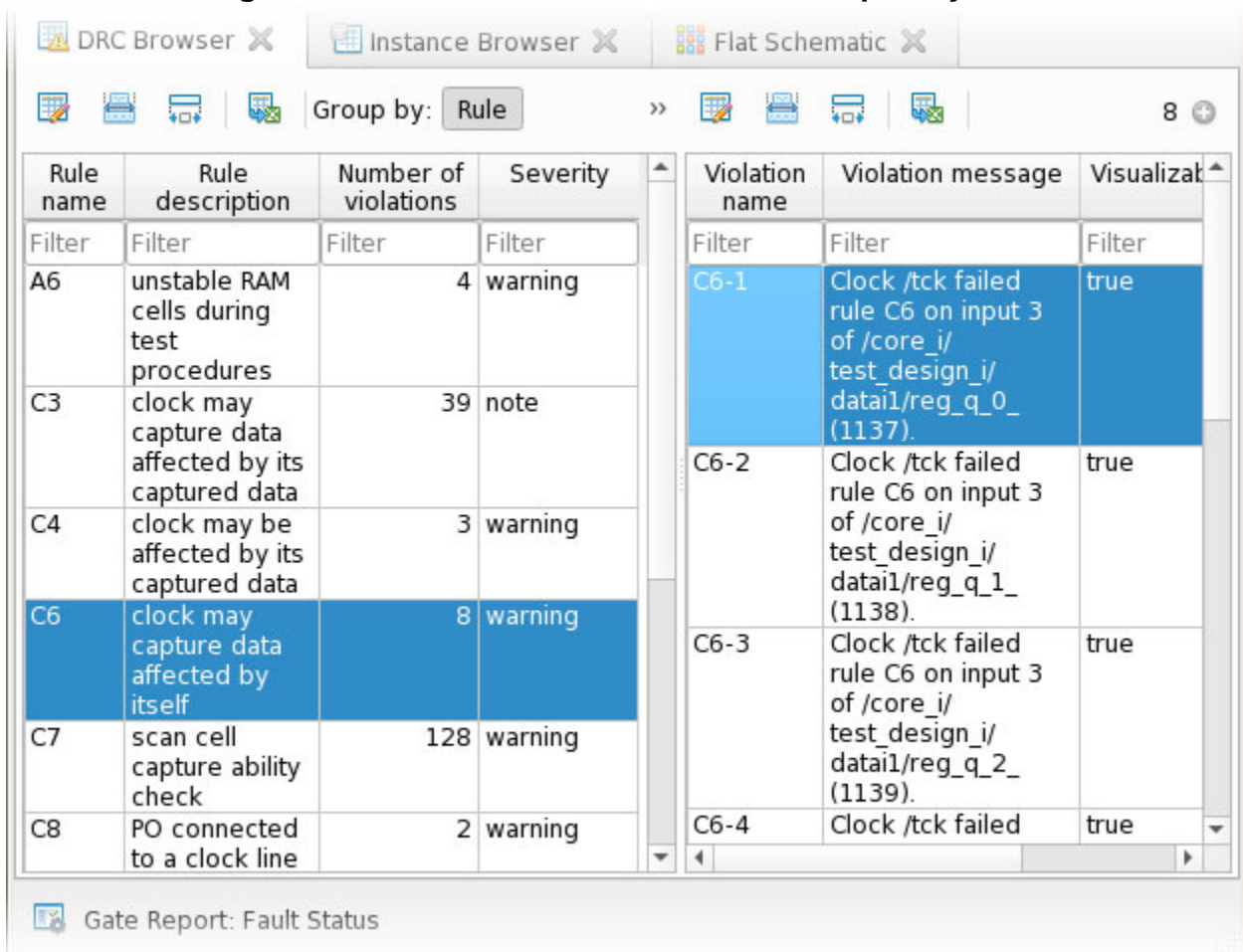
The DRC Browser consists of two panes. The left pane enables you to group the violation list by rule name (the default), instance, or module.

The right pane shows specific information about the violations associated with the rule, instance, or module selected in the left pane, such as:

- Violation name
- Description of the violation
- Indication of whether the violation is visualizable

If nothing is selected in the left grouping table, the right table displays all violations currently registered in Tessent Shell.

Figure 11-63. DRC Browser With Data Grouped By Rule



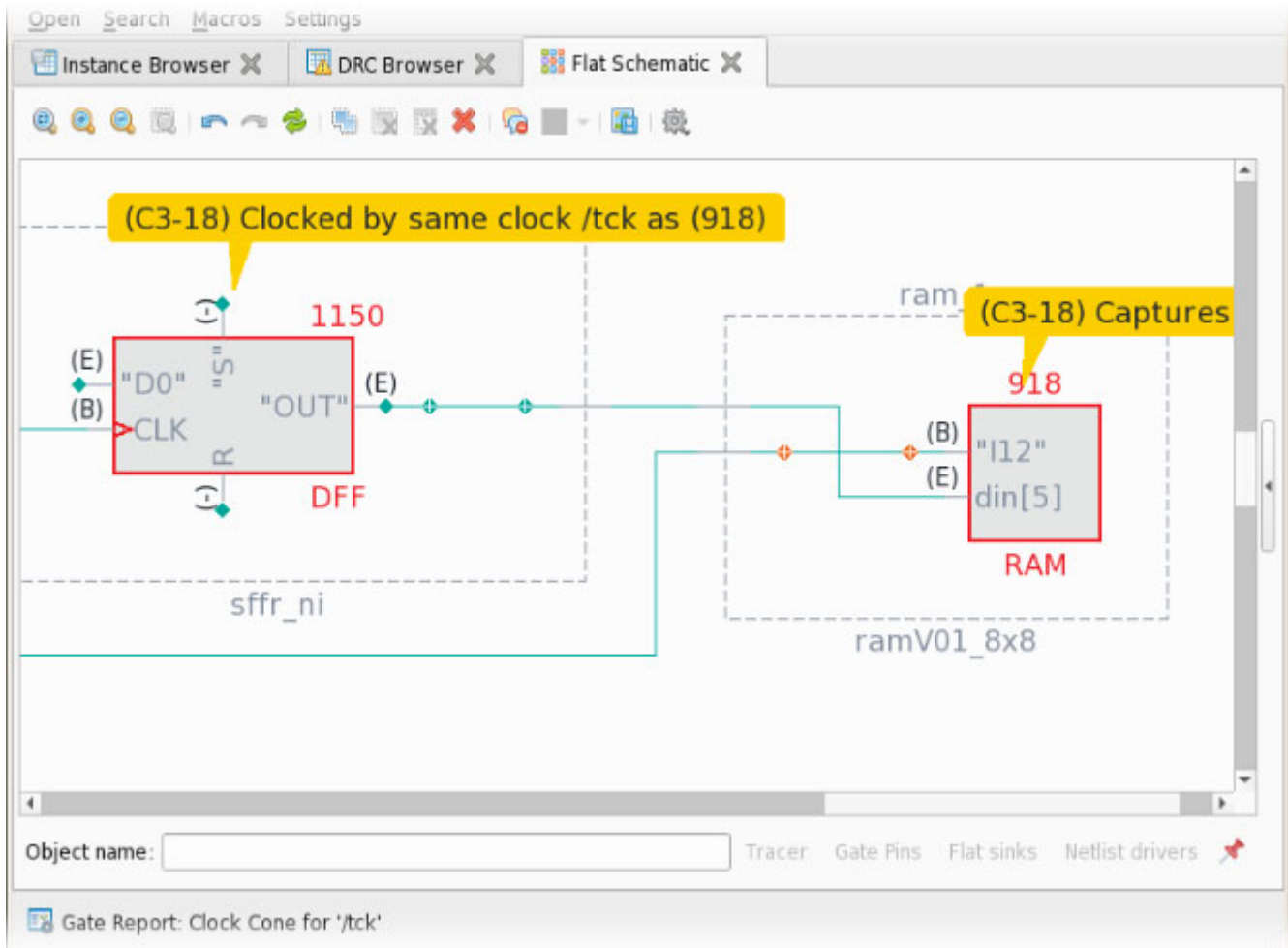
If you group by instance or module, data for the violations in the selected instance or module is displayed.

You can display the specifics for a visualizable rule violation in a schematic or the Text/HDL Viewer by double-clicking a row in the right pane of the DRC Browser, or by right-clicking and choosing the appropriate item from the popup menu. You cannot visualize all DRC violations.

Note

This method is equivalent to issuing the `analyze_drc_violation` command on the Tessent Shell or Transcript command line.

Figure 11-64. Flat Schematic Showing a DRC Violation



Related Topics

[analyze_drc_violation](#)

Pin Data

The **Pin Data** tab contains a tabular report of specified pins in your design. Use this report to visualize gate report information in tabular form.

Add selected pins to Pin Data by right-clicking the pins and choosing **Add to Pin Data**.

Figure 11-65. Pin Data Tab

Pin name	Clock Cone	State Stability
Filter	Filter	Filter
core_i/test_design_edt_i/ edt_bypass	(-)	(1) (1) (111~1) (1) (111) (111111111) (111) (1)
core_i/test_design_edt_i/edt_clock	(-)	(X) (X) (XXX~X) (X) (XXX) (XXXXXXXX) (XXX) (X)
core_i/test_design_edt_i/ edt_channels_in	(-)	(0) (X) (XXX~X) (X) (XXX) (XXXXXXXX) (XXX) (X)
core_i/test_design_edt_i/ edt_channels_out	(E)	(X) (X) (XXX~X) (X) (XXX) (XXXXXXXX) (XXX) (X)
core_i/test_design_edt_i/tclk	(C)	(0) (0) (010~0) (0) (010) (00000000) (010) (0)
core_i/test_design_edt_i/ edt_update	(-)	(X) (X) (XXX~X) (X) (XXX) (XXXXXXXX) (XXX) (X)

Related Topics


[Using Tessent Visualizer to Debug Design Issues](#)

Transcript

The Tessent Visualizer **Transcript** tab provides access to the Tessent Shell command line and a record of commands used and responses to those commands.

Text in the Transcript is highlighted as shown in [Figure 11-66](#):

- **Green** — informational notes
- **Orange** — warning messages
- **Red** — error messages

Figure 11-66. Transcript Tab With an Interactive Command Line

```
// command: report_clocks
User-defined Clocks (2):
=====

Sync and Async Source Clocks
=====
-----
Name      Off State  Internal
-----
'tck'     0          No
'clear'   0          No

// command: display_message -note "This is a note"
// Note: This is a note
// command: display_message -warning "This is a warning"
// Warning: This is a warning
// command: display_message -error "This is an error"
// Error: This is an error
// command: for {set i 0} {$i < 3} {incr i} {
//     puts "Test $i"
// }
Test 0
Test 1
Test 2

for {set i 0} {$i < 3} {incr i} {
    puts "Test $i"
}
ANALYSIS>
```

Gate Report: Normal

You can issue Tessent Shell commands directly from the Transcript, with the same features as the standard Tessent Shell command line. (For example, command history using the up and down arrow keys, command completion using the Tab key, and multi-line commands.) Commands are syntactically highlighted as you type them.

Press Ctrl+Enter to enable multi-line command mode. The background color of the command entry box changes to light blue to indicate that multi-line command mode is active. Use Shift+Enter to insert a new line and the arrow keys to navigate over the multi-line command. Multi-line commands can be recalled with the up-arrow and edited as shown in [Figure 11-66](#)

Text/HDL Viewer

The **Text/HDL Viewer** enables you to examine the HDL description of your design or cell library. The text displayed in this window includes syntax highlighting, and can be copied to the system clipboard for use elsewhere.

Open the Text/HDL Viewer by right-clicking objects in the Instance Browser, Cell Library Browser, or a schematic and choosing **Show HDL definition** or **Show HDL instantiation** from the popup menu. [Figure 11-67](#) shows the Text/HDL Viewer for an object with the appropriate line in the HDL highlighted where the definition or instantiation was found. These actions are available on any hierarchical instance, including library cells, and any table that lists these instances.


Figure 11-67. Text/HDL Viewer



The files opened by the Text/HDL Viewer display on tabs at the bottom of the viewer. To open the current file in an external text editor, right-click the tab and choose **Open in external editor** from the popup menu. Use the [Tessent Visualizer Preferences](#) dialog box to specify the external text editor.

To copy the file path of the current file to the system clipboard, right-click the tab and choose **Copy file path** from the popup menu.

Note

 When Tessent Shell is in the “dft -rtl” context, you can select a text object in the viewer and display it in the Hierarchical Schematic. To do so, select a text fragment, right-click, and choose **Show on Hierarchical Schematic**.

Diagnosis Report Viewer

Use the Diagnosis Report Viewer to open reports created with the `write_diagnosis` command.

Open the Diagnosis Report Viewer by choosing the **Open > Diagnosis Report Viewer** menu item. Then you can open a diagnosis report using the toolbar at the top of the Diagnosis Report Viewer or by issuing the `display_diagnosis_report` command. You can open multiple reports using either of these methods. Select either text view or table view mode from the same toolbar.

In text view mode ([Figure 11-68](#)), the diagnosis report opens in the window, annotated with hypertext links for each design object affected. These links refer to the following:

- Symptoms
- Suspects
- Sub-suspects
- Pin and net objects associated with suspects

You can click a link to show the indicated object in the Hierarchical Schematic. Right-click to display a popup menu for other actions available to examine these objects.

Figure 11-68. Diagnosis Report Viewer (Text View)

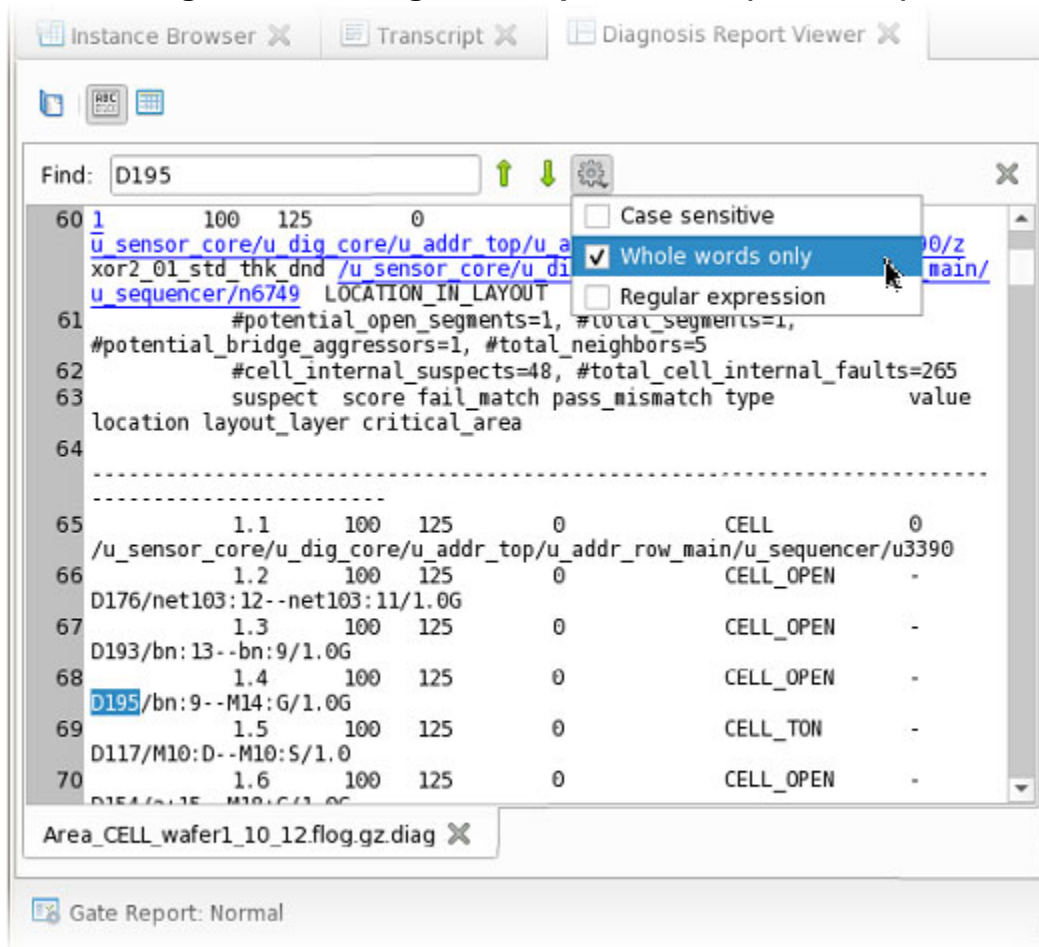


Figure 11-68 also illustrates text searching in the Diagnosis Report Viewer.

Figure 11-69 shows the three panes of the tabular layout for the diagnosis report. The left pane is a list of the symptoms denoted with integer IDs. Select a symptom in this pane to display a list of suspects in the right top table. Double-click an object in the table, or right-click and choose **Show on Hierarchical Schematic**, to show it in the Hierarchical Schematic. If a suspect has sub-suspects, the bottom table displays a list of sub-suspects.

Figure 11-69. Diagnosis Report Viewer (Table View)

The screenshot shows the 'Diagnosis Report Viewer' window with two tables. The top table is filtered to show 3 items. The bottom table is filtered to show 51 items.

Symptom id	Suspect id	Score	Type	Fail match
Filter	Filter	Filter	Filter	Filter
1	1	100	INDETERMINATE	12!
1	2	100	INDETERMINATE	12!
1	3	81	INDETERMINATE	12!

Symptom id	Suspect id	Physical id	Score	Type
Filter	Filter	Filter	Filter	Filter
1	1	1	100	CELL
1	1	2	100	CELL_OPEN
1	1	3	100	CELL_OPEN
1	1	4	100	CELL_OPEN

Area_CELL_wafer1_10_12.flog.gz.diag

Gate Report: Normal

Search Features

From the main menu bar, open the **Search** menu to search for instances, pins, or gate pins in your design.

Figure 11-70 shows a search for all instances in the design with module names that begin with the string “nor”.

Figure 11-70. Search Tab: Instances

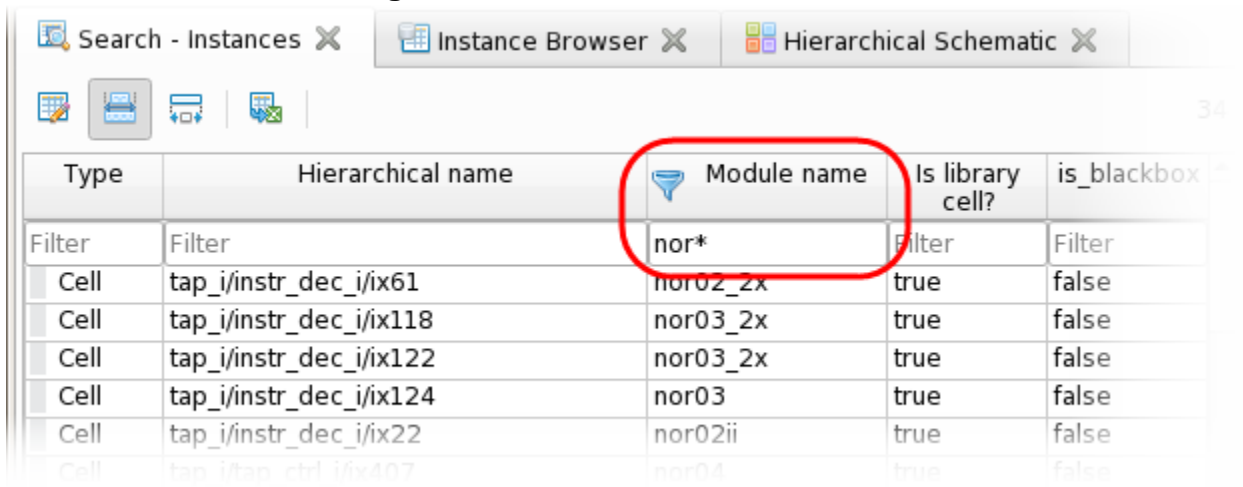


Figure 11-71 shows a search for all hierarchical pins with the string “mode” in the pin name.

Note

The search for pins in a large design can be time consuming because the tool searches across all hierarchical instances and their pins.

Figure 11-71. Search Tab: Pins

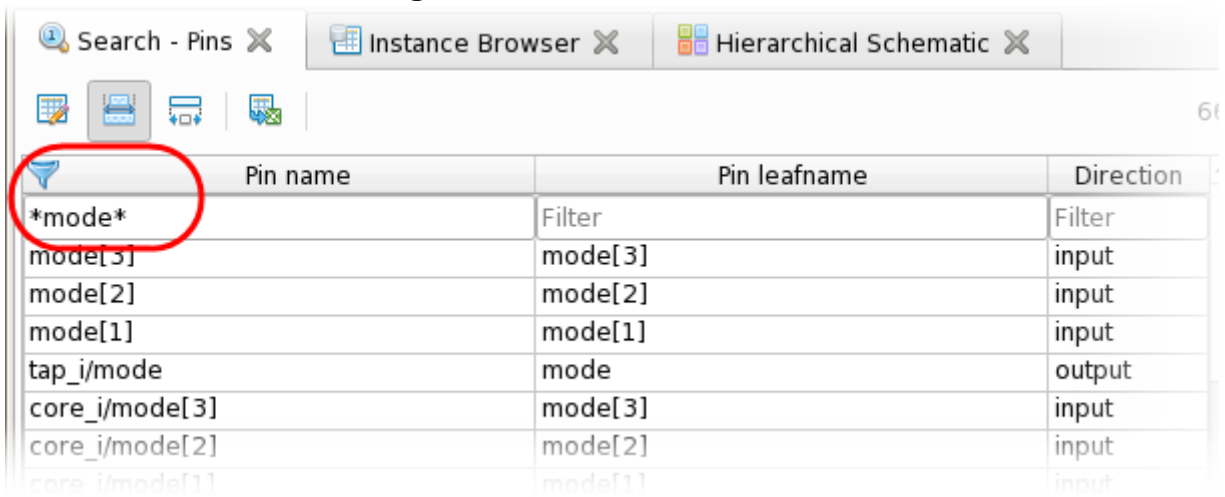


Figure 11-72 shows a search for flat pins (gate pins) that have a hierarchical pin with a leaf name of “Q” below the bsr_i1 instance.

Figure 11-72. Search Tab: Gate Pins

Gate id	Gate pin ID	Pin pathname	Direction
Filter	Filter	bsr_i1/*/Q	Filter
341	341.0	bsr_i1/bsc_dosync/reg_latch_out/Q	output
343	343.0	bsr_i1/bsc_dosync/reg_serial_output/Q	output
345	345.0	bsr_i1/bsc_v_dout_7/reg_latch_out/Q	output
347	347.0	bsr_i1/bsc_v_dout_7/reg_serial_output/Q	output
349	349.0	bsr_i1/bsc_v_dout_6/reg_latch_out/Q	output
351	351.0	bsr_i1/bsc_v_dout_6/reg_serial_output/Q	output
353	353.0	bsr_i1/bsc_v_dout_5/reg_latch_out/Q	output

You can select one or more rows in a filtered search table. Then, use the right-click popup menu to show those objects in a schematic, add them to the pin data (pin searches), or show the HDL definition (instance searches).

Note

The display properties for Search Instances and Search Pins are based on the Hierarchical Schematic. The display properties for Search Gate Pins are based on the Flat Schematic



See “[Tessent Visualizer Components and Preferences](#)” on page 630 for information about interacting with these tables in Tessent Visualizer.

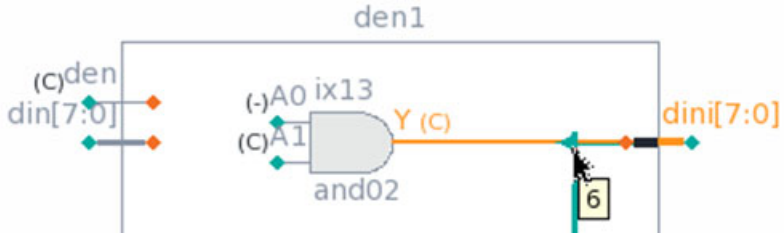
Using Tessent Visualizer to Debug Design Issues

You can use Tessent Visualizer to solve various design problems. You can view state elements or observation points from a pin fanout, add or hide pins on an instance in the hierarchical schematic, search for pins by name, trace the bits of a bus, and debug issues by viewing simulation waveforms on pins.

Procedure

Perform any of the following actions:

If you want to...	Do the following:
List all state elements in the fanout of a pin.	<ol style="list-style-type: none"> 1. Click a pin in a schematic. The Tracer table displays below the schematic. 2. Click the “Strategy:” dropdown list and select “Nearest state element.” <p>The Tracer table lists all the state elements and the distance from the source in terms of number of pins in the traced path. The pin count for the same traced path is higher in the Hierarchical Schematic than in the Flat Schematic because of additional pins on hierarchical boundaries.</p>
Add pins to an instance on a hierarchical schematic.	<ol style="list-style-type: none"> 1. Select an instance on a Hierarchical Schematic, then click the Pins button. A table listing the pins on the selected instance displays. 2. Click the Pins button below the schematic to display a table listing the pins on the selected instance. 3. Use the filter feature to search for the desired pin. 4. Add the pin to the schematic by double-clicking or by dragging it onto the schematic.
Search for pins by name.	<ol style="list-style-type: none"> 1. Choose the Search > Search - Pins menu item. 2. Filter or sort the list of pins as needed. 3. Right-click the pin name of interest in the table and choose one of these menu items to display the pin in a schematic: <ul style="list-style-type: none"> • Show on Hierarchical Schematic • Show on Flat Schematic
Debug test setup or test end issues using a waveform viewer.	<ol style="list-style-type: none"> 1. Select one or more pins on a schematic. 2. Right-click the selected pins and choose Add to Pin Data from the popup menu. This displays the Pin Data tab, which lists all of the added pin names. 3. Select one or more pin names listed in the table. 4. Click the Options button () in the toolbar and select which VCD exportable procedure to include as a waveform in the viewer: Test Setup or Test End. 5. Launch the specified waveform viewer by clicking the external waveform viewer button ()

If you want to...	Do the following:
Trace a single bit of a bus.	<p>When tracing a single bit of a bus, the bit remains split from the bus. When tracing a net that recombines into a bus, the bit indices display in the schematic when you hover the mouse pointer, over the split point as shown in the following figure:</p> 

Results

Listing all state elements in the fanout of a pin: the Tracer table lists all the state elements and the distance from the source in terms of number of pins in the traced path. The pin count for the same traced path is higher in the Hierarchical Schematic than in the Flat Schematic because of additional pins on hierarchical boundaries.

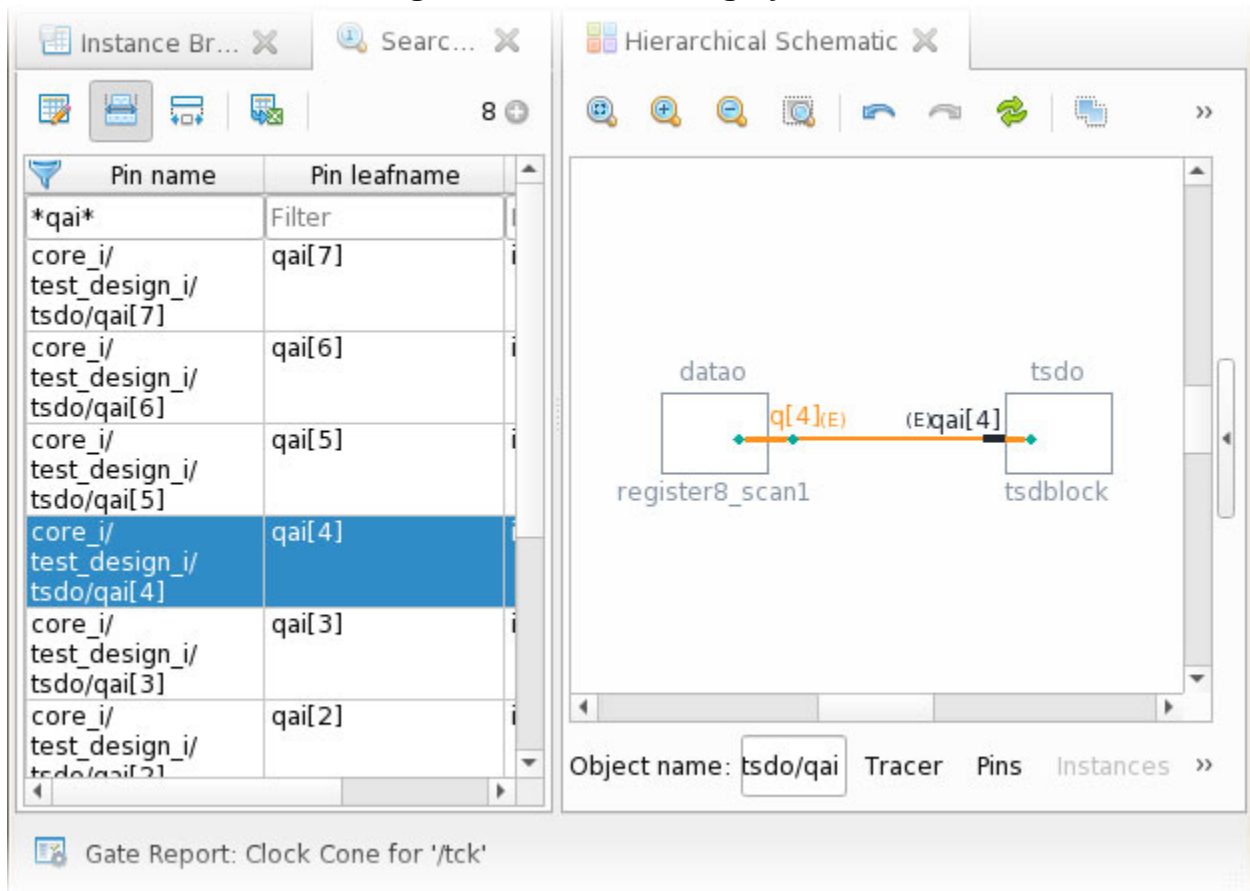
Adding pins to an instance in a hierarchical schematic: when you add an instance to a schematic directly or by tracing to it, the schematic displays only the pins that have connections (except for cases where there are very few pins in total). This feature simplifies the schematic to focus on pins of interest, improving run time.

Examples

Searching for Pins by Name

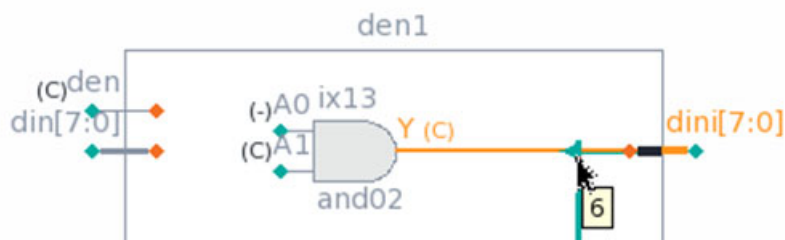
The following figure shows a search for the “qai” bus, and subsequent selection of an element of that bus, which is displayed in the hierarchical schematic. You can then trace the signal.

Figure 11-73. Searching by Name



Tracing a Single Bit of a Bus

When tracing a single bit of a bus, the bit remains split from the bus. When tracing a net that recombines into a bus, the bit indices display in the schematic when you hover the mouse pointer, over the split point as shown in the following figure:



Related Topics

- [Nearest State Element Tracing Strategy](#)
- [Wave Generator](#)

Accessing Tutorials for Tessent Visualizer

Tutorial instructions and design data are available for download in an Example Kit (eKit) file. The tutorials demonstrate some basic operations you can perform with Tessent Visualizer. The eKit is a ZIP file containing the instructions and design data that you need to perform the Tessent Visualizer tutorials.

Procedure

1. Log on to Support Center:
<https://support.sw.siemens.com>
2. Find the Tessent product page.
3. Click the Documentation link in the page banner.
4. Use the “Restrict content to version” dropdown list to specify the current Tessent version.
5. Click the Getting Started Guide in Document Types on the left side of the page.
6. Click the link for “Try It: Tessent Visualizer Quick Start and Example Kit.”
This downloads the eKit ZIP file to your default location.
7. Move the downloaded file to a working directory that you can access with a Linux shell.
8. From a Linux shell, unpack the download file:

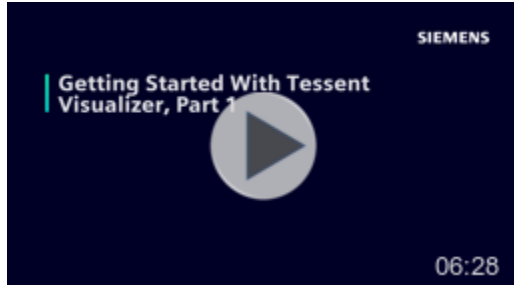
```
$ unzip <path>/tesvis_qs_ekit.zip
```
9. Navigate to the *tesvis_qs_ekit/Docs* directory and open the *tesvis_quickstart.pdf* file.
This file contains instructions for performing the tutorials.

Results

You are now ready to perform the Tessent Visualizer tutorials.

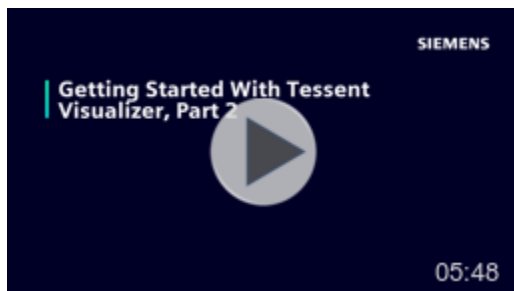
Accessing Videos for Tessent Visualizer

Getting Started with Tessent Visualizer is a set of two videos that show some basic operations you can perform with Tessent Visualizer.



Part 1: Analyzing and Improving Test Coverage demonstrates how to analyze test coverage and explore areas of the design that need coverage improvement:

- Invoking Tessent Visualizer
- Using the Instance Browser to navigate a design
- Adding data columns to a table
- Sorting and filtering data
- Examining DRC violations and locating a violation on a schematic
- Using the Cell Library Browser to examine a library module



Part 2: Troubleshooting Problems With Verilog Design Files shows how to troubleshoot a problem with a Verilog design file, which includes the following operations:

- Examining the source of a DRC violation on a schematic
- Troubleshooting an error with scan chain insertion
- Tracing a signal path on a schematic
- Recognizing problems on a schematic and locating the problem source in HDL code
- Determining the number of times a library module is instantiated in a design

Chapter 12

Configuration Data Visualizer

The Configuration Data Visualizer enables you to view specifications and to modify the configuration tree elements and their properties using a graphical interface.

Modifying the Contents of the Configuration Data Visualizer	707
Adding a Test Data Register to a SIB Example	708
Adding a Multiplexer to a SIB Example	708
Customizing the DFT Specification for EDT	710

Modifying the Contents of the Configuration Data Visualizer

View and modify specifications with the Configuration Data Visualizer.

Prerequisites

- Tessent Shell is invoked.


Procedure



1. Open the Configuration Data Visualizer by issuing the [display_specification](#) command to view and modify a DftSpecification. The specification can be one that was read by using the [read_config_data](#) command, or created with the “display_specification -create” command.

The Configuration Data Visualizer window opens and displays any existing configuration data.

2. Select an element from the Configuration Tree and choose an **Add** > menu item from the right-click popup menu.

The Add submenu displays a context-sensitive list of elements that shows the types of elements you can add based on the element you selected.

3. Specify the parameters for the newly added element in the Configuration Options panel:
 - Enter the values of the element’s listed properties. Predefined values appear when click you the  icon.
 - Click **Interface[+]** to define parameters of the element’s interface.
 - Click **DataInPorts[+]** and **DataOutPorts[+]** to define the number of ports on the element, their names, and their connections.

- Click **Attributes[+]** to define arbitrary attributes on the element's ICL module.
4. Click **Apply** to update the specification.
 5. When you have created and fully defined all of the elements that you want to add, click the  icon to validate the current specification and look for errors. You can also issue the “process_dft_specification -validate_only” command, which is an equivalent action.
 6. After you have validated the specification and resolved all errors, click the  icon to process the specification. You can also issue the process_dft_specification command, which is an equivalent action.

Results

New elements have been created and defined new in your specification, and those changes have been validated. The specified elements have been created by processing the specification.



Adding a Test Data Register to a SIB Example

Add a Test Data Register (TDR) to a SIB with the Configuration Data Visualizer.

Prerequisites

- Tessent Shell is invoked.

Procedure

1. Open the Configuration Data Visualizer from the Tessent Shell command line.
2. Select the SIB to which you want to add the TDR.
3. Right-click and choose **Add > Tdr**.
Change its position using the **Move Up** and **Move Down** menu items.
4. In the Configuration Options panel, enter a name in the ID text box. Type appropriate values in any of the other displayed fields.
5. Click  and  and modify the value of the “count” property in each to change the number of ports on the TDR.
6. Click **Apply** to update the specification.

Results

A TDR with the appropriate number of ports has been added to a SIB.

Adding a Multiplexer to a SIB Example

Add a ScanMux to a SIB with the Configuration Data Visualizer.



Prerequisites

- Tessent Shell is invoked.

Procedure

1. Open the Configuration Data Visualizer from the Tessent Shell command line.
2. Add a ScanMux by selecting the SIB to which you want to add it, then right-click and choose **Add > ScanMux**.

Change its position using the **Move Up** and **Move Down** menu items.

- a. Name the ScanMux by typing a name in the id field of the Configuration Options panel. Specify the desired value of any of the other displayed fields.
 - b. Define the ScanMux connections by clicking **Interface[+]** and typing the connection for the MUX select line. If a TDR is defined, you can use any TDR data output port to control the mux select signal.
 - c. Click **Apply** to update.
3. Add the first input port to the new ScanMux by right-clicking it and choosing **Add > Input**.
 - a. Select the new input port and type a number in the int text box of the Configuration Options panel.
 - b. Click **Apply** to update.
 - c. Add a TDR that is accessed when this mux input port is selected by right-clicking the Input port and choosing **Add > Tdr**.
 - i. Name the TDR by selecting it and typing a name in the ID field of the Configuration Options panel.
 - ii. Connect the TDR to the design by doing the following:
 - a. Click the **DataInPorts[+]** button, click the **Connections[+]** button, and then click the  icon. Type the range that matches the design instance data bus size into the range field (for example, "7:0"), and type the pathname to the design instance in the pin_name field.
 - b. Click the **DataOutPorts[+]** button, click the **Connections[+]** button, and then click the  icon. Type the range that matches the design instance data bus size into the range field, and type the pathname to the design instance in the pin_name field.
4. Define the connections for the second input port of the ScanMux using the instructions in Step 3.

Results

A ScanMux has been created and the select signal has been defined. Two Input wrappers with TDRs defined have been added to the ScanMux.

Customizing the DFT Specification for EDT

In the pre-scan DFT insertion flow, you create a DFT specification for inserting the EDT hardware. You may need to customize the DFT specification that Tessent Shell generates to suit your design requirements. Edit the DFT specification with the Configuration Data Visualizer.

Prerequisites

- You have created a DFT specification for inserting EDT. This procedure assumes that you specified the `create_dft_specification` command with the `-sri_sib_list` option.
- Tessent Shell is invoked.

Procedure

1. Open a DFT specification in the Configuration Data Visualizer by invoking the `display_specification` command.
2. In the Configuration Tree pane, right-click the `DftSpecification` wrapper and then choose **Add > EDT**.
3. In the `ijtag_host_interface` text box, type **Sib(edt)**, and then click **Apply**.
4. Right-click the EDT wrapper and choose **Add > Controller**.
5. Select the Controller wrapper, and then type the name of the controller in the `id` text box in the **Configuration Options** pane.
6. In the **Configuration Options** pane, type values in the following text boxes to configure how you want Tessent Shell to create the EDT controller hardware, and then click **Apply**.
 - a. **longest_chain_range** (two number boxes)
 - b. **scan_chain_count**
 - c. **input_channel_count**
 - d. **output_channel_count**
7. If you are inserting EDT at the chip level (`design_level chip`), do the following:
 - a. Click the plus sign next to the Controller wrapper, right-click the **Connections** wrapper, and then choose **Add > EdtChannelsIn** and **Add > EdtChannelsOut** as many times as required for the EDT controller.

- b. Within the Connection wrapper, select one of the input or output channels you created, and type the pin name for the selected channel in the port_pin_name text box in the **Configuration Options** pane. Click **Apply**.
 - c. Repeat the previous step to name all the input and output channels you created.
8. (Optional) View the edited DFT specification by invoking the report_config_data command, and then invoking the read_config_data command to add the edited DFT specification to a dofile for later reuse in an automated flow.

Results

To view the modified DFT specification, specify the report_config_data command. For example:

```
report_config_data $spec
```

```
DftSpecification(cpu_top,rtl2) +{
  IjtagNetwork +{
    HostScanInterface(sri) +{
      Interface {
        design_instance : cpu_top_rtl_tessent_sib_sti_inst;
        scan_interface : client;
      }
    }
    Sib(sri) +{
      Attributes {
        tessent_dft_function : scan_resource_instrument_host;
      }
    }
    Tdr(sri_ctrl) {
      Attributes {
        tessent_dft_function : scan_resource_instrument_dft_control;
      }
    }
    Sib(occ) {
    }
    Sib(edt) {
    }
  }
}
EDT +{
  ijtag_host_interface : Sib(edt);
  Controller(c1) +{
    longest_chain_range : 65, 100;
    scan_chain_count : 85;
    input_channel_count : 3;
    output_channel_count : 3;
    Connections +{
      EdtChannelsIn(1) {
        port_pin_name : edt_channels_in[0];
      }
      EdtChannelsIn(2) {
        port_pin_name : edt_channels_in[1];
      }
      EdtChannelsOut(1) {
        port_pin_name : edt_channels_out[0];
      }
      EdtChannelsIn(3) {
        port_pin_name : edt_channels_in[2];
      }
      EdtChannelsOut(2) {
        port_pin_name : edt_channels_out[1];
      }
      EdtChannelsOut(3) {
        port_pin_name : edt_channels_out[2];
      }
    }
  }
}
```


Examples

Use the `read_config_data` command to save the modifications to the DFT specification into a dofile for re-use.

```
read_config_data -in $spec -from_string {
  EDT {
    ijtag_host_interface : Sib(edt);
    Controller (c1) {
      longest_chain_range : 65, 100;
      scan_chain_count : 85;
      input_channel_count : 3;
      output_channel_count : 3;
      Connections +{
        EdtChannelsIn(1) {
          port_pin_name : edt_channels_in[1];
        }
        EdtChannelsOut(1) {
          port_pin_name : edt_channels_out[0];
        }
        EdtChannelsIn(3) {
          port_pin_name : edt_channels_in[2];
        }
        EdtChannelsOut(2) {
          port_pin_name : edt_channels_out[1];
        }
        EdtChannelsOut(3) {
          port_pin_name : edt_channels_out[2];
        }
      }
    }
  }
}
```


Chapter 13

Timing Constraints (SDC)

This chapter describes the Synopsys Design Constraints (SDC) generated by Tessent Shell, and how you use it to constrain the test mode of the Embedded Test hardware generated by the tool.

Generating and Using SDC for Tessent Shell Embedded Test IP	716
SDC File Generation with Tessent Shell	716
SDC Design Synthesis with Tessent Shell	718
Preparation Step 1: Sourcing SDC File	718
Preparation Step 2: Setting and Redefining Tessent Tcl Variables	718
Preparation Step 3: Verifying the Declaration of Functional Clocks	720
Preparation Step 4: Redefining Other Tessent Tcl Variables	721
Synthesis Step 1: Applying the SDC Constraints	722
Synthesis Step 2: Preparing the DFT Logic for Synthesis	722
Synthesis Step 3: Synthesizing Your Design	723
Synthesis Step 4: Writing Out Your Final SDC	723
Synthesis Step 5: Writing Out Your Final Netlist	723
Running Layout with Tessent Shell DFT	723
SDC for Modal Static Timing Analysis	726
Checking Your Functional Logic Alone	726
Checking Your Embedded Test Logic	726
VHDL and Mixed Language Designs	728
VHDL Generate Blocks	728
SDC File Contents	730
tessent_set_default_variables	730
tessent_create_functional_clocks	731
tessent_<design_name>_set_dft_signals	731
<lttest_prefix>_disable	732
tessent_set_non_modal	732
tessent_<design_name>_kill_functional_paths	733
IJTAG Instrument	734
LOGICTEST Instruments	735
MemoryBIST Instrument	748
BoundaryScan Instrument	751
Hierarchical STA in Tessent	752
Mapping Procs	756
Synthesis Helper Procs	758
Example Scripts using Tessent Tool-Generated SDC	760
Example Design Compiler Synthesis Script	760

Example Genus Synthesis Script	761
Example PrimeTime STA Script	764

Generating and Using SDC for Tessent Shell Embedded Test IP

We guide you through an example SDC flow, from generation of the Tessent SDC file, through adjusting the variables used within the file, to using it in your design synthesis.

First, you learn how to generate an SDC file for your design, which may be at the chip or the physical block level. SDCs for sub blocks are automatically merged by Tessent with the constraints of their parent physical block.

Then we show you how to adjust variables in the SDC file to match your particular synthesis setup or requirements. There is no need however, to ever directly edit the generated SDC files.

Next, you see how to use the generated SDC file in a step-by-step example synthesis flow, followed by additional notes on Static Timing Analysis (STA), mixed-language and VHDL specific issues.

Key SDC procs are then explained. These procedures are used by Tessent instruments (like MBIST or OCC) or maybe useful to use in your own synthesis scripts.

The appendices show full examples of using and adjusting Tessent SDC files in the Synopsys Design Compiler and Cadence Encounter synthesis environments, respectively. Also a complete example for using Tessent SDC files for STA is given in “[Example PrimeTime STA Script](#)” on page 764.

Note: The SDC described in this manual is to be used in conjunction with your functional SDC for the synthesis of your design after Tessent Shell RTL insertion and the static timing analysis of your design after synthesis. The Tessent Shell gate insertion flow uses a different set of SDC constraints during the [run_synthesis](#) command, but the generated SDC should still be used for layout.

SDC File Generation with Tessent Shell

You generate your design SDC file using Tessent Shell.

In the standard Tessent Shell flow, the tool creates the SDC for the current design when you execute the [extract_icl](#) command, which does the following:

- Stores the SDC in Tcl procs
- Writes the SDC to a single file named *design_name.sdc*, where *design_name* is the name of the current design loaded in Tessent Shell

The SDC file is located next to the extracted ICL, which is typically in the [dft_inserted_designs](#) directory as follows:

```
${tsdb}/dft_inserted_designs/${design_name}_${design_id}.dft_inserted_design
```

Every instrument type (for example, MBIST, IJTAG) of the current design and all sub-blocks that provide SDC constraints are represented by a separate proc in the SDC file. Constraints for logictest-related instruments such as OCC, EDT, or LBIST, including logictest-related DFT signals, are all grouped under similar placeholder “ltest” instrument procs.

There is no need to call each instrument’s individual SDC proc. Tessent Shell provides user procs, which are customized to call all relevant SDC procs for your design.

If you encounter a problem with SDC generation preventing ICL extraction, you can temporarily skip SDC generation by using the following command options:

```
extract_icl -skip_sdc_extraction
```

To regenerate your SDC file for a given design and to take advantage of changes in the SDC constraints in a newer version that does not reflect a hardware change, you can extract the SDC of your design without running ICL extraction by loading the design’s full view and executing the Tessent Shell [extract_sdc](#) command. You should load the interface view of all physical blocks of your design. For example:

```
read_design design_name -design_identifier design_id -view full  
extract_sdc
```

When it is time to use the SDC, locating your SDC file requires the following:

- Your latest TSDB location for the design you wish to synthesize/analyze.
- Your design’s name.
- The *design_id* of the latest insertion pass of your design.

For example, if your design “myChip” used only a single TSDB directory, the default “tsdb_outdir”, then your SDC files would be located in the following places:


- For the processed DftSpecification(myChip,rtl1):

```
tsdb_outdir/dft_inserted_designs/myChip_rtl1.dft_inserted_design/myChip.sdc
```

- For the processed DftSpecification(myChip,rtl2):

```
tsdb_outdir/dft_inserted_designs/myChip_rtl2.dft_inserted_design/myChip.sdc
```

Note

 The latest SDC file is always a superset of the previous one. If you use a two-pass flow, you only need to load the latest file.

SDC Design Synthesis with Tessent Shell

What follows is a list of steps you must follow during your synthesis flow to properly handle Tessent Embedded Test IP. We use a non-modal approach to constrain the inserted test logic. This means that we let both functional and test clocks through muxes and we do not set any constant anywhere. This lets the synthesis tools optimize both the functional and test timing paths simultaneously.

The following steps are run after you have loaded the functional design and applied functional SDC. These steps do not describe the entire flow, but, rather, only highlight typical steps affected by the presence of Tessent IP. Design Compiler commands are used as an example: see “[Example Design Compiler Synthesis Script](#)” on page 760 and “[Example Genus Synthesis Script](#)” on page 761 for complete example scripts.

Preparation Step 1: Sourcing SDC File	718
Preparation Step 2: Setting and Redefining Tessent Tcl Variables	718
Preparation Step 3: Verifying the Declaration of Functional Clocks	720
Preparation Step 4: Redefining Other Tessent Tcl Variables	721
Synthesis Step 1: Applying the SDC Constraints	722
Synthesis Step 2: Preparing the DFT Logic for Synthesis	722
Synthesis Step 3: Synthesizing Your Design	723
Synthesis Step 4: Writing Out Your Final SDC	723
Synthesis Step 5: Writing Out Your Final Netlist	723

Preparation Step 1: Sourcing SDC File

The first step is to source the Tessent Shell tool-generated SDC file, associated to the last insertion pass, from within your synthesis script. We suggest creating central variables for the TSDB location and design name:

```
set design_name myChip
set tsdb ../../../../golden_repo/${design_name}_tsdb
set design_id rtl2
source \
${tsdb}/dft_inserted_designs/${design_name}_${design_id}.dft_inserted_design \
/${design_name}.sdc
```

Preparation Step 2: Setting and Redefining Tessent Tcl Variables

The Tessent Shell tool-generated SDC file makes extensive use of Tcl variables. This section discusses the redefinition of those variables


Their definitions are contained in the proc `tessent_set_default_variables`. You must call this procedure to set all Tessent variables to their design dependent default value. All other Tessent SDC-related procs make use of these variables. This procedure call is mandatory:

```
tessent_set_default_variables
```

Tessent SDC variables can and should be redefined to better suit your setup and design. For example, you can change the TCK period from the default of 100.0ns to any value by redefining the variable `tessent_tck_period`.

```
set tessent_tck_period 80.0
```

Note

 The value of `tessent_tck_period` might depend on the maximum tck clock frequency that can be applied to the circuit. See the “[IJTAG Network Performance Optimization](#)” section in the *Tessent IJTAG User’s Manual* showing how to maximize the frequency of the IJTAG network test clock.

You do not need to edit the Tessent Shell-generated SDC file, but rather use the “set” command in your synthesis script to overwrite the value of the Tessent Shell tool-specific SDC variable `tessent_tck_period`. Refer to “[Example Design Compiler Synthesis Script](#)” on page 760 and “[Example Genus Synthesis Script](#)” on page 761 for synthesis script examples.

For designs with Siemens EDA Logictest IP, you possibly need to update the global Tcl variables that reproduce your fastscan test_proc timeplate specifications, so that your SDC constraints closely match your simulation waveforms. For more information on these Tcl variables, see “[LOGICTEST Instruments](#)” on page 735.

When the design contains LogicBIST instruments, create the `shift_clock_src` of the LogicBIST controller and indicate its name to the SDC procs by using the `tessent_lbist_shift_clock_src` TCL array variable. Creating this clock and setting its name with the `tessent_lbist_shift_clock_src` variable is mandatory.

Specify this variable as follows:

```
set tessent_lbist_shift_clock_src(lbist_inst<index>) clock_name
```

For example:

```
create_clock -name lbist_clock pll/clk_out_2 -period 20  
set tessent_lbist_shift_clock_src(lbist_inst0) lbist_clock
```

If multiple controllers use the same clock, create the clock once only. However, you must still specify the `tessent_lbist_shift_clock_src` variable for each controller with the same clock name. Starting at 0, specify as many `lbist_inst` indices as the number of LogicBIST controllers in the current design level minus one. For example, for a design with two LogicBIST controllers, set the `tessent_lbist_shift_clock_src(lbist_inst0)` and `tessent_lbist_shift_clock_src(lbist_inst1)` variables.

You can find the mapping of the `lbist_inst<index>` identifier to the instance path name of the LogicBIST controllers in the `tessent_lbist_mapping` TCL array variable inside the `tessent_set_default_variables` SDC proc. When the clock is created with a different name than its source, specify the name of the clock as specified with `-name` option of the `create_clock` SDC command. Existence of this TCL variable and the clock it refers to is rule checked by the generated SDC.

Similarly, when the design contains InSystemTest instruments, indicate the clocks to SDC by using the `tessent_ist_clock` TCL array variable. You can find the mapping from the `ist_inst<index>` identifier to the InSystemTest controller instance path name in the `tessent_ist_mapping` variable.

Another key variable to possibly overwrite is the `tessent_clock_mapping` array explained in the next section. Other, much less frequently used variables are discussed in “[Preparation Step 4: Redefining Other Tessent Tcl Variables](#)” on page 721.

Preparation Step 3: Verifying the Declaration of Functional Clocks


In the Tessent Shell tool’s system mode ‘setup’, you may have specified asynchronous clocks, reference clocks and/or branch clocks. Some might be functional clock domain bases of some instruments, namely MemoryBist. They could be used in some timing constraints.

This information is stored in the array `tessent_clock_mapping`. It is defined in the generated SDC file, within the proc `tessent_set_default_variables`. It has the following form:

```
array set tessent_clock_mapping {
  <TessentShell ClockLabel> <FunctionalSDC ClockLabel>
  [...]
}
```

This array is used by the Tessent Shell tool-generated SDC constraints to refer to your functional clocks. They do so only through a remapped “`tessent_clock_mapping(<TessentShell ClockLabel>)`” array element. By default, in `tessent_set_default_variables`, `<TessentShell ClockLabel>` and `<FunctionalSDC ClockLabel>` are identical.

Note

 It is imperative that you examine this array and look for cases where this default mapping must be updated. Overriding names are not necessary if you have used the “`-label`” option of the [add_clocks](#) command in Tessent Shell to match the “`-name`” value in the SDC.

If you need to update one of the `tessent_clock_mapping()` array values, please do so in your main synthesis script, immediately after the call to the `tessent_set_default_variables` proc. For

example, if you had a clock defined with a label "CK25" in Tessent Shell, but in your SDC constraints the same clock is defined with a name "PIN_CK25", you need to specify:

```
set tessent_clock_mapping(CK25) PIN_CK25
```

The new definition of `tessent_clock_mapping(CK25)` overrides the default one contained in `tessent_set_default_variables`.

It is important to understand that you do not need to edit the Tessent-generated SDC file, but rather use the 'set' command in your synthesis script to overwrite the value of the Tessent SDC variable array element you want to change. Alternatively, if many of your Tessent defined clocks and respective SDC clocks have different names, you might want to redefine a block of the array:

```
set tessent_clock_mapping(CLK25) pCLK25
set tessent_clock_mapping(CLK_PLL_REF) pCLK100
```

OR

```
array set tessent_clock_mapping {
  CLK25 pCLK25
  CLK_PLL_REF pCLK100
}
```


IMPORTANT: You do not need to define `tessent_clock_mapping()` entries for clocks that you have not specified in Tessent Shell. Those are not used in the timing constraints.

Preparation Step 4: Redefining Other Tessent Tcl Variables

The `tessent_tck_period` and `tessent_clock_mapping` array variables are the most important Tessent SDC variables you should double check and overwrite as needed. Tessent SDC uses a number of additional variables that are all defined and set to a default value in the `tessent_set_default_variables` proc.

For example, you can independently change the input and output delay (from the default, 25% of the `tessent_tck_period`) or change the hierarchy separator character. Please see the header of the generated SDC file and the embedded comments in the `tessent_set_default_variables` proc, explaining each such variable. Within the synthesis tool, you can use the TCL command "info body tessent_set_default_variables" to see what variables are being defined.

Note

 The value of `tessent_tck_period` might depend on the maximum tck clock frequency that can be applied to the circuit. See the "[IJTAG Network Performance Optimization](#)" section in the *Tessent IJTAG User's Manual* showing how to maximize the frequency of the IJTAG network test clock.

Synthesis Step 1: Applying the SDC Constraints

If you use Design Compiler, you need to set the variable “timing_enable_multiple_clocks_per_reg” to true. This is needed to properly constrain MemoryBIST in the presence of MemoryBIST clock muxing.

```
set_app_var timing_enable_multiple_clocks_per_reg true
```

To apply the Tessent Shell-generated SDC constraints, run the merged non_modal proc:

```
tessent_set_non_modal
```

This Tessent SDC proc in turn calls all instrument non-modal procs as needed by your design and its sub-blocks. There is nothing else for you to do than call this one proc to apply all non-modal SDC constraints.

Synthesis Step 2: Preparing the DFT Logic for Synthesis

Once your design is loaded, you need to make sure to preserve instances of Tessent Shell-inserted “persistent” cells and possibly instrument instances, depending on your downstream needs.

Refer to “[Synthesis Helper Procs](#)” on page 758 for more information on the different options.

Persistent cells should not be optimized as Tessent Shell SDC uses them as anchor points for layout and STA, and different instances need to be preserved depending on if you do scan insertion with Tessent Shell or more DFT insertion. Procs are provided in the .sdc file to generate a list of all instances that need to be preserved in different ways:

tessent_get_preserve_instances, tessent_get_optimize_instances, and tessent_get_size_only_instances.

First you need to prevent boundary optimization of DFT objects using the proc tessent_get_preserve_instances:

```
set_app_var compile_enable_constant_propagation_with_no_boundary_opt false
set_preserve_instances [tessent_get_preserve_instances scan_insertion]
set_boundary_optimization $preserve_instances false
set_ungroup $preserve_instances false
set_app_var compile_seqmap_propagate_high_effort false
set_app_var compile_delete_unloaded_sequential_cells false
```

Then, because boundary optimization applies hierarchically, you re-enable boundary optimization of the child instances of DFT objects using tessent_get_optimize_instances:

```
set_boundary_optimization [tessent_get_optimize_instances] true
```

Cells from your provided Tessent Cell Library instantiated in Tessent Shell must also be preserved but can be resized to enable critical data path timing optimization during synthesis. You can get them using `tessent_get_size_only_instances`:

```
set_size_only -all_instances [tessent_get_size_only_instances]
```

If your design contains shared bus assemblies, you can save significant area by ungrouping their content, with the exception of the MemoryBIST controller. Refer to the “Synthesis Step 3” portions in “[Example Design Compiler Synthesis Script](#)” and “[Example Genus Synthesis Script](#)” for examples of the suggested procedures.

Synthesis Step 3: Synthesizing Your Design

You can now proceed with the compilation of your design:

```
link  
check_design  
compile -boundary_optimization [...]
```

Synthesis Step 4: Writing Out Your Final SDC

Use the `write_sdc` command after this step to export the merged SDC (Functional and Tessent Shell) for use with your layout tool:

```
write_sdc ${design_name}.merged_sdc
```

Synthesis Step 5: Writing Out Your Final Netlist

You can now proceed with writing out your synthesized netlist.

```
write -format verilog -output ${design_name}.vg ${design_name} -hier
```

Running Layout with Tessent Shell DFT

There exist a few possible ways to apply your Tessent SDC constraints in layout.

Note



This section uses the following convention to shorten proc names:

```
<ltest_prefix> := tessent_set_ltest
```

You can do either of the following:

- Use the SDC that you wrote out in previous synthesis Step 5 and feed it directly to your layout tool, or

- Define all your functional SDC constraints and add Tessent DFT constraints, like you previously did in synthesis, by repeating the steps:
 - Define your functional constraints.
 - Set your global TCL tessent variables.
 - Call `tessent_set_non_modal`.

You also must preserve your Tessent DFT persistent cells from further layout logic optimization. Get the list of these cells by calling your SDC file proc `tessent_get_size_only_instances`.

That is all you must do if your design does *not* feature Tessent logictest DFT. If you used 3rd-party scan insertion tools, please refer to their instructions as to how to constrain that mode. The rest of this section details what you must do in layout with Tessent logictest DFT.

Clock Tree Synthesis With Tessent On-Chip Clock Controllers (OCCs)

If your design includes Tessent OCC IP, you may need to instruct your clock tree synthesis command to stop balancing the inner flop clock pins of your OCC with the leaves of the functional domain clock tree that the OCC controls. This balancing adds a significant amount of latency to the OCC flop clocks, which causes a drastic reduction to the setup margin of the internal `fast_clock_en` and `slow_clock_en` timing paths of the OCC to the OCC clock gaters at the root of the controlled functional domain tree.

Locate the `tessent_get_cts_skew_groups_dict` inside the output `.sdc` file from the `extract_sdc` command, and follow the instructions provided in the banner of that proc to disable this balancing.

Loading SDC Constraints in Layout With Tessent Logictest

In layout, some issues may limit one's ability to constrain the logictest DFT simultaneously with all other logic. For instance:

1. Letting `scan_en=X`, unblocks a large number of false at-speed path between neighboring flops on the same scan chains, unless you have your own custom way to globally disable these paths in your own design environment and flow.
2. As opposed to synthesis, where all clocks are ideal, propagating `test_clock` with `scan_en=X` in layout may enable false cross-domain paths between unbalanced functional domains. Those paths become single-cycle paths of `test_clock` which, if `test_clock` is unbalanced, may cause both setup and hold false timing violation.
3. Some false at-speed capture paths might get unblocked between test-only dedicated wrapper cells and functional logic.

If your layout flow provides custom methods to work around these issues, you can run layout using only one global set of constraints that covers both your functional and DFT modes. In

such a case, you would only need to call the `tessent_set_non_modal` proc with no argument, or use the extracted `.sdc` file from synthesis. If not, then you need to use the multi-mode feature of your layout tool and sequentially load the following modal SDC constraints:

Mode 1:

- Set your functional SDC constraints.
- Call `tessent_set_non_modal` with the argument “off”, like in:

```
tessent_set_non_modal off
```

This adds all Tessent DFT constraints, but disables all logictest DFT paths.

Mode 2:

- Call proc `<ltest_prefix>_modal_shift`
- This forces the “shift” mode over your design, asserting `scan_en` to 1 creating scan mode clocks and taking care of DFT signals, OCC, and EDT logic setup. This covers all of your scan chain and EDT channels paths, as well as any pipelining flop timing along the way.
- Run layout incrementally, so as to fix any leftover timing paths that were blocked in the first pass.

Mode 3:

- Call proc `<ltest_prefix>_modal_edt_slow_capture`. This proc does the following:
 - Creates `test_clock` and propagates it across the design.
 - Lets `scan=X`, which enables covering the timing of that signal as well as some leftover timing paths inside Tessent’s OCC clock gating circuit.
 - Disables same-edge paths from `test_clock` to `test_clock`, leaving only retimed cross-domain paths enabled, so as to prevent false timing violations across functional domains. Intra domain paths are assumed covered more tightly by the functional mode constraints.
 - Enables capture paths across your design’s top ports as single-cycle paths of `test_clock`.
- Run layout incrementally again.

To save time and resources, you can choose to skip mode #3 if you know you are running scan mode at a very slow clock speed and that all your capture paths are guarded with extra cycles of `test_clock`.

SDC for Modal Static Timing Analysis

The following are guidelines to help you perform static timing analysis of the embedded test hardware that was inserted by Tessent Shell.

Checking Your Functional Logic Alone	726
Checking Your Embedded Test Logic	726

Checking Your Functional Logic Alone

If you want to only focus on functional logic and completely disable all Tessent Shell embedded test IP, include the following steps in your primary functional STA script.

First, if flops in your cell library do not propagate constant settings through their set or reset pin to their data output pin, you need to set a PrimeTime variable:

```
set case_analysis_sequential_propagation always
```

This enables disabling all embedded test modes with just a few asserts of the TAP's reset port.

For example:

```
-----  
... loading your functional constraints ....  
set case_analysis_sequential_propagation always  
# Hold the DFT in reset and kill TCK to make sure only functional logic is  
# active.  
set_case_analysis TRST 0  
set_case_analysis TCK 0  
-----
```

You also need to disable your scan circuit timing by forcing your scan_enable pin to its inactive value, and by issuing a set_false_path to/from all your lockup cells and your pipeline flops. Those can normally be found with regular expressions such as:

```
[get_pins -hier <standard naming>*/d]  
[get_pins -hier <standard naming>*/q]
```

You might also have to turn your BISR clock or reset pin off, if present, like in:

```
set_case_analysis bisr_clk      0  
set_case_analysis bisr_reset   0
```

Checking Your Embedded Test Logic

You can find a self-documenting example script of how to run STA using Tessent Shell generated hardware.

Refer to “[Example PrimeTime STA Script](#)” on page 764. Carefully read the instructions in this script before proceeding.

The script executes the following:

- Loads the generated SDC file and initializes some Tcl variables used by Tessent Shell SDC constraints.
- Sets some PrimeTime control variables to their required value.
- Sources user input files if present in the same directory as follows:
 - *user_timing_data.tcl*
 - *user_remapping_procs.tcl*
 - *load_design.pt*
- Loads your final netlist.
- Sequentially runs all EmbeddedTest modes present in your design, each separated by a “reset_design” command.
- Runs an example “report_constraints” command for each mode and redirects its output to a file.

VHDL and Mixed Language Designs

Synthesis tools have similar but sometimes different internal representations of elaborated designs. Here are some instructions on how to deal with those differences to be able to use the SDC constraints generated by Tessent Shell.

VHDL Generate Blocks..... 728

VHDL Generate Blocks

Required tool settings and other mixed-language issues.

Synopsys Design Compiler

Constraints applying to cells and pins in VHDL generate loops are problematic to apply in `dc_shell`. The default naming scheme of the internally synthesized design is different for Verilog and VHDL. To illustrate, here is a Verilog path of an instance within 2 nested generate for loops: `blk[0].jblk[0].i0`. The equivalent configuration in VHDL would be changed during `dc_shell`'s GTECH pre-synthesis and would be named `i0_0_0`. Because Tessent Shell's SDC does not "flatten" generate loops, if you have VHDL generate loops in your design, you must use the following setting in `dc_shell` to restore the loop naming:

```
set_app_var hdlin_enable_upf_compatible_naming true
```

This variable ill restores VHDL generate loop naming to follow Verilog style: `blk(0)/jblk(0)/i0`

Cadence Encounter and Genus

There are no special steps for these two synthesis tools to be able to apply Tessent Shell generated SDC constraints to a mixed language design.

Oasys

Use the following commands in Oasys to generate a naming scheme for generate loops (Verilog and VHDL) that is compatible with the Tessent SDC constraints:

```
set_parameter generate_naming_style %s[%d]  
set_parameter if_generate_naming_style %s
```

Dealing with Unrolled VHDL Generate for Loops

Tessent Shell tries to minimize unrolling of generate loops. If VHDL generate loops are unrolled then one "if-generate" block is generated for each loop iteration. The trailing closing square brace, that would normally be changed to an underscore, has to be truncated because of VHDL language restrictions. This means that functional SDC constraints pointing to cells or pins within those generate blocks do not match even if special characters are mapped to


wildcards. Ex: the instance path blk[0].i0 which would need to be unrolled would be written out as blk_0.i0.

For this reason, you should use the provided procs `tessent_get_pins` and `tessent_get_cells`. They are able to match your functional instance path “blk[0].i0” to blk_0.i0 using a caching algorithm that searches for generate blocks in the path and tries to match with our unrolling strategy if the path does not match outright. Refer to the “[Mapping Procs](#)” on page 756 to see the procs’ description.

SDC File Contents

This section describes the procs contained in the generated SDC file. Note that many of these procs are called automatically by other procs in the file. You do not need to call all of these procs.

Note

 This section uses the following convention to shorten proc names:


<ltest_prefix> := tessent_set_ltest

tessent_set_default_variables	730
tessent_create_functional_clocks	731
tessent_<design_name>_set_dft_signals	731
<ltest_prefix>_disable	732
tessent_set_non_modal	732
tessent_<design_name>_kill_functional_paths	733
IJTAG Instrument	734
LOGICTEST Instruments	735
MemoryBIST Instrument	748
BoundaryScan Instrument	751
Hierarchical STA in Tessent	752
Mapping Procs	756
Synthesis Helper Procs	758

tessent_set_default_variables

This proc defines the initial/default value of all global variables used in all procs of the sdc file. The variables contained in this proc are there for you to customize the constraints without having to edit the constraints themselves. A good example of this would be the variable “tessent_tck_period”. It currently defaults to 100.0 (nanoseconds). If your TCK clock has a different period than 100ns, you can overwrite the value of this variable after having called the proc tessent_set_default_variables but before calling the constraint procs.

Note

 The value of tessent_tck_period might depend on the maximum tck clock frequency that can be applied to the circuit. See the “[IJTAG Network Performance Optimization](#)” section in the *Tessent IJTAG User’s Manual* showing how to maximize the frequency of the IJTAG network test clock.

The mapping between LogicBIST and InSystemTest controller IDs and their instance path names is available in this proc. Use the mapping information to indicate the clocks of these instruments to the other ltest SDC procs as described in “[Preparation Step 2: Setting and Redefining Tessent Tcl Variables](#)” on page 718.

This proc must always be the first proc to be run.

tessent_create_functional_clocks

This proc is generated for your convenience. It contains the SDC create_clock and create_generated_clock commands to define the various functional clocks needed by the instruments constrained by the SDC procs. Typically, your functional clocks would already be defined by your functional SDC. If that is the case, you should override the default values of the tessent_clock_mapping array to match the clock names you used in the definition of your clocks.

This proc would typically not be called in your synthesis SDC.

Required clocks are determined with ICL tracing. create_clock constraints are added for all clock sources like top level ClockPort ports and source ToClockPort ports (embedded crystals). create_generated_clock constraints are added for all generated clocks (add_clocks -reference), branch clocks and unidentified ICL ToClockPort ports (typically PLLs). If your design contains an ICL instrument with a ToClockPort that should not require a new generated clock, please set the [exclude_from_sdc](#) attribute on the port.

tessent_<design_name>_set_dft_signals

This procedure forces all your specified static dft_signal sources to either reset or get their default value during all_test, depending on the proc's argument.

```
argument mode ::= reset | logic_off
```

- **logic_off** — Recommended argument for running non-logictest STA modes such as memoryBIST. But if you added Siemens EDA TestKompress, your sdc file will also get the proc <ltest_prefix>_disable. You then only need to call that proc for running the other non-logictest modes such as modal memoryBIST.
- **reset** — Disables all DFT-inserted logic.

If you are using functional-only mode, you can optionally call the proc with the “reset” argument under one of the following circumstances:

1. Many static dft signals come from top-level ports.
2. Although all static dft signals sourced by the Siemens EDA IJTAG Test Data Registers (TDR) reset when the IJTAG reset port is asserted, the timing tool would not propagate

that assert through the TDR flops. That may happen in Synopsys PrimeTime when the control variable “case_analysis_sequential_propagation” is set to “never”.

<ltest_prefix>_disable

This proc disables all Logictest logic in your design. You call this proc when setting up exclusive STA mode for your IJTAG, BoundaryScan or MemoryBIST, and you do not want to bother about ltest logic and scan chains timing at the same time.

This proc is invoked by proc `tessent_set_non_modal` when called with the `logictest` argument set to “off”. Specifying “on” would invoke proc `tessent_set_ltest_non_modal` instead. You would typically use “on” during pre-layout synthesis and “off” during layout. In layout, you would then need to apply two different mode scripts in your layout tool. The second mode would time logictest-only shift logic, for which you would invoke the proc `<ltest_prefix>_shift`. For more information about this proc, see “<ltest_prefix>_modal_shift” on page 741.

You can call the proc with or without an argument:

- `<ltest_prefix>_disable` — Disables logictest controller and forces the `all_test dft` signal active. Use this when setting up the membist STA mode. For an example, see “[Example PrimeTime STA Script](#)” on page 764.
- `<ltest_prefix>_disable all_test_x` — Disables the logictest controller but does not force the `all_test dft` signal. Use this when constraining your design for synthesis or for layout. For more information, see “[tessent_set_non_modal](#)” on page 732

tessent_set_non_modal

This proc contains calls to the `non_modal` procs of all constrained instruments, which are documented in the sections that follow. For synthesis, this is the only constraint proc you need to call.

You can call this proc with no argument, or with the argument “off”:

```
tessent_set_non_modal off
```

The argument “off” disables logictest circuitry by asserting `dft_signal` source pins and prevents adding any non-modal logictest clocks or constraints. The default is to call the proc `tessent_set_ltest_non_modal`.

This is an example of the `tessent_set_non_modal` proc using the design “blka”:


```
proc tessent_set_non_modal {{logictest "on"}} {
  tessent_set_ijtag_non_modal
  tessent_set_memory_bisr_non_modal
  tessent_set_memory_bist_non_modal
  if {$logictest eq "on"} {
    tessent_set_ltest_non_modal
  } else {
    tessent_set_ltest_disable all_test_x
  }
}
```

For an example of how to use the proc, see “Single-Mode vs Dual-Mode Constraining in Synthesis/Layout” in “<ltest_prefix>_non_modal” on page 744.

tessent_<design_name>_kill_functional_paths

This procedure disables all timing paths involving non-Tessent flops. It enables running test-only timing analysis without having to fix functional path violations with your own functional SDC constraints. It helps making functional mode STA and Siemens EDA DFT mode STA fully orthogonal, and reduces the Siemens EDA DFT mode STA run-time.

Caution

 Do not call this procedure for any logic test STA modes. It is only intended for use in the Siemens EDA DFT mode, as shown in “[Example PrimeTime STA Script](#)” on page 764.


You can optionally run this procedure when you know that your functional logic has many intra-domain timing paths that cannot meet default single-cycle path timing with the clocks being set by `tessent_create_functional_clocks` and `tessent_set_*` procs.

Disabling some sequential cells in your design might accidentally block the test clocks feeding your inserted instruments. Examples of such cells are Integrated Clock Gating (ICG) cells or flops that belong to clock dividers. You can fix this by specifying those cells either by module name or by instance names.

By module name:

```
set ClockSeqCellModuleRegExp "<regularExpression>"
```

Note

 This only finds library cells that match this module name pattern. If your module is hierarchical and the cells you are trying to preserve are below it, use the instance name option described in the following.

By instance name:

```
set ClockSeqCellInstanceRegExp "<regularExpression>"
```

Note



This expression is tested against the full hierarchical instance paths. A pattern that would match a hierarchical instance preserves all cells within it.

The procedure optionally creates a report file containing a sorted list of all disabled flops instance names. To enable this option, you need to define the following variable prior to calling `tessent_kill_functional_paths`:

```
set CreateDisabledFlopsReport 1
```

IJTAG Instrument

Tessent IJTAG provides a single proc:

- `tessent_set_ijtag_non_modal`
 - This proc would typically not be called directly in your synthesis script, it is called as part of `tessent_set_non_modal`.

This proc contains the clock creation for your TCK clock and configurable input and output delays for created ports at the `sub_block` and `physical_block` design levels.

It also creates an asynchronous clock group with all TCK clocks. It is used to prevent synthesis from balancing paths between TCK and functional clocks. This clock group could be recreated with additional clocks if your design contains Tessent BoundaryScan or Tessent OCC hardware. This is managed via the “`tessent_tck_clock_list`” global variable. This variable is built from different procs (`ijtag` and `jtag_bscan`) and used to create an asynchronous clock group. If you have your own TCK generated clocks that are not used in the Tessent Shell SDC constraints, we suggest adding their name to this list right after calling `tessent_set_default_variables`.

```
tessent_set_default_variables  
lappend tessent_tck_clocks_list my_tck_generate_clock
```

At the sub-block and physical block levels, constraints are added to reflect the timing of the IJTAG protocol. The IJTAG reset port is declared a false path and the IJTAG select port's hold paths are declared false and its setup paths are given two cycles with a `multicycle_path` constraint.


The select signal source of all non-scan SIBs (part of the SRI network) is relaxed to MCP setup 2 hold 2. This is to reflect the protocol and allow really deep combinational paths to close timing. The SIBs part of the STI network were not included in this constraint as those should be local to each physical region and it should be easier for timing tools to close timing. If you

experience problems with scan-testable SIB select signals, you can add those to the list of SIBs that are relaxed, however, this will have an adverse impact on scan patterns.

LOGICTEST Instruments

The section lists the logic test-based procs. LogicBIST and InSystemTest-related procs listed in the following are included only when LogicBIST and InSystemTest controllers are present in the design.

Note

 This section uses the following convention to shorten proc names:

<ltest_prefix> := tessent_set_ltest

1. Non-modal constraints, to be merged with other DFT instrument constraints, for synthesis or layout. See this proc description for a discussion on single-mode synthesis/layout vs multi-mode synthesis/layout flow:
 - o [<ltest_prefix>_non_modal](#)
 - o tessent_set_in_system_test_non_modal
2. Modal procs, used for signoff STA and optionally for synthesis or layout constraining:
 - o [<ltest_prefix>_modal_shift](#)
 - o [<ltest_prefix>_disable](#)
 - o [<ltest_prefix>_modal_lbist_shift](#)
3. Procs only for per-mode signoff STA constraints:
 - o [<ltest_prefix>_edt_shift](#)
 - o [<ltest_prefix>_bypass_shift](#)
 - o [<ltest_prefix>_single_bypass_chain_shift](#)
 - o [tessent_set_edt_slow_capture](#)
 - o [<ltest_prefix>_edt_fast_capture](#)
 - o [<ltest_prefix>_modal_lbist_shift](#)
 - o [<ltest_prefix>_modal_lbist_capture](#)
 - o [<ltest_prefix>_modal_lbist_setup](#)
 - o [<ltest_prefix>_modal_lbist_single_chain](#)
 - o [<ltest_prefix>_modal_lbist_controller_chain](#)

4. Procs sub-invoked by the preceding procs.
 - [<ltest_prefix>_create_clocks](#)
 - [<ltest_prefix>_set_pin_delays](#)

<ltest_prefix>_create_clocks

This proc creates the slow-speed test clocks for use during scan mode. There are a few possible clock configurations:

- **tessent_test_clock** — This is a DFT signal in Tessent Shell. It should be your tester's synchronous clock and is used to create the `shift_capture_clock` and the `edt_clock`.
- **tessent_edt_clock** — This clock can either be a primary input port or it can be generated from the `test_clock` and is used to time the shift paths of the EDT logic.
- **tessent_shift_capture_clock** — This clock can either be a primary input port or it can be generated from the `test_clock` and is the main scan mode clock propagating to your functional logic through the OCCs.
- **tessent_virtual_force_pi** and **tessent_virtual_measure_po** — Two virtual clocks of the same frequency as the preceding `tessent_test_clock`, used to time all top-level ports that directly interface with your EDT controllers and your scan chains, through the “`set_input_delay`” and “`set_output_delay`” constraints.
- **tessent_lbist_shift_clock_src** — This clock is typically an internally generated free-running clock used as the source for LogicBIST tests.

When using LogicBIST, the clock gaters (CGC) shown in the following figure for `tessent_edt_clock` and `tessent_shift_capture_clock` are located inside the LogicBIST controller, and the clock source automatically updates to this new location. This clock is created by the user outside of the Tessent-generated SDC.

This proc is also called by every SDC proc that propagates the `edt_clock` or shift clocks, specifically:

- `<ltest_prefix>_non_modal`
- `<ltest_prefix>_shift`
- `<ltest_prefix>_modal_slow_capture`
- `<ltest_prefix>_modal_lbist_shift`
- `<ltest_prefix>_modal_lbist_capture`

The following is an example of the created clocks you would get if you ran the following commands:

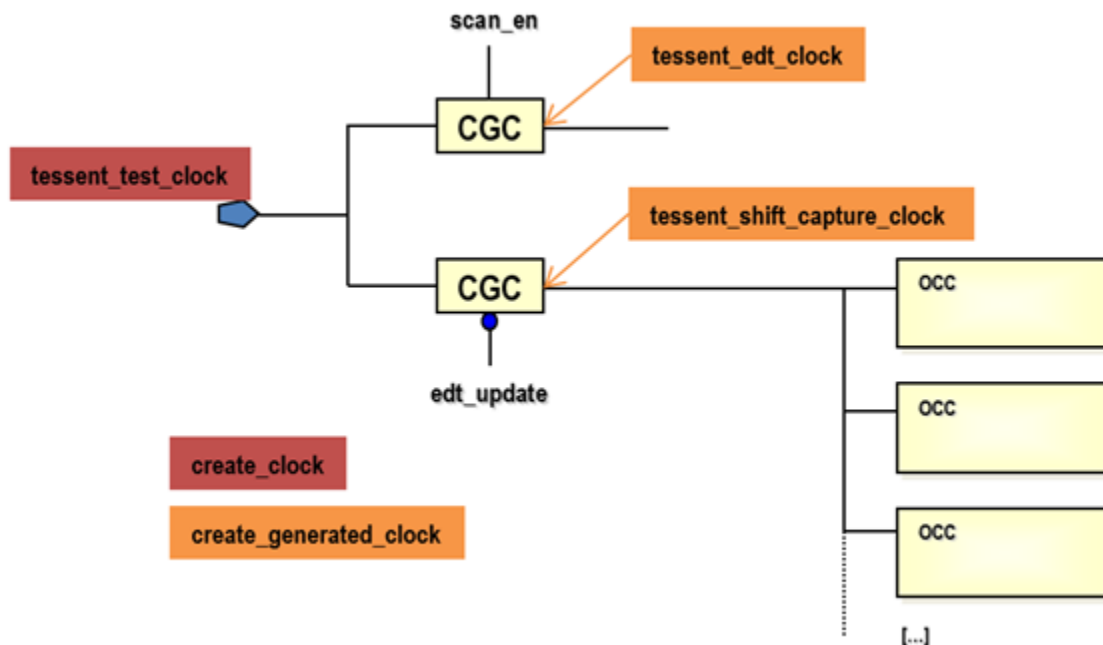
```
add_dft_signals test_clock -source_nodes [get_ports test_clock]
```


add_dft_signals {edt_clock shift_capture_clock} -create_from_other_signals

```

set slow_clock_period [expr $tessent_slow_clock_period * $time_unit_multiplier]
set sc_rise_time       [expr $tessent_shift_clock_edge1_percentage/100. * $slow_clock_period]
set sc_fall_time       [expr $tessent_shift_clock_edge2_percentage/100. * $slow_clock_period]
set sc_waveform        "$sc_rise_time $sc_fall_time"
set force_pi_rise      [expr $tessent_force_pi_percentage/100. * $slow_clock_period]
set measure_po_rise    [expr $tessent_measure_po_percentage/100. * $slow_clock_period]
set min_width          [expr 0.25 * $slow_clock_period]
set force_pi_waveform "$force_pi_rise [expr $force_pi_rise + $min_width]"
set measure_po_waveform "$measure_po_rise [expr $measure_po_rise + $min_width]" #
test_clock:
  create_clock [tessent_get_ports test_clock] \
    -add -period $slow_clock_period -waveform $sc_waveform\
    -name tessent_test_clock # edt clock:      create_generated_clock -source
[tessent_get_ports test_clock] \      [tessent_get_pins tessent_persistent_cell_edt_clock/
GCK] -divide_by 1 \      -name tessent_edt_clock \      -add -master_clock tessent_test_clock
# Virtual force_pi clock, to comply with your timeplate "force_pi" specifications \
  create_clock \
    -period $slow_clock_period -waveform $force_pi_waveform \
    -name tessent_virtual_force_pi # Virtual measure_po clock, to comply with your
timeplate "measure_po" specifications \
    create_clock \
    -period $slow_clock_period -waveform $measure_po_waveform \
    -name tessent_virtual_measure_po

```

Figure 13-1. tessent_test_clock and tessent_shift_capture_clock Creation**Tessent Slow Clocks Waveform Definitions**

To maintain consistency of the scan clocks and scan ports timing between your SDC and your backannotated scan simulations, you need to define the following variables based on your Fastscan timeplate specifications. These variables shape the waveforms of the

tessent_test_clock, tessent_virtual_force_pi, and tessent_virtual_measure_po clock definitions in your SDC. The variables are as follows:

- **tessent_slow_clock_period**
Tessent slow clock period (the same for all clocks).
Default: 40ns.
- **tessent_shift_clock_edge1_percentage**
Timeplate position of the shift clock rising edge as a percentage of its period.
Default: 50.
- **tessent_shift_clock_edge2_percentage**
Timeplate position of the shift clock falling edge as a percentage of its period.
Default: 75.
- **tessent_force_pi_percentage**
Timeplate position of the force_pi point as a percentage of the shift clock period.
Default: 0.
- **tessent_measure_po_percentage**
Timeplate position of the measure_po point as a percentage of the shift clock period.
Default: 25.

Using percentages instead of absolute time values enables you to quickly modify the tessent_slow_clock_period specification without also having to update the other Tcl variables.

The defaults described in the preceding reflect the Tessent FastScan timeplate default specifications, that is:

```
timeplate gen_tpl =  
    force_pi 0 ;  
    measure_po 10 ;  
    pulse_clock 20 10 ;  
    period 40 ;  
end;
```

resulting into the following default clock definitions:

```
create_clock <port> -add -period 40. -waveform {20 30} -name tessent_test_clock  
create_clock -period 40. -waveform {0 10} -name tessent_virtual_force_pi  
create_clock -period 40. -waveform {10 20} -name tessent_virtual_measure_po
```

Finally, two more global Tcl variables might be necessary to specify the timing budget of the scan data paths outside of the current module:

- **tessent_scan_input_delay**

Default value: 0.

- **tessent_scan_output_delay**

Default value: 0.

Update these variables if you plan to retarget your test vectors from a higher level module and you budgeted some propagation delay for your scan signals in that module. Here is how the variables are used:

```
set_input_delay $tessent_scan_input_delay -clock tessent_virtual_force_pi <port>
set_output_delay $tessent_scan_output_delay -clock tessent_virtual_measure_po <port>
```

Currently, the [extract_sdc](#) command assumes that timeplate specified values are identical for both load_unload and capture phases, which is the Tessent FastScan default. Although, the “non_modal” ltest proc has to use only one set of specifications, modal shift and capture STA may have to run with different timeplate specifications. If it is your case, you can do it by updating the preceding Tcl variables prior to calling the selected shift or capture modal proc. The fast_capture modal proc ignores those values, because it uses your functional waveform specifications.

<ltest_prefix>_edt_fast_capture

This proc must be called in your STA to constrain the EDT in fast capture mode. Logictest fast capture mode is the one that detects your at-speed transition faults. It counts on user SDC to provide the capture clocks and timing exceptions that would actually time the capture paths. Typically this would be your original functional SDC constraints, but those might still require some adjustments if your test conditions (clock frequencies, false paths) slightly differ from what they are in functional mode. Obviously, you also need to remove any constraint that resets the IJTAG network or ties the scan_enable or test_enable control pins to zero.

If present, the following dft signals are turned off:

- scan_enable (putting the circuit in capture mode)
- control_test_point_en (these hold during capture)

The proc also enables you to check that some dft signals such as x_bounding_en, memory_bypass_en, observe_testpoint_en are doing their isolation job correctly. You can optionally force them to a known value by setting global Tcl variables prior to calling the proc.

The following demonstrates how this is done (annotations are in red):

```
global memory_bypass_en_value <<< set this variable if needed in
                                   your primary script
if {[info exists memory_bypass_en_value]} {
    set_case_analysis $memory_bypass_en_value \
    [tessent_get_pins elt1_mbist_tessent_tdr_sri_ctrl_inst/
tessent_persistent_cell_memory_bypass_en/Y]
}
global x_bounding_en_value <<< set this variable if needed in
                               your primary script
if {[info exists x_bounding_en_value]} {
    set_case_analysis $x_bounding_en_value \
    [tessent_get_pins elt1_dft_tessent_tdr_sri_ctrl_inst/
tessent_persistent_cell_x_bounding_en/Y]
}
global observe_test_point_en_value <<< set this variable if needed
                                       in your primary script
if {[info exists observe_test_point_en_value]} {
    set_case_analysis $observe_test_point_en_value \
    [tessent_get_pins elt1_dft_tessent_tdr_sri_ctrl_inst/
tessent_persistent_cell_observe_test_point_en/Y]
}
```

Finally, the proc also disables any other timing paths that are ignored during capture, such as those going through the EDT channel pins.


tessent_set_edt_slow_capture

This proc must be called in your STA to constrain the EDT in slow capture mode.

The proc does the following:

- Forces the test_enable dft signals on.
- Declares the test_clock and lets your scan_en dft signal toggling, so that it can be timed and relaxed with your number of dead cycles specification.
- Disables same-edge paths from test_clock to test_clock, leaving only retimed cross-domain paths enabled, so as to prevent false timing violations across functional domains. Intra domain paths are assumed covered more tightly by the functional mode constraints; enables capture paths across your design's top ports as single-cycle paths of test_clock.
- Disables ignored timing paths.
- Sets an MCP for your scan_en signal, using your global TCL variables tessent_scan_en_setup_extra_cycles and tessent_scan_en_hold_extra_cycles.
- Set an MCP for your edt_update signal, if present, using your global variables tessent_edt_update_setup_extra_cycles and tessent_edt_update_hold_extra_cycles.
- Enables capture paths across your design's top ports as single-cycle paths of test_clock.

Note

 Just as with the preceding `<prefix>shift_ltest_non_modal` proc, propagating `test_clock` may create false capture timing paths across asynchronous domains as single-cycle paths of `test_clock`. This may or may not be a problem, given that `test_clock` fanout is balanced and running at slow speed. By default, this proc assumes that all valid intra-domain hold timing paths were already covered by your functional modes STA, and therefore sets a `multicycle_path` of 1 hold to all paths from `test_clock` to itself. Remember that all scan mode shift paths are properly covered for both setup and hold by the “`xxx_ltest_shift`” mode proc defined in the preceding. If you need to, you can still force the proc to revert to zero-hold constraint by setting the Tcl variable `tessent_time_hold_in_slow_capture` to value 1 in your calling script.

`<ltest_prefix>_modal_shift`

This proc sets the circuit in scan shift mode. It covers both EDT shift modes and all available bypass modes, assuming you run them all with the same shift clock frequency. If you need more specific timing analysis, you can choose to apply the following procs in separate STA runs. These procs sub-invoke the `<ltest_prefix>_shift` proc:


- `<ltest_prefix>_edt_shift`
- `<ltest_prefix>_bypass_shift`
- `<ltest_prefix>_single_bypass_chain_shift`

The `<ltest_prefix>_shift` proc applies a set of common constraints required for shifting. Then they only need to complete their mode setting by forcing the `edt_bypass` and `single_chain_bypass` signals to the required constant value.


The proc `<ltest_prefix>_shift` does the following:

- Creates the shift clocks.
- Applies `set_input/output_delays` to scan control pins.
- Forces dft signals `ltest_en` and `scan_en` to 1.
- Applies any LogicBIST mode constraints as applicable.

Note

 You can save on total STA time and flow complexity if you already plan to apply the same `test_clock` (with the same period) for all shift and bypass modes. In this case you only need to call the `<ltest_prefix>_shift` proc directly in place of all of the three procs defined in the preceding.

Note

 If you plan to feed your synthesis or layout tool with two separate mode scripts (functional/dft and logictest), the `<ltest_prefix>_shift` proc is the one you need to apply for the logictest mode. That way, all possible shift paths are timed, and because `scan_enable` is forced active, it prevents the existence of a potentially large number of false capture timing paths across your functional logic. It also covers all of your possible scan chain configurations at once, including the internal and external mode for cores. See later discussion on logictest single-mode or dual mode usage—see `<ltest_prefix>_non_modal`.

`<ltest_prefix>_modal_lbist_shift`

You can configure timing analysis for this mode to run with either `shift_clock_src` or `test_clock`. The default is `shift_clock_src`. Add the optional `test_clock` argument when calling this SDC TCL proc to analyze timing with `test_clock`.

The proc does the following:

- Adds multi-cycle path exceptions on dynamic signals from the LogicBIST controller to design scan cells and hybrid EDT blocks. These include LogicBIST mode scan enable, `prpg_en`, `misr_en`, and LogicBIST synchronous reset for the hardware default mode of operation.
- Adds case analysis on the mux, which chooses between the top-level scan enable for ATPG and the LogicBIST controller-generated scan enable to enable only the LogicBIST mode paths.
- Adds case analysis to set `lbist_en` to active.
- Disables single chain mode scan chain concatenation paths.
- Disables paths from `edt_chain_mask` masking registers to the scan cells. This constraint is required because even though this register is configured using `tck` as the source, the clock net connected to the `edt_lbist_clock` signal carries both `edt_clock` and `shift_clock_src` clocks.

`<ltest_prefix>_modal_slow_capture` and `<ltest_prefix>_modal_fast_capture`

You get these procs in your SDC file only under the following conditions:

- You added the following dft signals in your current design level: `scan_en`, `test_clock`, or (`edt_clock` and `shift_capture_clock`).
- And you did not insert a Tessent logictest-related controller, such as EDT or `logicbist` in that same level.

These two procs properly assert all static dft signals, such as scan_en and ltest_en. They are the “no-EDT” version of the following two procs:

- `<ltest_prefix>_modal_edt_slow_capture`
- `<ltest_prefix>_modal_edt_fast_capture`

Because no EDT is present in the current design, extract_sdc does not generate these procs:

- `<ltest_prefix>_edt_bypass_shift`
- `<ltest_prefix>_edt_single_chain_shift`

However, if your design meets all of the following conditions:

- Your core instances actually do support edt_bypass chain or single_chain modes.
- You intend to run these modes at a test_clock frequency that differs from their EDT mode.
- You intend to use core-extracted timing models in a hierarchical STA flow.

Then you should load a per-mode core timing model, and then run the `<ltest_prefix>_modal_shift` proc as a general setup for the top-level. For each such STA run, you must properly set the global Tcl variable tessent_slow_clock_period value.

For `<ltest_prefix>_modal_edt_fast_capture`, all edt_bypass and edt_chain_bypass asserts are skipped by default. If you do not want to skip these asserts, define the “tessent_block_edt_bypass_in_fast_capture” global variable.

<ltest_prefix>_modal_lbist_capture

This mode is similar to `<ltest_prefix>_modal_lbist_shift`, except that the LogicBIST mode scan enable is constrained to off.

<ltest_prefix>_modal_lbist_setup

This mode propagates tck through the IJTAG network within the LogicBIST controller and hybrid EDT blocks to time the LogicBIST test_setup paths that initialize registers such as PRPG and edt_chain_mask, as well as test_end paths that read the MISR signature.

This mode does not propagate tck to the design scan cells because it is only intended to check the IJTAG network paths.

<ltest_prefix>_modal_lbist_single_chain

This mode is available when single chain mode logic that enables LogicBIST diagnosis is enabled during IP generation. This mode propagates tck through the IJTAG network paths and design scan cells, including the concatenation of the internal scan chains for single-chain

shifting. The LogicBIST scan enable is constrained to 1 to time only the shift paths in the design with the tck signal.

<ltest_prefix>_modal_lbist_controller_chain

This mode is available when controller chain mode (CCM) logic is enabled during IP generation. When CCM is present, all of the previously described LogicBIST modes disable the controller chain logic by constraining the `ccm_en` signal to off. This mode tests the CCM paths by constraining the `ccm_en` signal to on. The clock for this mode is either `tck` or `test_clock`, as specified during IP creation.

This mode only tests the LogicBIST and hybrid EDT controller blocks; clocks are not propagated to design scan cells.

Paths that are normally disabled during the LogicBIST operation modes described in the preceding become valid single-cycle paths in CCM and are timed accordingly.

- Signals such as `lbist_en` are not constrained to 1 because paths from the TDR register in the LogicBIST controller to the hybrid EDT blocks that use this signal are tested with ATPG in CCM. Paths from static signals—such as `x_bounding_en`, `test_point_en`, and `capture_per_cycle`—to the design scan cells are correctly excluded because no clocks are propagated to the scan cells.
- Paths from dynamic control signals that are declared as multi-cycle for other modes—such as `scan_en`, `prpg_en`, and `misr_en`—that go from the LogicBIST controller to the EDT blocks are tested as valid single-cycle paths.
- Paths from IJTAG network nodes such as SIBs are timed as valid single-cycle paths of the CCM clock.
- Input and output pin delays from top-level ports to EDT blocks and CCM scan I/O ports are included for this mode. This differs from the LogicBIST shift and capture modes, which do not have any active paths from or to design ports.

<ltest_prefix>_non_modal

This procedure provides SDC timing constraints to add to your combined functional-dft non-modal timing scripts for one-pass synthesis or layout. Its contents is merged with both your functional constraints and all other Tessent controller's non-modal constraints. It is called by the umbrella proc “`tessent_set_non_modal`” proc.

The procedure calls the previously described procs:

- `<ltest_prefix>_create_clocks`
- `<ltest_prefix>_set_pin_delays`

It also:

- Defines a specific clock group (using 'set_clock_groups') for all logictest slow clocks, isolating them from your declared functional clocks.
- Adds 'set_false_path' constraints from each of your primary ports assigned to a static dft signals such as 'all_test', 'ltest_en' 'edt_mode'. If the same signals come instead from an internal IJTAG Test Data Register (TDR), then the “set_clock_groups” command between tessent_tck and the scan clocks replaces the individual set_false_path commands in the following.
- Provides constraints to prevent “tck” and your functional clocks to propagate to unwanted paths inside your Siemens EDA OCCs, as well as constraints that prevent false timing violations on the select pin of clock muxes inside that OCC.
- Provides constraints that disable automatic clock gating checks across input pins of the Siemens EDA OCC clock multiplexers, because all OCC clock selection is either performed statically during test setup or at slow speed in a glitchless way during scan.
- Adds an optional set_multicycle_path constraint from your primary 'scan_en' port and “edt_update” signal, based on your setting of the global variables:
 - tessent_scan_en_hold_extra_cycles
 - tessent_scan_en_setup_extra_cycles
 - tessent_edt_update_hold_extra_cycles
 - tessent_edt_update_setup_extra_cycleswhich all default to zero, meaning single-cycle paths of slow clock.
- When using LogicBIST, includes a 3-to-1 shift_clock_select mux at the clock structure root of the LogicBIST controller. This mux enables any one of three clocks—shift_clock_src, test_clock or tck—to be used as the source clock for LogicBIST test.
 - Blocks tck propagation through the shift_clock_select mux into the design scan cells. This removes analysis of the slowest LogicBIST mode of operation using tck to reduce timing analysis run time.
 - Propagates both shift_clock_src and test_clock to the design scan cells, but all interactions between them are blocked.
- Adds the following LogicBIST-related exceptions:
 - Declares multi-cycle paths from LogicBIST scan enable generation logic to the design scan cells. The number of cycles are based on the pre_post_shift_dead_cycles IP generation parameter and defaults to eight.
 - Declares multi-cycle paths from logic that generates the prpg_en, misr_en, and LogicBIST synchronous reset for hardware default mode operation signals to the

hybrid EDT blocks. The number of cycles are based on the `pre_post_shift_dead_cycles` IP generation parameter.

- Disables paths from static control registers in the LogicBIST controller—such as `lbist_en` and `x_bounding_en`—to design scan cells and hybrid EDT blocks. This is implicitly performed by declaring `tck` as asynchronous to other clocks in the IJTAG non-modal proc. When CCM is implemented with the EDT clock as the CCM clock, explicit false paths are added to disable such paths.
- Excludes single chain mode logic scan chain concatenation paths through case analysis on the single bypass chain control TDR output.
- When CCM is implemented with `test_clock` as the CCM clock, the `test_clock` propagates to all IJTAG scan elements on the `tck` domain. Constraints are added to disable such paths to the `tessent_lbist_shift_clock_src` clock.

Single-Mode vs Dual-Mode Constraining in Synthesis/Layout

Important: As it stands in the current release, leaving the `scan_enable` signal free to toggle in this proc is known to create false `shift_clock`-frequency timing paths across your functional design, which may or may not affect timing closure or quality of results in layout. Siemens EDA customers' experience greatly varies on this front. Here are some examples of such false paths:

1. Scan chains intra-domain shift-only paths are constrained at the speed of the functional clock.
2. As a result of creating only one `shift_capture_clock` source and letting it propagate to all functional clock domains, all cross-domain capture paths become single-cycle of the shift clock, regardless of whether they are asynchronous in functional mode or not.

This said, these new timing paths do not instantly condemn your layout or physical synthesis tool to fail timing closure or perform a bad P&R job, knowing that clock tree synthesis also balances the `test_clock` fanout by default. Non-physical synthesis is generally not a problem.

On the other hand, re-constraining of all those bogus timing paths would typically require a very large number of design-dependent timing exceptions, which `extract_sdc` is not able to provide at this point. Applying them could also slow down some layout tools considerably.

The alternative to single-mode constraining is to apply individual mode scripts at different times in the synthesis or layout tool. It is a well-known solution that has been applied by most Siemens EDA customers historically.

1. Functional/Dft mode:
 - Apply your functional constraints
 - Invoke `proc constrain_<design_name>_non_modal off`

2. Logictest-only mode, covering all EDT scan configurations, such as EDT shift and EDT bypass shift:
 - Invoke proc `<ltest_prefix>_modal_shift`
3. If your design contains hybrid EDT/LBIST, another logictest-only mode to cover the LBIST shift configuration:
 - Invoke proc `<ltest_prefix>_modal_lbist_shift`

This script cannot be combined with EDT scan configuration script because these modes are not compatible; they propagate different shift clocks, have different case analysis constraints on the `lbist_en` signal, and have different timing requirements for scan enable with respect to the shift clock.

`<ltest_prefix>_set_pin_delays`

This procedure assigns the clock “`tessent_virtual_slow_clock`” to your top-level ports that directly interface with your EDT controllers or your scan chains during scan or EDT mode, using the “`set_input_delay`” and “`set_output_delay`” timing constraints.

This proc is called by every SDC proc which propagate the EDT or shift clocks, that is:

- `<ltest_prefix>_non_modal`
- `<ltest_prefix>_shift`
- `<ltest_prefix>_modal_slow_capture`

Input/Output Pin Delay Constraints

Input and output delay constraints are declared for the EDT control and channel pins. For example:

```
# channel_input:
set_input_delay $tessent_scan_input_delay \
                -clock tessent_virtual_slow_clock \
                [get_ports {my_edt_channels_in[0]}]

# channel_output:
set_output_delay $tessent_scan_output_delay \
                -clock tessent_virtual_slow_clock \
                [get_ports {my_edt_channels_out[0]}]

# edt_update:
set_input_delay $tessent_scan_input_delay \
                -clock tessent_virtual_slow_clock \
                [get_ports edt_update]
```

DFT Signals Handling in ltest STA Procs

All EDT-related modal STA procs in the following force these dft signals, when present, to their active value:

- all_test (active in all tests)
- ltest_en (active in logictest only)

For example:

```
set_case_analysis 1 [get_ports all_test]
set_case_analysis 1 [get_ports ltest_en]
```

The same procs leave all other logictest-related dft signals toggling, and add a false_path constraint if they come from a primary port. For example:

```
set_false_path -from [get_ports async_set_reset_static_disable]
set_false_path -from [get_ports control_test_point_en]
set_false_path -from [get_ports ext_ltest_en]
set_false_path -from [get_ports ext_mode]
```

If the same signals come instead from an internal IJTAG Test Data Register (TDR), then the “set_clock_groups” command between tessent_tck and the scan clocks replaces the individual set_false_path commands.

Procedure “tessent_set_ltest_edt_fast_capture”, on the other hand, optionally asserts some of these dft_signals under control of a user-declared variable.

MemoryBIST Instrument

Tessent MemoryBIST provides the following procs:

- tessent_set_memory_bist_non_modal for chip synthesis
 - This proc would typically not be called in your synthesis script, it is called as part of tessent_set_non_modal.
- tessent_set_memory_bist_modal for STA
 - This proc must be called in your STA script if you want to constrain the MemoryBist mode.

- tessent_<design_name>_top_set_dft_signals logic_off

Specify this call if your design also feature logictest-based dft signals that need to be turned off in membist STA mode.

- tessent_mbist_set_ai_timing_mode

This procedure assures all functional clocks are defined and properly propagated, but leaves the asynchronous interface free of constraints so they can be formally analyzed by procedure `tessent_mbist_report_controller_ai_timing`. This proc will not be present or needed if none of the controllers being constrained utilize an asynchronous interface.

- `tessent_mbist_report_ai_timing [-verbose 1|0] [-tck_period time_in_ns]`

This procedure runs `tessent_mbist_report_controller_ai_timing` once for every MemoryBIST controller in the current design. This proc will not be present or needed if none of the controllers being constrained utilize an asynchronous interface.

- `tessent_mbist_report_controller_ai_timing [-verbose 1|0] [-controller_id id] [-tck_period time_in_ns]`

This procedure accurately measures the BAP AI timing paths to the specified MemoryBIST controller. The covered AI signals are: BIST_SHIFT, BIST_SI, BIST_SO, and BIST_HOLD.

Because of the asynchronous nature of this interface, the signals listed in the preceding cannot be accurately timed with SDC constraints. Although the circuit is designed to always work with a low TCK frequency, this procedure validates that it works at the frequency you specified.

Each AI signal timing margin depends on a combination of the TCK period, the controller's BIST_CLK period, and the skew related to the TCK branch reaching that AI TCK transition detector. We recommend running this procedure only once with your worst case operating conditions. In most cases, AI signal timing is expected to largely exceed their setup timing requirements. Hold timing is never a problem because of the design of the interface.

This proc will not be present or needed if none of the controllers being constrained utilize an asynchronous interface.

They contain many multicycle path constraints to and from controller registers aiming to minimize the impact of the MemoryBIST DFT on your timing closure.

Multi-Clock Memories

Additional timing exceptions were added for synthesis if you have multi-port memories functionally driven by multiple clocks. A mux is inserted in the memory interface to enable both clock ports of the memory to be driven using a single clock during the Memory BIST mode. The insertion of this clock multiplexer adds different timing modes between functional and test modes which must be described correctly in the SDC file.

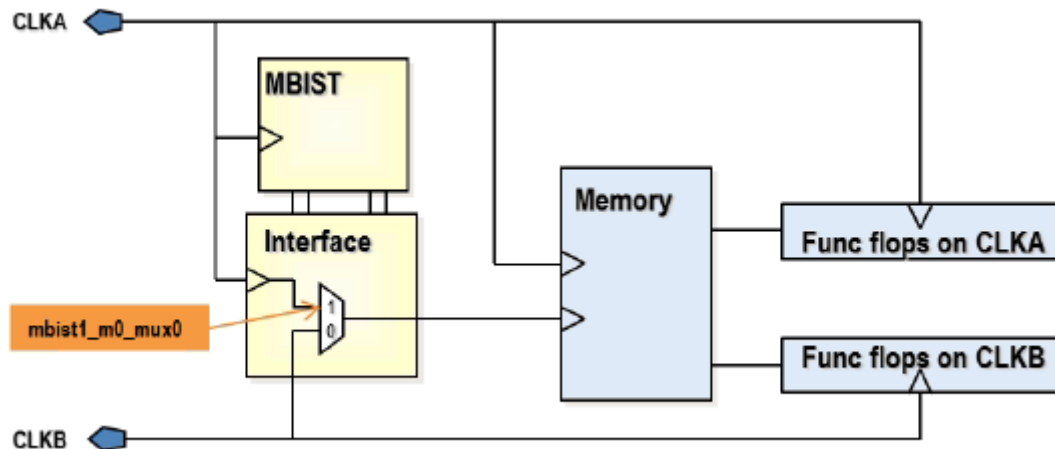
Note

If you wish to skip the memoryBIST constraints of clock muxing (creating the generated clock and false paths) for multi-clock memories, set the variable `tessent_apply_mbist_mux_constraints` to “0” in the `tessent_set_default_variable` proc. This variable should be set to “0” in two cases:

- For synthesis: When the BIST clock used by the MemoryBIST tests is declared as false path with the memory’s functional clock in the user’s SDC script.
 - For STA: When verifying the timing of the scan modes.
-

In the following, we explain key points of the MemoryBist constraints to handle these modes harmoniously. Suppose the situation where CLKA and CLKB are two of your functional clocks that drive two clock ports of a single memory. The MemoryBIST logic is inserted and runs on CLKA. CLKA is multiplexed onto the CLKB branch of the memory during tests.

Figure 13-2. Generated Clock Muxed Multi-Clock Memories



First, we need to create a generated clock on the input of the clock mux. In our example, this clock is called `mbist1_m0_mux0`.

```
create_generated_clock [tessent_get_pins  
$tessent_memory_bist_mapping(mbist1_m3)/tessent_persistent_cell_MUX1/b] \  
-name mbist1_m0_mux0 \  
-source [get_ports CLKA] \  
-add -master_clock $tessent_clock_mapping(CLKA) \  
-divide_by 1
```

Creating this clock at the input of the clock mux (instead of the output) enables all input clocks to propagate through the mux and times the following interactions precisely:

- Timing paths between the memory BIST logic and the memory
- Timing paths between the functional flops and the memory

With the `mbist1_m0_mux0` defined, we can now define clock-based exceptions to declare as false any interaction between `CLKB` and the memory BIST flops, and between `mbist1_m0_mux0` and the functional flops on `CLKB`. Those false paths are described using the following SDC commands:

```
set nonMBISTclocks [remove_from_collection [all_clocks] [get_clocks
"$tessent_clock_mapping(CLKA) mbist1_m0_mux0 "]]
set_false_path -from [get_clocks mbist1_m0_mux0] \
-to $nonMBISTclocks
set_false_path -from $nonMBISTclocks \
-to [get_clocks mbist1_m0_mux0]
set_false_path -from [get_clocks $tessent_clock_mapping(CLKB)] \
-to [tessent_get_cells [concat \
$tessent_memory_bist_mapping(mbist1_m3)/SCAN_OBS_FLOPS* \
$tessent_memory_bist_mapping(mbist1_m3)/MBISTPG_STATUS/GO_ID_REG* \
$tessent_memory_bist_mapping(mbist1_m3)/FREEZE_STOP_ERROR*]]
set_false_path -from [tessent_get_cells [concat \
$tessent_memory_bist_mapping(mbist1_m3)/Q1_SCAN_IN* \
$tessent_memory_bist_mapping(mbist1_m3)/Q2_SCAN_IN* \
$tessent_memory_bist_mapping(mbist1_m3)/BIST_INPUT_SELECT*]] \
-to [get_clocks $tessent_clock_mapping(CLKB)]
set_false_path -from [tessent_get_cells [concat \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_PORT_COUNTER/PORT_COUNTER* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_FSM/STATE* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_POINTER_CNTRL/
EXECUTE_OP_SELECT_CMD** \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_SIGNAL_GEN/JCNT* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_SIGNAL_GEN/OPSET_SELECT_REG* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_DATA_GEN/WDATA_REG* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_DATA_GEN/X_ADDR_BIT_SEL_REG* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_DATA_GEN/Y_ADDR_BIT_SEL_REG* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_ADD_GEN/AX_ADD_REG* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_ADD_GEN/AY_ADD_REG* \
$tessent_memory_bist_mapping(mbist1)/MBISTPG_OPTION/BIST_EN_RETIME2* \
$tessent_memory_bist_mapping(mbist1)/MEM_SELECT_REG*]] \
-to [get_clocks $tessent_clock_mapping(CLKB)]
```

By making these timing exceptions, we are precisely timing the memory BIST interaction with the memory using `CLKA` reaching the two-clock ports of the memory and the functional logic to memory interaction using the `CLKB` reaching the memory clock port. The added logic is precisely timed to simplify timing closure. These constraints make physical design tools aware of the mux on the clock port of the memory and they place the mux in a location where it does not affect timing.

BoundaryScan Instrument

Tessent BoundaryScan provides the following proc:

- `tessent_set_jtag_bscan_non_modal`
 - This proc would typically not be called directly in your synthesis script, it is called as part of `tessent_set_non_modal`.

It creates generated clocks from each BoundaryScan interface and relaxes the timing between them.


Hierarchical STA in Tessent

This section explains how to use Tessent-generated SDC constraint procs to run the hierarchical STA flow with Siemens EDA DFT. This flow runs STA separately on each individually laid-out physical block. If a block instantiates other lower-level physical blocks, partial or full back-annotated netlists or extracted models of those physical blocks are loaded. The procs enable you to cover all Siemens EDA DFT timing paths crossing the lower physical block boundaries, including those related to the IJTAG interface, BISR register interface, bscan interface, and logictest scan/capture paths, while preventing interference and false violations from irrelevant functional paths inside these physical blocks.

When the following conditions occur, the [extract_sdc](#) command creates a few callable SDC timing procs to help with the hierarchical STA flow:

- The current DFT-inserted design instantiates lower-level physical blocks.
- Physical blocks have been DFT-inserted with Siemens EDA IP, using [add_dft_signals](#) commands, along with the DftSpecification flow.
- For “xxx_ltest_xxx” procs, lower-level physical blocks are assumed to have the following:
 - One or more EDT controllers.
 - Optional Tessent OCCs.
 - An external logictest mode, controlled with DFT signals ext_ltest_en and optionally int_ltest_en.

Note

 This hierarchical STA flow is not the same as running flat logictest STA on a full design netlist with multiple physical levels. Running flat STA would require an extra layer of complexity for the needed constraints, especially if you intend to run each level with a different test_clock frequency. Tessent Shell currently generates no SDC procs to support this type of procedure.

The [extract_sdc](#) command provides the following hierarchical STA procs:

- **tessent_set_modal_lower_pbs** — Enables all Siemens EDA DFT non-logictest timing paths through physical block pins and disables all paths through the rest of the same physical block pins.
- **tessent_set_ltest_pb_external_mode** — Enables the external logictest mode on an isolated physical block, forcing its dedicated wrapper cells to select the boundary logic.

Call this proc before extracting your current physical block design timing model, for use at the next level up.

- **<ltest_prefix>_lower_pbs_external_mode** — Configures the instantiated lower physical blocks of the parent design in their external mode, enabling running parent logictest modes with loaded lower physical block full or partial netlists.

The `extract_sdc` command generates the `<xxx>_mentor_modal_lower_pbs` proc even when the Logictest IP comes from non-Siemens EDA sources. The command only generates the other two if the IP comes from Tessent TestKompress.

Hierarchical STA Procs Descriptions

The following describe the hierarchical STA procs in detail.

`tessent_set_modal_lower_pbs`

The tool creates this proc only for parent designs instantiating lower-level physical blocks.

The proc provides STA coverage of the Siemens EDA non-logictest DFT timing path going across the physical block pins while blocking all other paths. Call it along with either the non-modal constraints of the current design or the modal DFT STA constraints (membist, BISR, bscan, IJTAG, with disabled logictest). You need to load some timing models for your lower physical block instances.

The proc does the following:

- Disables timing through your physical block interface pins, except those active used Siemens EDA non-logictest DFT, such as the IJTAG pins, the BISR register pins, or the embedded boundary scan control pins. All Siemens EDA logictest IP related pins are also disabled.
- Relaxes some paths in physical block's IJTAG network.
- Blocks TCK from propagating to physical block logic through the eventual Siemens EDA OCC muxes.

Assuming that your current hierarchical STA flow methodology may require loading a mix of black boxes from lower physical blocks, extracted timing models, or backannotated netlists, the proc always checks whether a physical block's internal pin is actually loaded in memory before attempting to apply a constraint to it. As a result, the call to the proc never errors out because of unloaded lower physical block logic.

Disabling timing on all your physical block input's clock pins purposely kills their non-DFT internal functional timing paths. Likewise, physical block scan logic does not require any additional SDC constraints, because of the following:

- The global `scan_enable` control signal is assumed already off in their parent design, leaving all scan flops in functional mode.

- All physical block relevant logic (such as controllers, functional scan flops, LogicBist) receives no clock.

The proc is called along with current level `<xxx>_non_modal` or Siemens EDA DFT modal constraints procs to complete timing coverage of both non-modal synthesis and pre/post-layout modal STA steps.


The following is a typical example:

```
<load your current design files>
<load your lower physical blocks timing models or annotated netlist>
<link your design>
source ${tsdb}/dft_inserted_designs/.../${design}.sdc
tessent_set_default_variables
<override global variable values if needed>
```

The following example includes non-modal constraints (*):

```
<define your functional clocks and your timing exceptions>
tessent_set_non_modal <arg>
tessent_set_modal_lower_pbs
update_timing
```

Note

 The `<arg>` value is required only if your design contains Siemens EDA logictest IP.

The following example includes Tessent MemoryBIST modal constraints (*):

```
tessent_create_functional_clocks
tessent_set_ijtag_non_modal
tessent_set_jtag_bscan_non_modal
tessent_set_memory_bisr_non_modal
tessent_set_memory_bist_modal
tessent_kill_functional_paths
tessent_set_modal_lower_pbs
update_timing
```

(*) IJTAG constraints are always present in any design. The presence of all other procs are design-dependent.

tessent_set_ltest_pb_external_mode

The tool creates this proc only for isolated physical blocks with the “ext_ltest_en” DFT signal present. It forces ext_ltest_en active and int_ltest_en (if present) inactive. Use this call after a previous call to one of the xxx_ltest_modal_xxx procs, to select the external mode version of your shift, bypass, or capture modes. The proc merely forces the lower physical block’s ext_ltest_en and int_ltest_en DFT signals to their external mode requirements. You should always call this proc when extracting your physical block’s timing model for later use in your parent ltest STA modes, because it prevents ambiguous timing paths in your extracted model timing arcs.

Current-level `<xxx>_ltest_modal*` STA procs aim to cover both internal and external mode test paths in the same STA run, despite that letting `ext_ltest_en` toggling may introduce a few false paths that could affect your design timing closure. For most cases, it is a risk worth taking, because merging multiple STA modes together is highly desirable for minimizing your total number of STA runs.

The following shows an example of this proc:

```
<load your current design files>
<link your design>
source ${tsdb}/dft_inserted_designs/.../${design}.sdc
tessent_set_default_variables
<override global variable values if needed>

# Extract one timing model per ltest mode
# Under some conditions, you can merge edt_shift and bypass_shift into
# "shift".
foreach mode {edt_shift bypass_shift edt_slow_capture} {
  reset_design
  tessent_set_ltest_modal_${mode}
  tessent_set_ltest_pb_external_mode
  set_propagated_clock [all_clocks]
  update_timing
  # SYNOPSIS PrimeTime-specific command
  extract_model -output $design_name}_ext_${mode}_etm -format lib -library
}
```

`<ltest_prefix>_lower_pbs_external_mode`

The `extract_sdc` command writes this proc only for designs instantiating lower-level physical blocks featuring the DFT signal `ext_ltest_en`, implying physical block wrapper isolation logic. The objective of this proc is to set all lower physical blocks into external mode when running one of the `logictest` STA modes at the parent level, which covers `logictest` mode timing paths through the boundary pins of these physical blocks. Path coverage includes all logic between the physical block pins and their wrapper cells, in addition to the wrapper chains shift mode paths.

Assuming that your current hierarchical STA flow methodology may require loading a mix of black boxes from lower physical blocks, extracted timing models, or back-annotated netlists, the proc always checks whether a physical block's internal pin is actually loaded in memory before attempting to apply a constraint on it. As a result, the call to the proc never errors out because of unloaded lower physical block logic.

Such constraints include:

- Preventing `logictest` slow clock reconvergent timing paths inside the physical block OCC logic.
- Asserting the `ltest_en dft_signal`.
- Asserting and deasserting the `ext_ltest_en` and `int_ltest_en dft_signals`.

- Deasserting the lbist_en dft_signal, if present.
- Conditionally asserting or deasserting the following dft_signals:
 - edt_mode, int_edt_mode, ext_edt_mode, edt_bypass
 - memory_bypass_en
 - async_set_reset_static_disable
- Preventing false timing checks within the physical block OCC clock multiplexing logic.

The following shows an example of this proc:

```
<load your current design files>
<load your lower physical blocks timing models or netlist>
<link your design>
source ${tsdb}/dft_inserted_designs/.../${design}.sdc
tessent_set_default_variables
<override some global tessent TCL variables value if needed>
foreach mode {shift edt_shift bypass_shift edt_slow_capture} {
  reset_design
  tessent_set_ltest_modal_${mode}
  tessent_set_ltest_lower_pbs_external_mode
  set_propagated_clock [all_clocks]
  update_timing
  <report_timing and other checks>
}
```

Mapping Procs

Because you can use the Tessent SDC constraints both pre- and post-synthesis, the tool must perform some mapping to find all pins and cells with the pre-synthesis instance path.

The mapping is accomplished with the following procs:

- tessent_get_ports
- tessent_get_pins
- tessent_get_cells
- tessent_get_flops
- tessent_map_to_verilog
- tessent_remap_vhdl_path_list

The procs tessent_get_ports, tessent_get_pins, and tessent_get_cells are wrapper procs of the original SDC commands get_ports, get_pins, and get_cells, and they are used throughout the Tessent SDC. These wrapper procs use the tessent_map_to_verilog and tessent_remap_vhdl_path_list procs as required. Use these mapping procs (as opposed to get_ports, get_pins, and get_cells) in your functional SDC if you have a design with unrolled

VHDL generate loops, and you have problems applying your functional SDC to the RTL. See [“Dealing with Unrolled VHDL Generate for Loops”](#) on page 728 for more information.

The `tessent_get_flops` proc runs the `tessent_get_cells` proc and then filters the result collection for sequential elements. The proc uses a filtering method appropriate for the current tool.

The `tessent_map_to_verilog` proc adds wildcarding to pre-synthesis instance paths to match post-synthesis instance paths for layout and STA. For example, the instance path `core_inst/my_reg[0]` would return the pattern `core_inst/my_reg?0?`. This is to be able to match both the RTL instance path and the post-synthesis path `core_inst/my_reg_0_` even if “change_names -rules Verilog” was used. It also maps the slash (/) coming from Tessent Shell instance paths to any user-defined “tessent_hierarchy_separator”. This proc is called by `tessent_get_cells` and `tessent_get_pins`. You should not need to call this proc directly.

The `tessent_remap_vhdl_path_list` proc is used when the instance path cannot be found with the simple mapping. It should only be needed when Tessent Shell has unrolled some VHDL generate loops to do uniquified insertion in the instances within them. See [“Dealing with Unrolled VHDL Generate for Loops”](#) on page 728 for more information. The proc supports finding cells/instances that have a trimmed last character, for example a closing brace or a question mark. This proc is called by `tessent_get_cells` and `tessent_get_pins` if needed. You should not need to call this proc directly. You can also use the proc in the unlikely event that your synthesis tool changed the names of cells in an unexpected way such as trimming any trailing closing brace, for example “my_reg[0]” changed to “my_reg_0” instead of the expected “my_reg_0_”

If you require additional mapping capabilities, the `tessent_map_to_verilog` proc provides a method for custom mapping of regular expressions/substitutions that run prior to the tool’s mapping process. To use this, you must add your own mapping by defining the global array variable.

For example:

```
global tessent_custom_mapping_regsub
array set tessent_custom_mapping_regsub {
  {\| (/| |$)} {\1}
}
```

This example maps all RTL instance paths from the SDC file to remove any closing bracket preceding a hierarchy separator (/) or at the end of the path (space for end of list element, \$ for end of list). You would need this mapping expression during your STA run if the synthesis tool was trimming the closing brace of registers after a `change_name`. For example, `my_reg[0]` -> `my_reg_0` instead of `my_reg_0_` as expected.

Synthesis Helper Procs

To be able to apply Tessent Shell SDC constraints on your post-synthesis netlist, some steps are required to preserve the boundary of certain instruments as well as to preserve certain “persistent” cells, which must not be optimized away.

Three procs are provided to return the design instances which need to be boundary preserved, shall be optimized or must be entirely preserved:

- `tessent_get_preserve_instances` *preservation_intent*
- `tessent_get_optimize_instances`
- `tessent_get_size_only_instances`

`tessent_get_preserve_instances` *preservation_intent* returns a collection of instances whose boundaries must be preserved. It must be provided an argument to establish the future use of the netlist and determine which instances need to be preserved. The “select” value you need to choose depends on whether you intend to use your post-synthesis netlist with our tools. Here are the valid values for the “select” argument, in increasing order of inclusiveness:

- [add_core_instances](#)

This usage selection returns instances that need to be preserved for applying SDC constraints for STA to your post-synthesis design, and instances required for the TCD automation flow with ATPG.

- `scan_insertion`

This usage selection returns all instances of the previous selection, plus any instance of modules having existing scan segments described in a [TCD scan file](#) and non-scan instances (ICL attribute `keep_active_during_scan_test=true`). This selection is required if you intend to do scan insertion on your design with Tessent Shell or a third party tool.

- `icl_extract`

This usage selection returns all instances of the two previous selections, plus any instance of modules providing an ICL definition. This context is needed if you intend to go through the `DftSpecification` flow again, this time with your gate level netlist. ICL extraction requires that all ICL instances be present and traceable in the design.

If persistent cells added by Tessent Shell are RTL constructs (RTL cells or wrappers from the cell library), they are returned by `tessent_get_preserve_instances`. If they are library leaf cells, they are not returned by `tessent_get_preserve_instances`. Use the `tessent_get_size_only_instances` proc to obtain these (see following).

The `tessent_get_optimize_instances` proc returns a collection of child instances within Tessent instruments whose boundaries can be optimized. This should be used when your synthesis tools propagates boundary optimization attributes downward hierarchically.

The `tessent_get_size_only_instances` proc returns a collection of instances of all persistent cells, although the collection may be empty under certain conditions. These cells are required to be intact to apply SDC constraints for layout and STA. However, the synthesis tool can change the size of the driving stage as needed.

Example Scripts using Tessent Tool-Generated SDC

This section provides example scripts for use when using Tessent Shell tool-generated SDC.

Example Design Compiler Synthesis Script	760
Example Genus Synthesis Script	761
Example PrimeTime STA Script	764

Example Design Compiler Synthesis Script

The following is an example synthesis script to use with Design Compiler.

```
# Prerequisite: Functional Synthesis Script
<load your functional design and constraints here, including all clock
definitions and functional timing exceptions>
#source ${design_name}.dc_shell_import_script
#elaborate $design_name
#create_clock [get_ports pCLK25] -name pCLK25 -period 25.0
#create_clock [get_ports pCLK100] -name pCLK100 -period 100.0

# Preparation Step 1: Sourcing SDC File
set design_name top
set tsdb ../tsdb_outdir

source \
${tsdb}/dft_inserted_designs/${design_name}_rtl.dft_inserted_design/ \
${design_name}.sdc

# Preparation Step 2: Setting and Redefining Tessent Tcl Variables
tessent_set_default_variables
set tessent_tck_period 80.0

# Uncomment the following line if you wish to skip the memoryBIST
# constraints of clock muxing for multi-clock memories:
#set tessent_apply_mbist_mux_constraints 0

# Preparation Step 3: Verifying the Declaration of Functional Clocks
# Set clock names in mapping array
array set tessent_clock_mapping {
  CLK25 pCLK25
  CLK_PLL_REF pCLK100
}
# Preparation Step 4: Redefining Other Tessent Tcl Variables
# set tessent_input_delay [expr 0.3 * $tessent_tck_period]

# Synthesis Step 1: Applying the SDC Constraints
set_app_var timing_enable_multiple_clocks_per_reg true
tessent_set_non_modal
```



```
# Synthesis Step 2: Preparing the DFT Logic for Synthesis
set_app_var compile_enable_constant_propagation_with_no_boundary_opt false
set preserve_instances [tessent_get_preserve_instances icl_extract]
set_boundary_optimization $preserve_instances false
set_ungroup $preserve_instances false
set_app_var compile_seqmap_propagate_high_effort false
set_app_var compile_delete_unloaded_sequential_cells false

set_boundary_optimization [tessent_get_optimize_instances] true

set_size_only -all_instances [tessent_get_size_only_instances]

# shared bus assembly ungrouping
foreach_in_collection assembly \
  [get_designs -hierarchical *_tessent_mbist*_shared_bus_assembly] { \
    current_design $assembly \
    set_ungroup \
    [get_cells -filter "ref_name!~*_tessent_mbist*_controller*"] true \
  }
current_design $design_name

# Synthesis Step 3: Synthesizing Your Design
link
check_design
compile -boundary_optimization

# Synthesis Step 4: Writing Out Your Final SDC
write_sdc ${design_name}.merged_sdc

# Synthesis Step 5: Writing Out Your Final Netlist
write -format verilog -output ${design_name}.vg ${design_name} -hier
```

Example Genus Synthesis Script

This section provides an example synthesis script to use with Genus.

```
# Proposed global settings
foreach {attrName attrValue} {
  hdl_bidirectional_assign false
  bus_naming_style {%s(%d)}
  uniquify_naming_style {%s_%d}
  hdl_auto_async_set_reset true
  init_blackbox_for_undefined false
  write_vlog_top_module_first true
  remove_assigns false
  minimize_uniquify true
  continue_on_error false
  log_command_error true
  report_tcl_command_error true
} {
  dict set global_attributes_dict $attrName $attrValue
  set_db $attrName $attrValue
}
```

Timing Constraints (SDC) Example Genus Synthesis Script

```
# Prerequisite: Functional Synthesis Script
<load your functional design and constraints here, including all
clock definitions and functional timing exceptions>

# Preparation Step 1: Sourcing SDC File
set design_name top
set tsdb ../tsdb_outdir
source ${tsdb}/dft_inserted_designs/ \
${design_name}_rtl.dft_inserted_design/${design_name}.sdc

# Preparation Step 2: Setting and Redefining Tessent Tcl Variables
tessent_set_default_variables
set tessent_tck_period 80.0
# Uncomment the following line to skip the memoryBIST constraints
#   of clock muxing for multi-clock memories:
#set tessent_apply_mbist_mux_constraints 0

# Preparation Step 3: Verifying the Declaration of Functional Clocks
# Set clock names in mapping array.
# The array keys are the names Tessent Shell knew of the clocks.
# The array values are the names of those clocks in your current SDC.
array set tessent_clock_mapping {
    CLK25 pCLK25
    CLK_PLL_REF pCLK100
}

# Preparation Step 4: Redefining Other Tessent Tcl Variables
# set tessent_input_delay_factor 0.3
# set tessent_hierarchy_separator "_"

# Synthesis Step 1: Applying the SDC Constraints
tessent_set_non_modal
```

```

# Synthesis Step 2: Preparing the DFT Logic for Synthesis
proc do_set_boundary_optimization { instances_collection state } {
  if { [sizeof_collection $instances_collection] == 0 } { return }
  puts "Modules of the instances"
  set modules_collection [get_db $instances_collection .module]
  puts "  [join [get_db $instances_collection .module] "\n "]"
  # Preserve all instances, even those on which boundary optimization
  # is requested.
  set_db $instances_collection .ungroup_ok false
  # Boundary optimization is avoided by preserving the footprint (pins).
  if { !$state } {
    # Boundary optimization is avoided for all modules of the instances.
    # - Preserve the footprint of all the pins
    # - Do not propagate constant inputs while optimizing internal logic
    # - Do not consider any input of opposite polarity
    set_db $modules_collection .boundary_opto strict_no
  }
}

proc do_set_size_only { instances_collection } {
  if { [sizeof_collection $instances_collection] == 0 } { return }
  foreach {preserveValue isSequential} [list map_size_ok \
    true size_ok false] {
    set instancesFiltered [filter_collection \
      $instances_collection "is_sequential==$isSequential"]
    if { [llength $instancesFiltered] == 0 } { continue }
    puts "\nSetting preserve attribute '$preserveValue' to \
      '$isSequential' on:"
    set_db $instancesFiltered .preserve $preserveValue
  }
}

set preserveInstances [tessent_get_preserve_instances icl_extract]
do_set_boundary_optimization $preserveInstances false
set optimizeInstances [tessent_get_optimize_instances]
do_set_boundary_optimization $optimizeInstances true
set set_size_only_collection [tessent_get_size_only_instances]
do_set_size_only $set_size_only_collection

# Ungroup shared bus assemblies if any are present
set sb_mods [vfind . -subdesign *_tessent_mbist_*_shared_bus_assembly]
set sb_insts [get_db $sb_mods .instance]
foreach sb_inst $sb_insts {
  # ungroup everything in the assembly but the controller
  ungroup [get_db $sb_inst -depth {1 1} -invert -if \
    {.module =~ *_tessent_mbist*_controller}]
}

# Synthesis Step 3: Synthesizing Your Design
syn_generic
syn_map
syn_opt
# Synthesis Step 4: Writing Out Your Final SDC
write_sdc $design_name > ${design_name}.merged_sdc
# Synthesis Step 5: Writing Out Your Final Netlist
write_hdl $design_name > $design_name.vg

```

Example PrimeTime STA Script

The following section provides an example STA script to use with PrimeTime.

```
# RunAndReport
#
proc RunAndReport { mode } {
    update_timing
    redirect violations_${mode}.report {report_constraint -all_violators}
}
set tsdb ./tsdb_outdir
set design_name top
set design_id rtl
set netlist ${design_name}.vg
# Allow asserts on flop reset pins to propagate to their Q output.
set case_analysis_sequential_propagation always

# Disable timing through the enable side of clock gating cells,
# so as to prevent clock reconverging paths.
set timing_clock_gating_propagate_enable false

# Source Tessent Shell generated sdc file and set default variables
source \
    ${tsdb}/dft_inserted_designs/${design_name}_${design_id}.dft_inserted_design \
    /${design_name}.sdc
tessent_set_default_variables

# Optionally override the above TCL variables values using "set" commands. For
# example:
# set tessent_slow_clock_period 20
# set tessent_tck_period 50
# array set tessent_clock_mapping {
#     tessent_tck my_tck
#     CLK3 CLK_F300
# }

# Change your hierarchy separator variable if not appropriate
set tessent_hierarchy_separator "/"

# Override data in above "tessent_set_default_variables" by creating your
# own timing data file and placing it at the same level as this script.
if {[file exists user_timing_data.tcl]} {
    source user_timing_data.tcl
}
```

```

# You can provide your own path remapping procedures file
# (tessent_get_xxx). You only need to copy and edit the procedures you
# want to change and put them in file ./user_remapping_procs.tcl.
if {[file exists user_remapping_procs.tcl]} {
    source user_remapping_procs.tcl
}

# If you have fake cells or cells that are not part of your library
# you can provide this file to load them.
if {[file exists load_lib_cells.tcl]} {
    source load_lib_cells.tcl
}

# Create the script "load_design.pt" only if you need to load your design
# with a different command than the "read_verilog" below.
if {[file exists load_design.pt]} {
    source load_design.pt
} else {
    read_verilog $netlist
}
current_design $design_name
link_design

#####
#
#           Tessent DFT Mode
#
#####
echo "\n\nAnalysis in Tessent DFT mode.***** \n" ;
# DFT mode covers all Tessent-inserted DFT IP (ijtag network, bscan, memoryBist,
# BISR), but NOT scan or edt mode.
reset_design
LoadBackAnnotationData

# Apply constraints for STA.
tessent_create_functional_clocks
tessent_set_ijtag_non_modal
tessent_set_memory_bist_modal
tessent_set_memory_bisr_non_modal
tessent_set_jtag_bscan_non_modal
tessent_set_ltest_disable

tessent_kill_functional_paths

RunAndReport TESSENT_DFT_MODE

```

The following block is only needed for v2020.3 and earlier designs that incorporate the MemoryBIST Asynchronous Interface. Designs from v2020.4 and later incorporate the Enhanced MemoryBIST Access hardware.

```
#####  
# For v2020.3 and earlier designs only, formally check your BAP Asynchronous  
# interface timing paths  
#####  
if {[info procs tesseract_mbist_report_ai_timing] ne ""} {  
  echo "\n\nChecking your BAP asynchronous interface paths timing***** \n" ;  
  echo "See the results in output report file AITiming.report"  
  reset_design  
  tesseract_set_default_variables  
  tesseract_mbist_set_ai_timing_mode  
  update_timing  
  # set "-verbose" to 1 for a more detailed timing report  
  # change the "-tck_period" value for analysis of different shift speeds  
  redirect AITiming.report { tesseract_mbist_report_ai_timing /  
    -tck_period $tesseract_tck_period -verbose 1 }  
}
```

The following steps assume you have EDT IP in your design:

```
#####  
# Combined Edt Shift Modes  
#####  
reset_design  
LoadBackAnnotationData  
tesseract_set_ltest_modal_shift  
set_propagated_clock [all_clocks]  
RunAndReport SHIFT  
  
#####  
# EDT Slow CaptureMode  
#####  
reset_design  
LoadBackAnnotationData  
set tesseract_time_hold_in_slow_capture 1  
set tesseract_scan_en_setup_extra_cycles 1  
set tesseract_scan_en_hold_extra_cycles 1  
set tesseract_edt_update_setup_extra_cycles 1  
set tesseract_edt_update_hold_extra_cycles 1  
tesseract_set_ltest_modal_edt_slow_capture  
set_propagated_clock [all_clocks]  
RunAndReport SLOW_CAPTURE
```

```
#####
#                               EDT Fast Capture Mode
#####
# Important: The following variable setting is important in order to allow
# at-speed coverage of your Tessent OCC's internal shift register and clock gaters
# paths with scan_enable=0:
set case_analysis_sequential_propagation never
reset_design
LoadBackAnnotationData
<load your functional constraints here, including all clock definitions and
functional timing exceptions>
set memory_bypass_en_value 1
set x_bounding_en_value 1
set observe_test_point_en_value 0
tessent_set_ltest_modal_edt_fast_capture
set_propagated_clock [all_clocks]
RunAndReport FAST_CAPTURE
```

The LogicBIST modes described in the following are required when using the hybrid TK/LBIST flow.

You can execute LogicBIST tests using one of three test clock sources chosen at pattern generation time: `shift_clock_src`, `test_clock`, or `TCK`. Because `TCK` is a slow clock, STA verification is not performed explicitly when using this clock source for LogicBIST; the tool verifies all paths with one of the other two clocks, except for the clock network differences prior to the LogicBIST controller.

The default clock is `shift_clock_src`, which is typically connected to an internally generated, free-running clock because this is the most suitable source for running LogicBIST in-system test.

If you generate LogicBIST patterns to run with `test_clock` as the LogicBIST clock source, specify “`test_clock`” as an argument in the following procs:

```
tessent_set_ltest_modal_lbist_shift test_clock
tessent_set_ltest_modal_lbist_capture test_clock
```

The tool supports the values `ltest_clock` and `edt_clock` as aliases for “`test_clock`”.

If you generate two sets of LogicBIST patterns to run LogicBIST tests, execute the LogicBIST shift mode proc described in the following twice: first with the `shift_clock_src` argument and second with `test_clock` argument.

For LogicBIST shift mode and LogicBIST capture mode using the `shift_clock_src` argument, create the `shift_clock_src` clock in the timing tool and indicate its name using the Tessent SDC TCL variable `tessent_lbist_shift_clock_src`, as mentioned in “[Preparation Step 2: Setting and Redefining Tessent Tcl Variables](#)” on page 718.

```
#####  
#                               LogicBIST Shift Mode  
#####  
reset_design  
LoadBackAnnotationData  
tessent_set_ltest_modal_lbist_shift  
set_propagated_clock [all_clocks]  
RunAndReport LBIST_SHIFT
```

Unlike EDT, which uses separate fast and slow capture modes, LogicBIST uses a combined capture mode. The NCPs and fault type used during fault simulation determine whether LogicBIST test targets stuck-at or transition faults. These NCPs can use single or multiple clock pulses sourced from either the LogicBIST clock or functional clock, as determined by the NcpIndexDecoder and OCC settings. Similar to the shift mode, use either one or both of `shift_clock_src` and `test_clock` arguments for analysis. The value for `observe_test_point_en` should reflect the setting used during fault simulation. Control test points are typically enabled for both stuck-at and transition tests.

```
#####  
#                               LogicBIST Capture Mode  
#####  
reset_design  
LoadBackAnnotationData  
tessent_set_ltest_modal_lbist_capture  
<when LogicBIST test targets transition faults, load your functional constraints  
here, including all clock definitions and functional timing exceptions>  
set memory_bypass_en_value 1  
set x_bounding_en_value 1  
# When observe_test_points are disabled for transition test:  
set observe_test_point_en_value 0 ;  
set_propagated_clock [all_clocks]  
RunAndReport LBIST_CAPTURE
```

The following proc analyzes timing for the IJTAG network paths used to initialize the BIST controller and read the MISR values after test is complete. This mode uses TCK.

```
#####  
#                               LogicBIST Setup (IJTAG Network Paths)  
#####  
reset_design  
LoadBackAnnotationData  
set_propagated_clock [all_clocks]  
tessent_set_ltest_modal_lbist_setup  
RunAndReport LBIST_SETUP
```

The following proc analyzes timing for the single chain-based diagnosis mode. This mode uses TCK for reading all the scan cells through the IJTAG network.


```
#####  
#                               LogicBIST Single Chain Diagnosis  
#####  
reset_design  
LoadBackAnnotationData  
set_propagated_clock [all_clocks]  
tessent_set_ltest_modal_lbist_single_chain  
RunAndReport LBIST_SINGLE_CHAIN
```

When you implement the optional controller chain mode (CCM), the following proc analyzes timing for CCM. In this mode, the tool removes timing exceptions between the LogicBIST controller and the hybrid TK/LBIST blocks (as seen in the shift and capture modes) to reflect CCM pattern generation setup.

```
#####  
#                               LogicBIST Controller Chain Mode  
#####  
reset_design  
LoadBackAnnotationData  
set_propagated_clock [all_clocks]  
tessent_set_ltest_modal_lbist_controller_chain  
RunAndReport LBIST_CONTROLLER_CHAIN
```


Appendix A

The Tessent Tcl Interface

The Tessent Shell tool provides a Tcl-based command interface.

General Tcl Guidelines in Tessent Shell	771
Guidelines for Modifying Existing Dofiles for Use with Tcl	773
Special Tcl Characters	775
Custom Tcl Packages in Tessent Shell	777
Tcl Resources	778

General Tcl Guidelines in Tessent Shell

If Tcl procedures are available in separate files, you can source these files from within the tool or dofiles. You can also place Tcl procedures in a *.tool_startup* file so that they are available for use. Tcl file input/output is also supported.


You should be aware of the following guidelines and behaviors when using Tcl in Tessent Shell:

- You can use Tcl variables interchangeably with legacy Tessent tool variables and environment variables.
- You should use Tcl syntax for setting and referencing variables, including using `$env(ENVARNAME)` for accessing environment variables.

For example, if the value of environment variable “foo” equals “mode1” (`$foo = mode1`), you can compare this value to a Tcl variable value using the following syntax:

```
set bar mode2
if {$env(foo) == $bar} {puts Match} else {puts "No match"}
```

Note

 Use Tcl namespaces to avoid creating procedures that conflict with existing tool or Tcl commands.

- When processing comments within a dofile, the tool does not write comments preceded with “//” characters to the transcript. However, the tool does write comments that are preceded with a pound sign (#) (the Tcl comment delimiter) to the transcript.
- If a variable can contain a command string or be empty, attempting to execute by referencing `$variable` results in an error if the variable is empty.

- When a tool command error occurs, the command interpreter prints the error message and does not return anything.
- You can use the `catch_output` command to issue a specified tool command line and prevent command errors from aborting an enclosing dofile or Tcl proc. The `catch_output` command can optionally capture the output of a command or the returned result.
- When a command error occurs nested inside a Tcl construct or Tcl proc, you can obtain additional information about the error by issuing the following command:

> set errorInfo

This command prints the value of the `$errorInfo` Tcl variable, which may contain a traceback of the nested Tcl calls so that you can determine the root cause of the error.

Difference Between the Dofile Command and the Tcl Source Command

You normally use the tool's `dofile` command to execute a file of tool commands. The Tcl “source” command also executes a file of commands, but you should only use it to load Tcl procs, set Tcl variables, or do other strictly Tcl commands.

You should use the `dofile` command to execute a command file containing tool commands for the following reasons:

- The `dofile` command is affected by the `set_dofile_abort` command which provides you with the ability to specify whether the tool aborts when an error condition is detected. The Tcl “source” command is not affected by the `set_dofile_abort` command.
- The `dofile` command transcripts the commands to the shell and logfile. The Tcl “source” command always stops execution if any command in the file encounters an error.

Example 1

In this example, the `add_scan_chains` command defines every scan chain in the design. This command is sometimes used hundreds of times and makes the dofile very long. Using Tcl, you can shorten the dofile substantially by using a loop construct as shown here:

```
for {set xx 1} {$xx < 257} {incr xx} {  
    add_scan_chains int chain$xx group1 /top/edt_si$xx /top/edt_so$xx  
}
```

For the values of `xx` from 1 up to 256, the tool executes a separate `add_scan_chains` command for each value. The transcript and logfile contains all 256 `add_scan_chains` commands (preceded with “// subcommand: ”).

Example 2

The following example controls the flow of creating test patterns and uses a variable to execute different commands based on the variable's value:

```
if {$mode == stuck} {
    set_fault_type stuck
    . . .
} elseif {$mode == transition} {
    set_fault_type transition
    set_pattern_type -sequential 2
    . . .
}
```

Example 3

Module hierarchies may become ungrouped through either synthesis or layout and ATPG may operate on a flattened view of the design. Automated DRCs find and trace the EDT IP and subsequently find and trace the flattened scan chains automatically. In the following example, we are capturing the output of the `report_scan_chains` command and placing it into the variable "rsc" by using the `catch_output` command. The `foreach` loop iterates on the `report_scan_chains` output, capturing the chain name, group name and input and output connections in the variables "chain", "group", "input", and "output," respectively. This information is written into the file `add_chains.txt`.

```
set out [open "add_chains.txt" w]
catch_output {report_scan_chains} -output rsc
set rscl [split $rsc "\n"]

foreach a $rscl {
    puts "NEWLINE =\t$a"
    regexp {.*chain = (.*)\s+group = (.*)\s+input = \'(.*)\'\s+output = \
    \'(.*)\'\s+.*} $a match chain group input output
    puts $out "add_scan_chains -internal $chain $group \{$input\} \{$output\
    }"
}

close $out
```

Guidelines for Modifying Existing Dofiles for Use with Tcl

When using an existing dofile with the Tcl interface, you should evaluate the dofile for issues that could cause incorrect evaluation by the Tessent Tcl interpreter.

[Table A-1](#) provides guidance for correcting common dofile issues.

The most common issues you can run across with an existing dofile is accounting for Tcl special characters. For more information, refer to [Table A-2](#) on page 775, which provides a list of typically-used Tcl special characters.

Table A-1. Common Dofile Issues and Solutions

Dofile Issue	Solution
Stopping dofile execution at a specific point	Use the native Tcl “error” command to stop execution of a dofile at any point. The “error” command requires a message string which the tool outputs. But to avoid any message you can use an empty string: error ""
Escaping Tcl special characters	Use the following techniques to escape Tcl special characters: <ul style="list-style-type: none"> • Double quotes (" ") group tokens but enable \$variable and [command] evaluations. • Braces ({ }) also group tokens and disable \$variable and [command] evaluations. • Brackets ([]) implement command substitution and are used to nest or embed commands. • A backslash (\) escapes the next character. Use this to tame a Tcl special character such as \$, [, {, or " .
Using dollar signs in pathnames	A dollar sign (\$) specifies variable substitution. In some netlists, pathnames (for example, <i>foo/pin\$p7</i>) can contain the dollar sign. When using pathnames with dollar signs, enclose the pathname with braces ({ }) to prevent the tool from substituting the value as shown in the following example: report_gates {foo/pin\$p7}
Escaping quotation marks	In Tcl, quotation marks (" ") instruct the Tcl interpreter to treat the enclosed words as a single argument. For example: puts " Hello World " If embedded quotes are required, you must use backslashes (\) to escape the embedded double quotes. For example: puts " Hello \"World\" " Otherwise, the tool issues an error message.
Using brackets	Brackets ([]) implement command substitution and are used to nest or embed commands. A command and its arguments enclosed in square brackets is evaluated and its result inserted in place in the enclosing command.

Table A-1. Common Dofile Issues and Solutions (cont.)

Dofile Issue	Solution
Optional single quotes	<p>Optional single quotes (' ') are no longer valid for Tessent commands. For example, the following produces an error:</p> <pre>set_design_sources '-v MODB.v -v MODC.v'</pre> <p>Correct this by using no quotes, double quotes, or braces:</p> <pre>set_design_sources -v MODB.v -v MODC.v</pre> <pre>set_design_sources "-v MODB.v -v MODC.v"</pre> <pre>set_design_sources {-v MODB.v -v MODC.v}</pre>
Environment variables	<p>You should use Tcl syntax for setting and referencing variables, including using <code>\$env(ENVARNAME)</code> for accessing environment variables. For example, if the value of environment variable “foo” equals “model” (<code>\$foo = model</code>), you can compare this value to a Tcl variable value using the following syntax:</p> <pre>set bar mode2 if {\$env(foo) == \$bar} {puts "Match"} else {puts "No match"}</pre>

Special Tcl Characters

In Tcl scripts, you often see characters used for special purposes. For a complete list of special characters, you should consult a Tcl resource.

[Table A-2](#) lists and describes the more common special characters you can encounter when reading the examples in this manual. See the additional resources described in [“Tcl Resources”](#) on page 778.

Table A-2. Common Tcl Characters

Character	Description
;	The semicolon terminates the previous command, enabling you to place more than one command on the same line.
\	Used at the end of a line, the backslash continues a command on the following line.
\ \\$	The backslash with other special characters, like a dollar sign, instructs the Tcl interpreter to treat the character literally.
\ \n	The backslash with the letter “n” instructs the Tcl interpreter to create a new line.
\$	The dollar sign in front of a variable name instructs the Tcl interpreter to access the value stored in the variable.

Table A-2. Common Tcl Characters (cont.)

Character	Description
[]	Square brackets group a command and its arguments, instructing the Tcl interpreter to treat everything within the brackets as a single syntactical object. You use square brackets to write nested commands. For example: <pre>set chain_report [report_scan_chains -subchains -verbose]</pre>
{ }	Curly braces instruct the Tcl interpreter to treat the enclosed words as a single string. The Tcl interpreter accepts the string as is, without performing any variable substitution or evaluation. For example, to create a string that contains special characters such as \$ or \: <pre>set my_string {This book costs \$25.98.}</pre>
" "	Quotes instruct the Tcl interpreter to treat the enclosed words as a single string. However, when the Tcl interpreter encounters variables or commands within string in quotes, it evaluates the variables and commands to generate a string. For example, to create a string that displays a final cost calculated by adding two numbers: <pre>set my_string "This book costs \[\$expr \$price + \$tax]"</pre>

Table A-2. Common Tcl Characters (cont.)

Character	Description
#	<p>The pound sign (#) indicates a comment and directs the Tcl compiler to not evaluate the rest of the line. When using the pound sign, you must use it where a command starts, and at the beginning of a command, not within a command. Be aware of the following:</p> <ul style="list-style-type: none"> • Evaluate does not equal parse. Despite the pound sign, the following comment gives an error because Tcl detects an open lexical clause. <pre data-bbox="391 533 821 646"># if (some condition) { if { new text condition } { ... }</pre> <ul style="list-style-type: none"> • The apparent beginning of a line is not always the beginning of a command: <pre data-bbox="391 737 691 764"># This is a comment</pre> <p>In the following code snippet, the line beginning with “# -type” is not a comment, because the line immediately above it has a line continuation character (\). In fact, the “#” confuses the Tcl interpreter, resulting in an error when it attempts to create the tk_messageBox.</p> <pre data-bbox="391 997 1365 1079">tk_messageBox -message "The diagnosis report was successfully written." \ # -type ok</pre> <p># and this is also a comment. This one spans \ multiple lines, even without the # at the beginning \ of the second and third lines.</p> <p>In general, it is good practice to begin all comment lines with a #. Be aware that the beginning of a command is not always at the beginning of a line. Usually, you begin new commands at the beginning of a line. That is, the first non-space character is the first character of the command name. However, you can combine multiple commands into one line using the semicolon “;” to designate the end of the previous command:</p> <pre data-bbox="391 1470 1247 1522">set myname "John Doe" ; set this_string "next command" set yourname "Ted Smith" ; # this is a comment</pre>

Custom Tcl Packages in Tessent Shell

Tessent Shell supports several standard techniques for adding your own Tcl packages with the Tcl “package require” command, which you can issue from Tessent Shell.

You can use any of the following ways to specify directory locations of Tcl packages. Any directory that contains a Tcl package must also contain a *pkgIndex.tcl* file within its hierarchy.

The following methods are listed in order of the precedence in effect if there is more than one package with the same name:

- Default location for Tessent Shell Tcl packages. You can place your package underneath a directory named *tessent_plugin/packages* that is located at the top of your Tessent install tree. When Tessent Shell finds a package in this directory upon invocation, it appends the directory path to *tessent_plugin/packages* to the `auto_path` Tcl global variable, if it exists.
- `TESSENT_PLUGIN_PATH` environment variable. You can set this variable to a colon-separated list of directory paths that contain Tcl packages in a *packages* subdirectory. When Tessent Shell finds a package, it appends the directory path to *packages* to the `auto_path` Tcl global variable, if it exists.
- `auto_path` Tcl global variable. You can issue the following command in Tessent Shell to specify a package location:

```
> lappend auto_path pathToTclPackageDir
```

- `TCLLIBPATH` environment variable. You can set this variable to a space-separated list of directory paths that contain Tcl packages. A *packages* subdirectory is not required under any of the paths.

Note



Tessent Shell ignores the `TCL_LIBRARY` environment variable.

Tcl Resources

The following website is a place to start in your search for the reference material that works best for you. It is not an endorsement of any book or website.

<http://www.tcl.tk/>

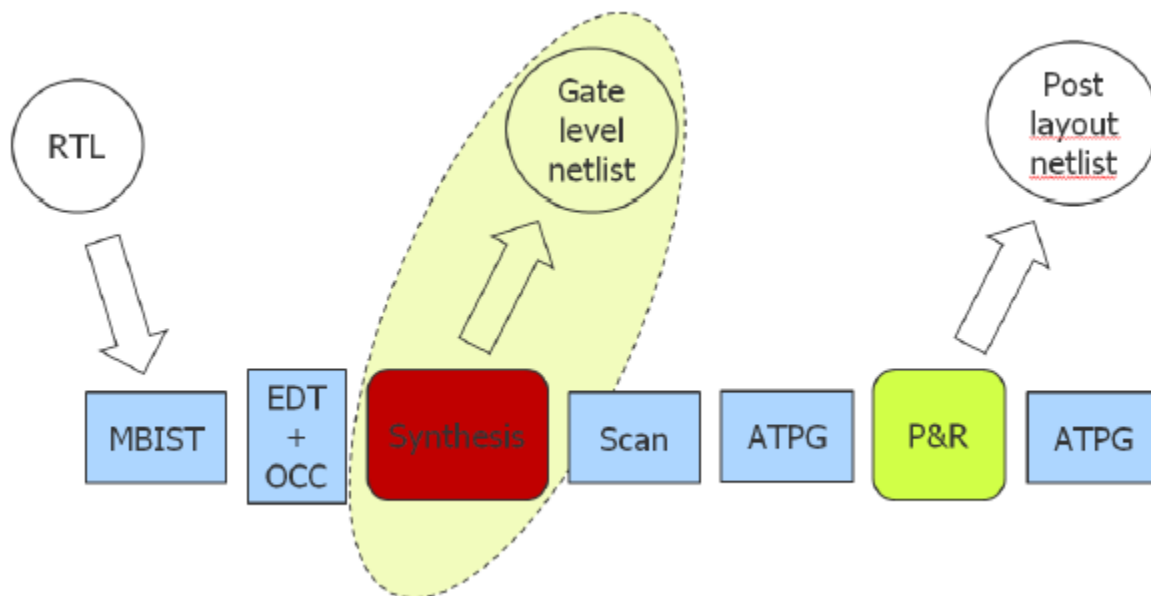
Appendix B

Synthesis Guidelines for RTL Designs with Tessent Inserted DFT

This appendix provides guidelines for you to use when synthesizing your design with Tessent created IP RTL with a third-party synthesis tool, specifically Synopsys DC Ultra™.

You may experience problems with synthesis optimization using DC Ultra caused by propagating constants even if the boundary optimization is false. This synthesis optimization results in some of the DFT logic that is not connected until scan insertion being optimized away, and the ports being kept. This creates a situation where some of the necessary Tessent logic is eliminated during optimization, which breaks the design.

Figure B-1. Problem Occurrence Creating the Gate Level Netlist



This problem does not appear to be present with the basic DC compile command; it seems to only be an issue when using the compile_ultra command in certain tool versions.

DFT Insertion With Tessent	780
Synthesis Guidelines	781
SystemVerilog and Port Name Matching	783

DFT Insertion With Tessent

When the Tessent tools create the IP logic to add to the design, the logic includes some persistent instances that are used as anchor points for items such as clocks, control signals, constraints, and so on. For automation, the tool uses modules and port matching, and, consequently, requires that some specific modules be preserved after synthesis and place and route.

There are two types of persistent instances for synthesis:

- Cells
 - Requires the use of a `set_size_only` command
- Design Modules
 - Requires `ungroup` to be set to off
 - Requires boundary optimization to be set to off

Boundaries of all instances of modules with Tessent Core Descriptions (TCDs) need to be preserved, which means the following:

- No boundary optimization.
- No ungrouping.
- No new ports.
- No logic optimization.

The logic test module types that require this include EDT, LBIST, OCC, STI SIB, Bscan interface, and any module with `tcd_scan` (pre-existing scan chain). ICL extraction uses modules and ports of ICL modules so they also need to be preserved. You can avoid preserving the IJTAG instance boundaries, which enables the synthesis tool to obtain a better optimization result. You can use the ICL file from RTL in post synthesis and place and route.

The boundary optimization option during synthesis is global in the sense that it applies to every design hierarchy on and below the specified module or instance. Hence, if you want to globally optimize your design, you specify it on the root (top) module and then boundary optimization runs through the entire design doing its work (assuming that there are no “don’t touch” cores, which were already synthesized earlier). The problem is that valid constrained or not (yet) connected ports or nets get optimized away. To prevent this, tell the DC tool not to apply this boundary optimization to selected modules or instances.

DC has two compilation modes:

- Basic compile (using the `compile` command):
 - Problems are avoided because the tool does not ungroup and optimize boundaries.

- Constants are not optimized away.
- Optimization On (using `compile_ultra` command):
 - The synthesis tool does its best to remove any dead logic (either because of constraints propagation or missing loads or unused).
 - DC propagates constraints even if boundary optimization is set false for design instances.

As this issue is becoming more prevalent, there is an informational message from DC Ultra when using the `compile_ultra` command that reports the following:

```
Information: Starting from 2013.12 release, constant propagation is
enabled even when boundary optimization is disabled. (OPT-1318)
```

If you receive this message, you may not fully understand the ramifications or how to correctly synthesize the Tessent IP in this environment. If not properly handled, you may see additional information messages from the tool similar to the following:

```
Information: Removing unused design 'corea_rtl_tessent_occ_shift_reg_1'.
(OPT-1055)
Information: The register
'corea_rtl_tessent_occ_clk_inst/occ_control/scan_out_reg' is a constant
and will be removed. (OPT-1206)
Information: Ungrouping hierarchy tessent_persistent_cell_edt_clock
before Pass 1 (OPT-776)
```

Synthesis Guidelines

When synthesizing a design for Tessent DFT, certain considerations must be taken into account to avoid issues later in the flow.

When using Cadence Genus, set the attribute `hdl_flatten_complex_port` to “true” for any designs that include complex ports declared using unions to facilitate post-synthesis port name matching.

When using any of the Synopsys Design Compiler family of synthesis tools, declare the ranges of any ports declared as SystemVerilog interface arrays be declared as follows to facilitate post-synthesis port-name matching:

```
<interface_type> <port_name> [0:<positive right index>]
```

When using DC Ultra, there are two key commands to be aware of to synthesize correctly with the Tessent IP.

- Turn off constant propagation with no boundary optimization by setting an application variable. This variable works globally and is used with the following syntax:

```
set_app_var compile_enable_constant_propagation_with_no_boundary_opt false
```

- For instance level control, use the following compile directive before using the `compile_ultra` command to turn off the constant propagation for cells:


```
set_compile_directives -constant_propagation false [get_cells <hier-cell-name>]
```

Or for pin-level granularity, use the following command:

```
set_compile_directives -constant_propagation false [get_pins <hier-pin-name>]
```

Other means of restricting the synthesis tool from changing and optimizing instances and levels of hierarchy include the `set_size_only` command and ungrouping. When `set_size_only` is used for a specified list of leaf cells, the tool can only change their drive strength, or sizing. The `ungroup`, `group`, `set_ungroup`, and `uniquify` commands can be used to control how the tool removes (or not) levels of hierarchy and how reused blocks are uniquified during synthesis. The `dont_touch` attribute is also effective for adding to designs, sub designs, and cells to prevent them from being ungrouped during optimization.

Note

 During the synthesis of MemoryBIST logic, there are two types of warnings that may be issued by synthesis tools that are related to the optimization of registers. These warnings may be ignored as synthesis tools are very reliable. Additionally, formal verification can be used to confirm the functionality is not affected. The warning types are:

- Registers with no fanout are removed, along with the combinatorial logic driving the inputs.
 - Registers with inputs and outputs that are always identical are merged.
-

Different guidelines apply to the type of flow you are using, whether a bottom-up synthesis flow, or a top-down approach.

Bottom-Up Flow Using DC Ultra

1. Read full design in synthesis tool.
2. Set current design to each DFT IP and compile.
3. Set don't touch or set size only to all the DFT IP.
4. Set current design TOP.
5. Read functional SDC.
6. Read DFT SDC.
7. Set ungroup false to all DFT IP and persistent modules.
8. Set boundary optimization false to all DFT IP and persistent modules.

9. Set size only to all persistent cells.
10. Disable constant propagation during boundary optimization.
11. Compile ultra.

Top-Down Flow Using DC Ultra

1. Read full design in synthesis tool.
2. Read functional SDC.
3. Read DFT SDC.
4. Set ungroup false to all DFT IP and persistent modules.
5. Set boundary optimization false to all DFT IP and persistent modules.
6. Set size only to all persistent cells.
7. Disable constant propagation during boundary optimization.
8. Compile ultra.

SystemVerilog and Port Name Matching

Any ports declared as interface arrays in a SystemVerilog RTL design should be declared as follows:

```
<interface_type><port_name> [0:N]
```

“N” is a positive integer that represents the size of the array minus 1. You can use this method to properly map the port names in a netlist generated by any of the Design Compiler family of tools.

Appendix C

Clocking Architecture Examples

The clocking architecture in designs may vary, and they play a role during test planning, especially for hierarchical test.

This appendix provides examples of common clocking architectures and how to insert the on-chip clock controllers (OCCs) given those architectures. The examples assume that you are performing hierarchical test as described in “[DFT Architecture Guidelines for Hierarchical Designs](#)” on page 95”.

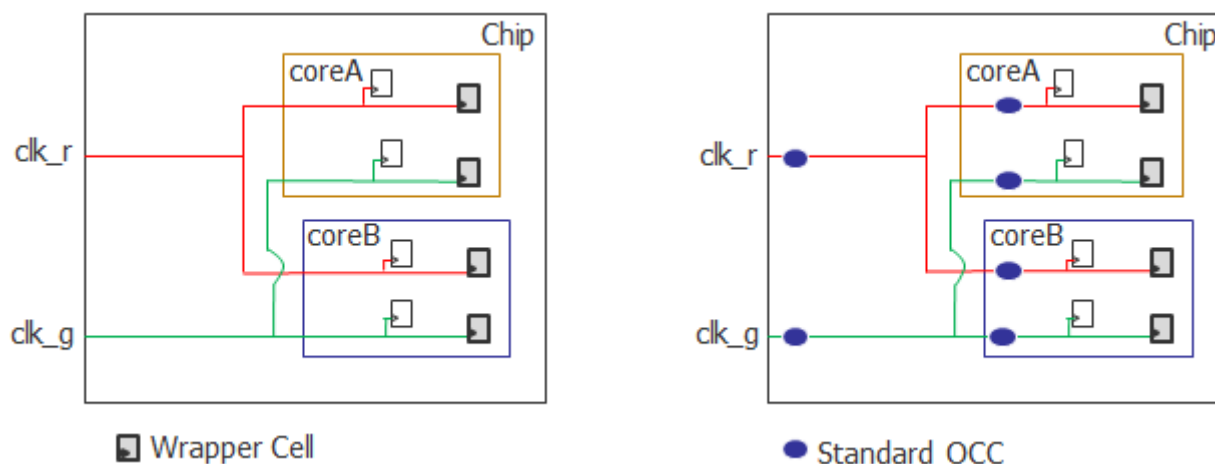
For information about the OCCs supported by Tessent, refer to “[Tessent On-Chip Clock Controller](#)” in the *Tessent Scan and ATPG User’s Manual*.

Clocks Driven by Primary Inputs	785
Clock Generators Outside the Cores	786
Clock Generators Inside the Cores	786
Clock Sourced From A Core With Embedded PLL	787
Clock Mesh Synthesis	788

Clocks Driven by Primary Inputs

In the following figure, coreA and coreB are wrapped cores, and the clk_r and clk_g clocks are asynchronous to each other. Insert standard OCCs inside coreA and coreB for both clk_r and clk_g. At the chip-level, insert standard OCCs on the clk_r and clk_g ports.

Figure C-1. OCCs for Clocks Driven by Primary Inputs

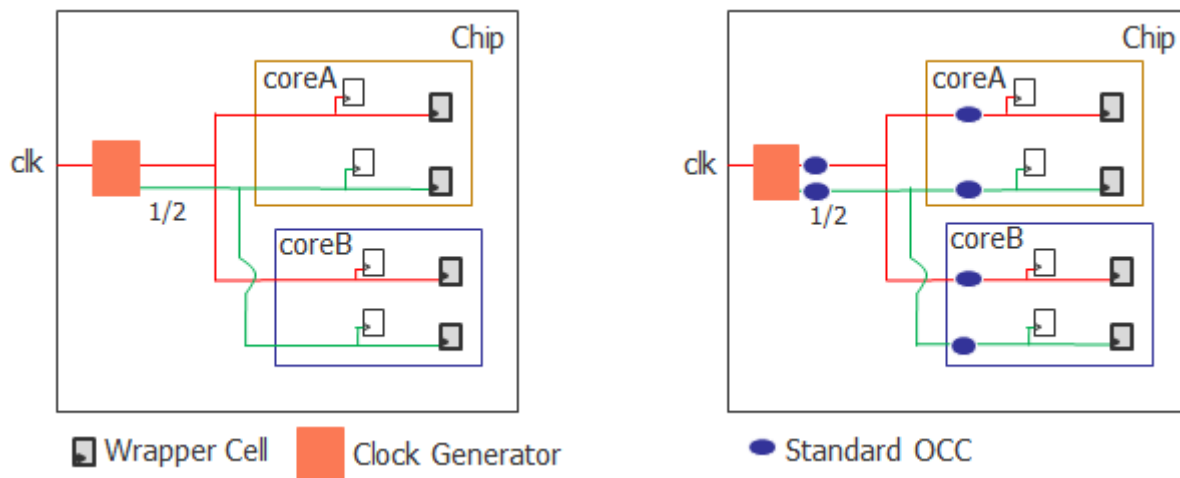


During intest mode for the wrapped cores, the OCCs inside the cores are active and the OCCs outside the cores are inactive. During extest, the OCCs outside the cores are active and the OCCs inside the cores are inactive.

Clock Generators Outside the Cores

In the following figure, coreA and coreB are wrapped cores, and a clock generator at the chip level divides the clock frequency by half to generate the green clock. The red and green clocks are asynchronous to each other. Insert standard OCCs inside coreA and coreB for both the red and green clocks. At the chip-level, insert standard OCCs on both the red and green clocks generated at the output of the clock generator.

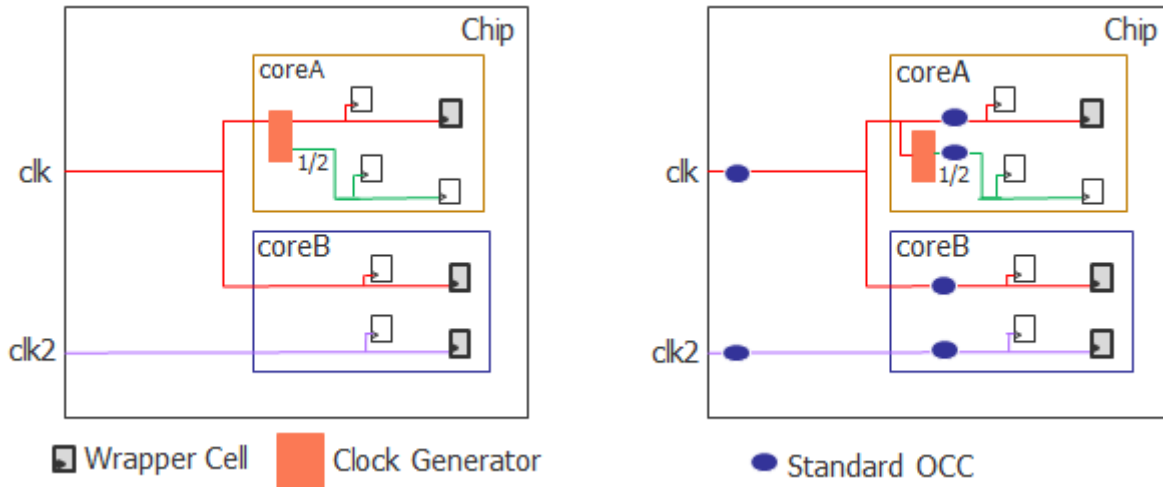
Figure C-2. OCCs With Clock Generators at Chip Level, Asynchronous Clocks



Clock Generators Inside the Cores

In the following example, coreA and coreB are wrapped cores and a clock generator is located inside coreA. The red and green clocks are asynchronous to each other inside coreA, so you would insert standard OCCs inside coreA. In addition, the red and purple clocks are asynchronous to each other, so you insert standard OCCs for these clocks inside coreB as well as at the chip-level.

Figure C-3. OCCs With Clock Generators Inside the Cores, Asynchronous Clocks

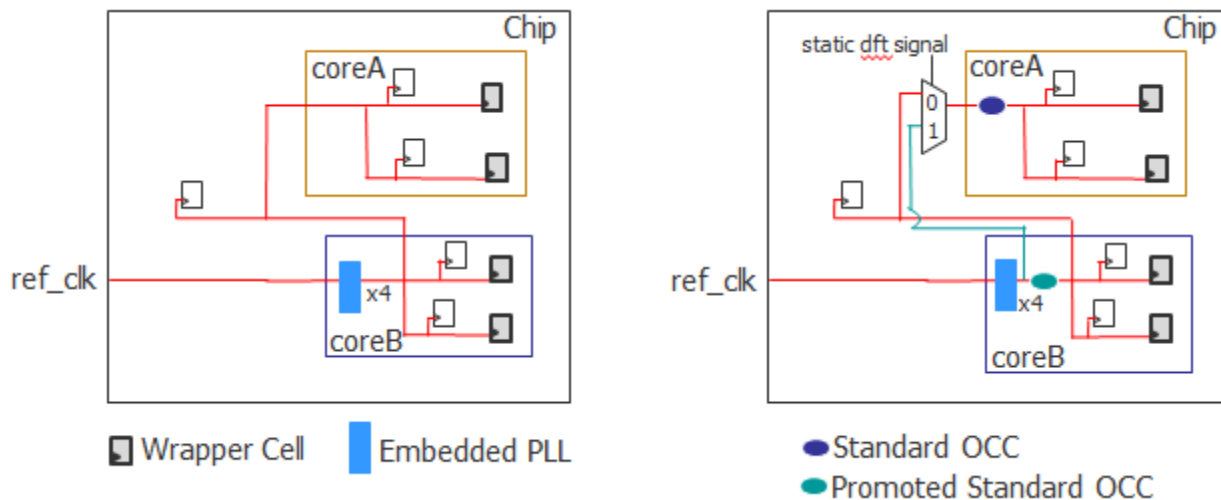


During intest mode for the wrapped cores, the OCCs inside the cores are active, and the OCCs outside the cores are inactive. During extest, the OCCs outside the cores are active, and the OCCs inside the cores are inactive.

Clock Sourced From A Core With Embedded PLL

In the following figure, coreA and coreB are wrapped cores, and coreB contains an embedded PLL module. coreB sources the clock that feeds both coreA and the chip. In this case, you would retarget coreA and coreB at the same time.

Figure C-4. Clock Sourced From A Core With Embedded PLL, With MUX



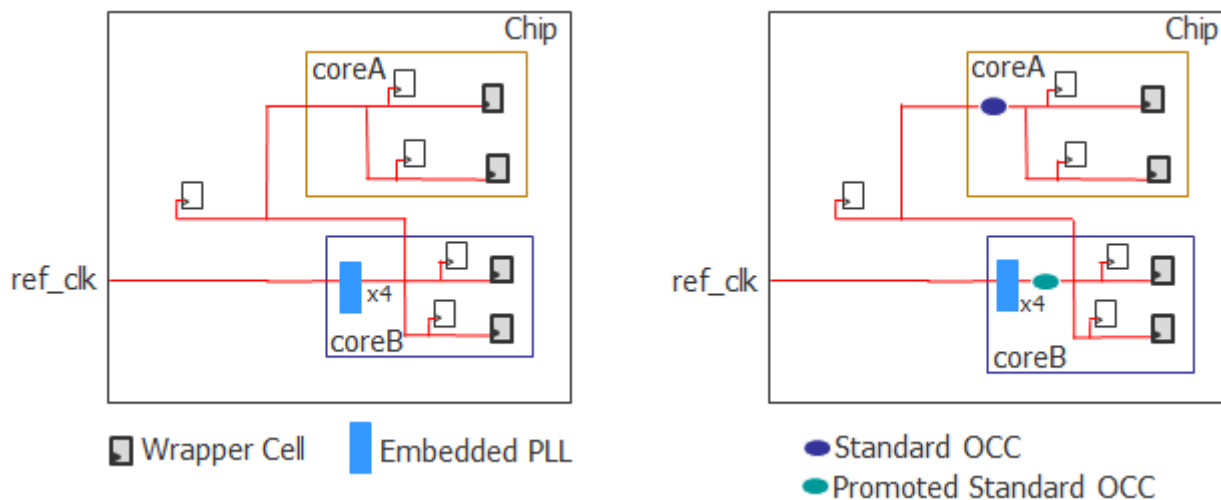
When you are retargeting coreA and coreB at the same time, you must add a mux. Use a clock net before reaching the OCC inside coreB as a clock to feed the OCC inside coreA to avoid having back-to-back OCCs. To retarget both coreA and coreB at the same time, the OCCs inside coreA and coreB must be active at the same time.

Insert standard OCCs inside the cores. The standard OCC inside the core with the embedded PLL (coreB) gets promoted so that it is also included in the external chains. When in external mode, the OCC inside coreB can be reused. For details, refer to “[How to Handle Clocks Sourced by Embedded PLLs During Logic Test](#)” on page 494.

During intest, the standard OCCs inside coreA and coreB are active. The mux is set to 1 to avoid cascading OCCs. During extest, only the promoted standard OCC within coreB is active, and the mux is held at 0.

In the following example, you are retargeting coreA and coreB in separate runs, so there is no need to insert a mux. During intest of coreA, only coreA’s OCC is active, and then during intest of coreB, only coreB’s OCC is active. During extest of the cores, only the promoted standard OCC within coreB is active.

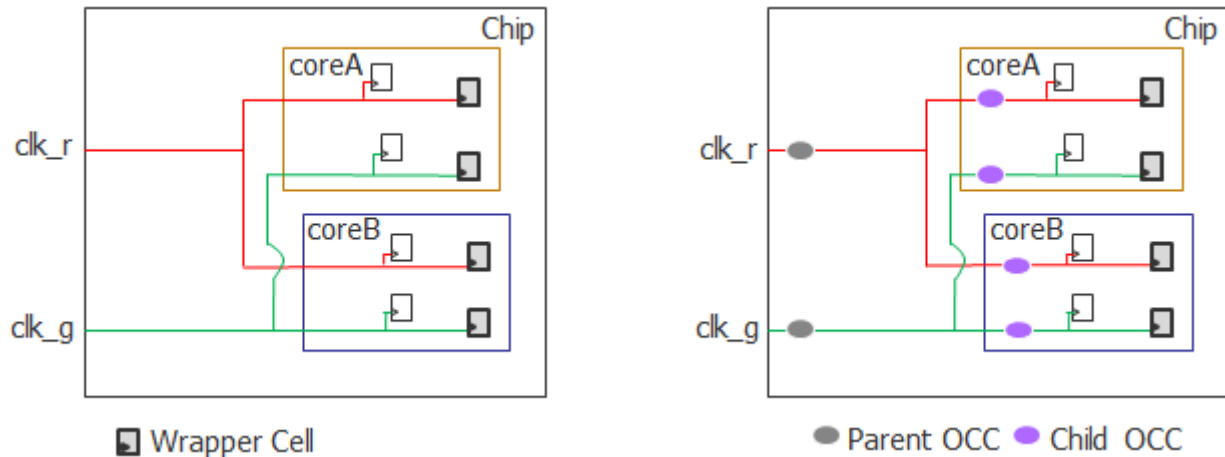
Figure C-5. Clock Sourced From A Core With Embedded PLL, Without MUX



Clock Mesh Synthesis

The most common method for clock balancing is clock tree synthesis. However, you can also use clock mesh synthesis to balance the clocks. For clock mesh, rather than standard OCCs, use parent OCCs at the chip level and child OCCs inside the wrapped cores. In the following figure, coreA and coreB are wrapped cores, and the clk_r and clk_g clocks are asynchronous to each other.

Figure C-6. Clock Mesh Synthesis



Core-level child OCCs have additional timing requirements because a single clock on the boundary of the wrapped core is used for shift as well as for slow and fast capture. For details, refer to “[Core OCC Recommendation](#)” in the *Tessent Scan and ATPG User’s Manual*. Using SDC for timing closure is described in “[Timing Constraints \(SDC\)](#)” on page 715.

During intest of coreA and coreB, the child OCCs are active, and the parent OCCs are in parent mode. During extest of coreA and coreB, the parent OCCs are in standard mode and the child OCCs are inactive.

Appendix D

Tessent Visualizer Keyboard Shortcuts

A number of keyboard shortcuts are available for Tessent Visualizer.

Table D-1. Global Shortcuts

Shortcut	Action
Ctrl+Shift+H	Open Hierarchical Schematic
Ctrl+Shift+F	Open Flat Schematic
Ctrl+Shift+B	Open Instance Browser
Ctrl+Shift+W	Open Waveform Generator
Ctrl+Shift+L	Open Cell Library Browser
Ctrl+Shift+D	Open DRC Browser
Ctrl+Shift+P	Open Pin Data
Ctrl+Shift+V	Open Text/HDL Viewer
Ctrl+Shift+R	Open Diagnosis Report Viewer
Ctrl+Shift+T	Open Transcript
Ctrl+Shift+Q	Quit Tessent Visualizer
Ctrl+Shift+W	Close Current Tab
Ctrl+Shift+Tab or Ctrl+PgUp	Go To Previous Tab
Ctrl+Tab or Ctrl+PgDown	Go To Next Tab

Table D-2. Schematic Shortcuts

Shortcut	Action
Ctrl+0	Zoom all
Ctrl++	Zoom in
Ctrl+-	Zoom out
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo
Ctrl+A	Select all

Table D-2. Schematic Shortcuts (cont.)

Shortcut	Action
Del	Delete selected
Ctrl+M	Mark with chosen color
Ctrl+S	Export schematic (also: export table)
Arrow Keys	Move schematic
Esc	Clear selection
Ctrl+R	Report gates

Table D-3. Instance Browser Shortcuts

Shortcut	Action
Enter	Go down in hierarchy
Backspace	Go up in hierarchy
Arrow Keys	Move to the chosen cell


Table D-4. Filter Editing Shortcuts

Shortcut	Action
Tab	Next filter
Shift+Tab	Previous filter
Enter	Accept filters
Esc	Discard changes to the current filter

Table D-5. Text Search Shortcuts

Shortcut	Action
Ctrl+F	Open text find function
Esc	Close text find function
Ctrl+G	Find next
Ctrl+Shift+G	Find previous

Note

 The Ctrl+G and Ctrl+Shift+G keyboard shortcuts are not supported in all desktop environments.

Appendix E

Formal Verification

Tessent™ Shell-based products currently do not generate scripts for use with Synopsys® Formality® or Cadence® Conformal®. You can, however, set constraints in your design that are used with these tools.

Constraints for Formality Scripts	793
Constraints for Conformal Scripts	794

Constraints for Formality Scripts

Use the following guidance to create a script for use with Formality.

Golden RTL Versus DFT-Inserted RTL

Set the following:

- Set the circuit to functional mode in order to hold the IJTAG network or TAP in reset.
- If you are at the chip level, set a constant 0 on TRST. If you are at the sub_block or physical_block level, set a constant on the ijtag_reset ports to their active value. This is typically 0.
- If your design has DftSignals coming from ports in both the chip and design levels, these ports must be held at their reset values. The reset automatically handles DftSignals from the TDR.
- If you are at the sub_block or physical_block level, suppress the verification of ports created by Tessent Shell to prevent reporting violations.
- Assert the ijtag_reset pins to their active values on all instruments and SIBs inserted by Tessent tools.
- Because the ICL network may already be present, the SIBs could result in mismatches on the ICL network scan path. Set the SIB scan in/outs as “don’t verify” point.
- Use the “Consistency” mode for the verification passing mode rather than the “Equality” mode by setting the following:

```
set verification_passing_mode Consistency
```

The default verification passing mode of Formality is the “Consistency” mode.

- If you want to use the “Equality” mode for verification, add the following setting to deal with clock gating logic inserted by Tessent:

```
set_app_var verification_clock_gate_hold_mode any
```

Pre-Layout Versus Post-Layout

Ensure that the layout step did not affect DFT functionality. You should ignore violations caused by scan chain reordering. Also set the scan_en signal to a constant low, but do not constrain anything else on the inserted DFT.

After scan chain reordering, during layout, you may see mismatches on the lockup latches that Tessent Shell adds that become stop/anchor points in the scan DEF file. That is, during Tessent Shell scan insertion, lockup elements are inserted at the end of the scan chains and marked as stop points in the scan DEF file, which means that they are not reordered.

```
GPIO_2/\p2out_reg[1] ( IN SI ) ( OUT Q )  
+ STOP GPIO_2/ts_1_lockup_latchn_clkc6_intno266_i D\
```

When this occurs and you perform formal verification on the pre- versus post-layout, the tool issues warnings that these lockups are no longer connected to the same functional flop as previously. This is not an issue with the other scan flops because you can disable the scan path by setting the scan enable to 0. However, the lockup is a flop/latch without scan-in.

If you encounter this situation, use the following workaround: Specify set_dont_verify_point for the retiming flops in both the reference and implemented designs to see if Formality passes. To find the retiming flops, introspect the design in Tessent Shell. For example:

```
get_instances ts_1_lockup*
```

EDT

Constraining scan_en to 0 is sufficient to proceed with equivalence checking. The formal verification tool reports some unmatched nodes like the newly added EDT pins (*clock/update/bypass/low_power/channel*) and registers inside the EDT logic.

OCC

If the OCC has an IjtagInterface, or if the test_mode pin of the OCC is connected to an IjtagNetwork, then keeping the ijtag network in reset state is sufficient for OCC. If the test_mode port is connected to a top-level port, this port needs to be constrained to 0.

Constraints for Conformal Scripts

Use the following guidance to create a script for use with Conformal.

Embedded Boundary Scan at the Physical Block Level

Most signals are blocked by `bscan_select`, but you should also set the following explicitly:

- `add_pin_constraints 0 bscan_select -golden`
- `add_pin_constraints 0 bscan_select -revised`
- `add_ignored_outputs bscan_scan_out -golden`
- `add_ignored_outputs bscan_scan_out -revised`
- `add_pin_constraints 0 bscan_force_disable -both`
- `add_pin_constraints 0 bscan_select_jtag_output -both`
- `add_pin_constraints 0 bscan_clock -both`
- `add_pin_constraints 0 bscan_select_jtag_input -both`

Appendix F

Transitioning from the Classic Point Tools

This appendix provides information to help you make the optional transition from the classic Tessent point tools to Tessent Shell. The point tool application commands are forward-compatible, so your dofiles work properly in Tessent Shell with only minor modifications. The main differences involved in using Tessent Shell is that you have to set the context before doing anything else, and then you must load the netlist and library.

The classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Siemens Digital Industries Software recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell continues to add new features that are not available in the classic point tools.

For information about specific point tools refer to the following sections:

Transitioning from the Classic FastScan Point Tool	797
Transitioning from the Classic TestKompress Point Tool	798
Transitioning from the Classic DFTAdvisor Tool	799
Transitioning from the Classic Diagnosis Tool	800

Transitioning from the Classic FastScan Point Tool

Beginning with the v2012.3 release, you can optionally transition from the classic FastScan point tool to Tessent Shell. Tessent Shell provides all of the commands available in the classic FastScan point tools (invoked with the “fastscan” shell command) plus additional features.

The classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Siemens Digital Industries Software recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell continues to add new features that are not available in the classic point tools.

If you are already familiar with classic FastScan, the main differences involved in using Tessent Shell is that you have to set the context before doing anything else, and then you must load the Verilog netlist and library:

```
set_context patterns -scan
read_verilog netlist_name
read_cell_library library_name
set_current_design
```

Another difference with using Tessent Shell is that several system modes used with classic FastScan (atpg, good, fault) have been replaced with a single system mode (called analysis). This means that “set_system_mode atpg” invocations (as well as transitions to good or fault modes) are replaced by “set_system_mode analysis.” To facilitate your transition to Tessent Shell, the set_system_mode command continues to accept the atpg/good/fault arguments, but the tool actually switches to analysis mode.

Also, it is highly recommended for any simulation to replace the old “set_pattern_source external” and “run” commands with the new read_patterns and simulate_patterns commands. Unlike the “run” command, which has slightly different behavior in the good/fault/atpg system modes, the simulate_patterns command enables you to explicitly control which pattern set to simulate and which patterns (if any) to store in the internal pattern set.

Transitioning from the Classic TestKompress Point Tool

Beginning with the v2012.3 release, you can optionally transition from the classic TestKompress point tool to Tessent Shell. Tessent Shell provides all of the commands available in the classic point tool (invoked with the “testkompress” shell command) for EDT IP creation and test pattern generation plus additional features.

The classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Siemens Digital Industries Software recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell continues to add new features that are not available in the classic point tools.

The following list describes how Tessent Shell is different from the classic TestKompress tool for creating EDT IP:

- Before issuing any commands, set the context to “dft -edt”:

```
set_context dft -edt
```

Once you have set the context, all commands in setup mode of classic TestKompress are available in the setup mode of Tessent Shell. All commands available in atpg mode of classic TestKompress (IP creation phase) are available in analysis mode of Tessent Shell.

- Use the existing `write_edt_files` command to generate all EDT files, including RTL, dofiles, the test procedure file, and synthesis and timing verification scripts. You can issue this command only when in analysis mode. The `write_edt_files` command does not write out the EDT IP netlist, so you must do this with the `write_design` command.
- For internal IP location flow-based EDT insertion with classic TestKompress, the tool writes out all EDT files, inserts EDT IP into the design, and automatically changes to insertion mode. You then have the ability to further edit the netlist, but you must explicitly save changes with the `write_design` command.

Unlike classic TestKompress, Tessent Shell does not automatically exit after executing a `write_edt_files` command, so you must explicitly exit when ready. This is so that after IP insertion into the design, you can perform further design editing before writing out the design, or configure how to write out the design using the `write_design` command. Also, Tessent Shell does not write out the netlist as part of the `write_edt_files` command, so you must write out the netlist using the `write_design` command.

- The classic TestKompress command `write_edt_files` has a switch named “-insertion tk.” In Tessent Shell, this switch is renamed “-insertion ts.”

The following list describes how Tessent Shell is different from the classic TestKompress tool for generating patterns:

- Before generating compressed patterns, set the context to “patterns -scan”:

`set_context patterns -scan`

Once you’ve set the context, all commands in the setup mode of classic TestKompress are available in setup mode of Tessent Shell. And all commands available in the atpg mode of classic TestKompress (Test Pattern Generation phase) are available in the analysis mode of Tessent Shell.

Transitioning from the Classic DFTAdvisor Tool

Beginning with the v2012.3 release, you can optionally transition from the classic DFTAdvisor point tool to Tessent Shell. Tessent Shell and Tessent Scan provide all of the commands available in the classic DFTAdvisor tool (invoked with the “dftadvisor” shell command), plus additional features.

The classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Siemens Digital Industries Software recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell continues to add new features that are not available in the classic point tools.

The following lists the differences between the classic DFTAdvisor tool and Tessent Scan:

- You no longer need to use the “run” command. Analysis that was previously done using the “run” command is now done automatically after DRC or as part of the [analyze_wrapper_cells](#) and [insert_test_logic](#) commands.
- After issuing `insert_test_logic` in analysis mode, the tool updates the design and then changes to insertion mode. This enables you to optionally perform further design editing before writing out the netlist with the `write_design` command.
- There is no longer a system mode called Dft. Use analysis mode instead.
- Before you issue any of the classic DFTAdvisor commands, you must first set the context:

```
set_context dft -scan
```

Transitioning from the Classic Diagnosis Tool

Beginning with the v2012.3 release, you can optionally transition from the classic Diagnosis point tool to Tessent Shell. Tessent Shell provides all of the commands available in the classic Diagnosis tool (invoked with the “tessent -diagnosis” shell command), plus additional features.

The classic Tessent point tools is available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Siemens Digital Industries Software recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell continues to add new features that are not available in the classic point tools.

After invoking Tessent Shell with the “tessent -shell” command, at a minimum, you must perform the following operations in sequence to enable scan diagnosis in Tessent Shell:

1. Set the context before doing anything else:

```
set_context patterns -scan_diagnosis
```

2. Read the flat model of your design:

```
read_flat_model flat_model_name
```

Previously, you specified the flattened design with the “tessent -diagnosis” command. Now you must use the [read_flat_model](#) command.

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Siemens EDA Support.

The Tessent Documentation System	801
Global Customer Support and Success	802

The Tessent Documentation System

At the center of the documentation system is the InfoHub that supports both PDF and HTML content. From the InfoHub, you can access all locally installed product documentation, system administration documentation, videos, and tutorials. For users who want to use PDF, you have a PDF bookcase file that provides access to all the installed PDF files.

For information on defining default HTML browsers, setting up browser options, and setting the default PDF viewer, refer to the *Siemens® Software and Mentor® Documentation System* manual.

You can access the documentation in the following ways:

- **Shell Command** — On Linux platforms, enter `mgcdocs` at the shell prompt or invoke a Tessent tool with the `-manual` invocation switch.
- **File System** — Access the Tessent InfoHub or PDF bookcase directly from your file system, without invoking a Tessent tool. For example:

HTML:

```
firefox <software_release_tree>/doc/infohubs/index.html
```

PDF:

```
acroread <software_release_tree>/doc/pdfdocs/_tessent_pdf_qref.pdf
```

- **Application Online Help** — You can get contextual online help within most Tessent tools by using the “`help -manual`” tool command. For example:

```
> help dofile -manual
```

This command opens the appropriate reference manual at the “`dofile`” command description.

Global Customer Support and Success

A support contract with Siemens Digital Industries Software is a valuable investment in your organization's success. With a support contract, you have 24/7 access to the comprehensive and personalized Support Center portal.

Support Center features an extensive knowledge base to quickly troubleshoot issues by product and version. You can also download the latest releases, access the most up-to-date documentation, and submit a support case through a streamlined process.

<https://support.sw.siemens.com>

If your site is under a current support contract, but you do not have a Support Center login, register here:

<https://support.sw.siemens.com/register>

— Symbols —

#include statement, [539](#)

— A —

Alias statements, [543](#)

Always block, [556](#)

Always block example, [557](#)

Always block statements

 always_statement, [557](#)

 force, [557](#)

 pulse, [557](#)

Always block syntax, [557](#)

— C —

Clock_run procedure, [591](#)

Cycle statements

 bidi_force_off, [565](#)

 bidi_force_pi, [564](#)

 bidi_measure_po, [564](#)

 condition, [566](#)

 expect, [566](#)

 force, [566](#)

 force_pi, [564](#)

 force_sci, [564](#)

 force_sci_equiv, [564](#)

 initialize, [566](#)

 measure, [566](#)

 measure_po, [564](#)

 measure_sco, [564](#)

 observe_method, [566](#)

 pulse, [566](#)

 pulse_capture_clock, [565](#)

 pulse_read_clock, [565](#)

 pulse_write_clock, [565](#)

 restore_bidi, [565](#)

 restore_pi, [564](#)

— E —

Environment Variables

 TESSENT_LICENSE_ORDER, [32](#)

Environment variables

 application-specific, [32](#)

 MGCDFT_STARTUP, [32](#)

external capture procedure, [593](#)

— I —

Include statement, [539](#)

— N —

Named capture procedure, [593](#)

— P —

Procedure statements

 apply, [562](#)

 cycle, [557](#)

 label, [562](#)

 scan_group, [560](#)

 timeplate, [560](#)

— S —

Set statement, [540](#)

Set statements

 default_timeplate, [542](#)

 strobe_window time, [541](#)

 time scale, [540](#)

Shift procedure, [580](#)

 alternate, [582](#)

Sub_procedure, [602](#)

— T —

Tessent Visualizer tutorials, [703](#)

Test procedure file

 defined, [534](#)

 DRC checking, [534](#)

 statements, [540](#)

 timing variables, [545](#)

Test procedures

 capture, [593](#)

 clock_po, [598](#)

 clock_sequential, [599](#)

 init_force, [599](#)

- load_unload, [583](#)
- master_observe, [588](#)
- shadow_control, [586](#)
- shadow_observe, [589](#)
- shift, [580](#)
- skew_load, [589](#)
- sub_procedure, [602](#)
- test_end, [600](#)
- test_setup, [577](#)
- Test_end procedure, [600](#)
- Time scale
 - setting, [540](#)
- Timeplate statements
 - bidi_force_pi, [549](#)
 - bidi_measure_po, [550](#)
 - force, [550](#)
 - force_pi, [549](#)
 - measure, [550](#)
 - measure_po, [550](#)
 - offstate, [549](#)
 - period, [552](#)
 - pulse, [550](#)
 - pulse_clock, [551](#)
- tscale, [540](#)
- Tutorials for Tessent Visualizer, [703](#)

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.

